DS1007 PPC Processor Board

RTLib Reference

Release 2021-A – May 2021

dSPACE

How to Contact dSPACE

Mail:	dSPACE GmbH Rathenaustraße 26 33102 Paderborn Germany
Tel.:	+49 5251 1638-0
Fax:	+49 5251 16198-0
E-mail:	info@dspace.de
Web:	http://www.dspace.com

How to Contact dSPACE Support

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: http://www.dspace.com/go/locations
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany.
  Tel.: +49 5251 1638-941 or e-mail: support@dspace.de

You can also use the support request form: http://www.dspace.com/go/supportrequest. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit http://www.dspace.com/go/patches for software updates and patches.

Important Notice

Contents

About This Reference

Content

This RTLib Reference (Real-Time Library) gives detailed descriptions of the C functions needed to program a DS1007 PPC Processor Board. The C functions can be used to program RTI-specific Simulink S-functions, or to implement your control models manually using C programs.

Demo files

There are examples for some features included in this documentation. You will find the relevant files after the installation of your dSPACE software in <RCP_HIL_InstallationPath>\Demos\DS1007. If there is a ZIP archive, you can open it as a project backup in ControlDesk to load and start the demo application on your real-time hardware.

Symbols

dSPACE user documentation uses the following symbols:

Symbol	Description
⚠ DANGER	Indicates a hazardous situation that, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazardous situation that, if not avoided, could result in death or serious injury.
⚠ CAUTION	Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a hazard that, if not avoided, could result in property damage.
Note	Indicates important information that you should take into account to avoid malfunctions.
Tip	Indicates tips that can make your work easier.
?	Indicates a link that refers to a definition in the glossary, which you can find at the end of the document unless stated otherwise.
📖	Precedes the document title in a link that refers to another document.

Naming conventions

dSPACE user documentation uses the following naming conventions:

%name% Names enclosed in percent signs refer to environment variables for file and path names.

< > Angle brackets contain wildcard characters or placeholders for variable file and path names, etc.

Special folders

Some software products use the following special folders:

Common Program Data folder A standard folder for application-specific configuration data that is used by all users.

```
%PROGRAMDATA%\dSPACE\<InstallationGUID>\<ProductName>
```

or

```
%PROGRAMDATA%\dSPACE\<ProductName>\<VersionNumber>
```

Documents folder A standard folder for user-specific documents.

```
%USERPROFILE%\Documents\dSPACE\<ProductName>\
<VersionNumber>
```

Local Program Data folder A standard folder for application-specific configuration data that is used by the current, non-roaming user.

```
%USERPROFILE%\AppData\Local\dSPACE\<InstallationGUID>\
<ProductName>
```

Accessing dSPACE Help and PDF Files

After you install and decrypt dSPACE software, the documentation for the installed products is available in dSPACE Help and as PDF files.

dSPACE Help (local) You can open your local installation of dSPACE Help:

- On its home page via Windows Start Menu
- On specific content using context-sensitive help via **F1**

dSPACE Help (Web) You can access the Web version of dSPACE Help at www.dspace.com.

To access the Web version, you must have a *mydSPACE* account.

PDF files You can access PDF files via the icon in dSPACE Help. The PDF opens on the first page.

Processor Core Modules

Introduction

You are informed about the elementary data types and the overall functions provided by the DS1007 PPC Processor Board.

Where to go from here

Information in this section

Data Types and Definitions

Elementary Data Types

Data types

The `dstypes.h` file defines the overall processor-independent data types as follows:

`typedef signed char`	`Int8;`
`typedef unsigned char`	`UInt8;`
`typedef signed short`	`Int16;`
`typedef unsigned short`	`UInt16;`
`typedef signed int`	`Int32;`
`typedef unsigned int`	`UInt32;`
`typedef struct {UInt32 low; Int32 high;}`	`Int64;`[1)]
`typedef struct {UInt32 low; UInt32 high;}`	`UInt64;`[1)]
`typedef long long`	`Long64;`
`typedef unsigned long long`	`ULong64;`
`typedef float`	`Float32;`
`typedef double`	`Float64;`
`typedef double`	`dsfloat;`
`typedef Int8 *`	`Int8Ptr;`
`typedef UInt8 *`	`UInt8Ptr;`
`typedef Int16 *`	`Int16Ptr;`
`typedef UInt16 *`	`UInt16Ptr;`
`typedef Int32 *`	`Int32Ptr;`
`typedef UInt32 *`	`UInt32Ptr;`
`typedef Int64 *`	`Int64Ptr;`
`typedef UInt64 *`	`UInt64Ptr;`
`typedef Long64 *`	`Long64Ptr;`
`typedef ULong64 *`	`ULong64Ptr;`
`typedef Float32 *`	`Float32Ptr;`
`typedef Float64 *`	`Float64Ptr;`

[1)] The `Int64` and `UInt64` data types are deprecated and are provided only for backward-compatibility to older applications. It is recommended to use `Long64` and `ULong64` whenever 64 bit integer data types are required.

Include file	`dstypes.h`

Initialization

init

Syntax

```
init(void)
```

Include file

`brtenv.h`

Purpose

To initialize all required hardware and software modules for the DS1007.

It is recommended to use the `RTLIB_INIT` macro. For further information, refer to **`RTLIB_INIT`** on page 209.

Note

The initialization function `init` must be executed at the beginning of each application. It can only be invoked once. Further calls to this function are ignored.
When you are using RTI, this function is called automatically in the simulation engine. Hence, you do not need to call `init` in S-functions. If you need to initialize single components that are not initialized by `init`, use the specific initialization functions that are described at the beginning of the function references.

Related topics

References

Background Service

Where to go from here

Information in this section

RTLIB_BACKGROUND_SERVICE

Syntax

```
RTLIB_BACKGROUND_SERVICE()
```

Include file

SrtkStd.h

Purpose

To call the essential functions in the model background loop.

Description

This macro executes all the required background services, for example, for the host communication. It must be continuously called in the background of your application, for example, within a `for` or a `while` construct. To constantly maintain its functionality, it must be called at least once per second.

Example

This is a code example for a background loop in an application program:

```
while(1)
{
    RTLIB_BACKGROUND_SERVICE();
}
```

rtlib_background_hook

Syntax

```
int rtlib_background_hook(rtlib_bg_fcn_t *fcnptr)
```

or

```
RTLIB_REGISTER_BACKGROUND_HANDLER(rtlib_bg_fcn_t *fcnptr)
```

Include file

`SrtkStd.h`

Purpose

To register a function to be executed in the background loop.

Description

You can register several functions by calling `rtlib_background_hook` subsequently.

Note

- The specified function must be of type `rtlib_bg_fcn_t`, which defines a function with no arguments and no return value.
- The background loop waits for the execution of the specified hook functions. Ensure that the hook functions do not completely block the background service.

Parameters

fcnptr Specifies the pointer to the background function.

Return value

This function returns the following values:

Return Value	Meaning
0	The background function has been registered successfully.
-1	An error occurred while registering the background function.

Example

This example shows how to implement a simple hook function within the background loop. The variable `bg_count` counts the number of executed background loops.

```
int bg_count=0;
void bg_fcn()
{
   bg_count++;
}
void main(void)
{
   int result;
   init();
  /* setup foreground, for e.g. a timer isr */
   …
   result = rtlib_background_hook(bg_fcn);
   …
```

```
  /* background loop */
   while(1)
   {
     /* call the background functions */
      RTLIB_BACKGROUND_SERVICE();
   }
}
```

Related topics

References

Time Interval Measurement

Introduction

Functions for measuring time intervals are used for profiling application code (execution time measurement) or for implementing time delays. The time is derived from the built-in PowerPC time base, which has a resolution of 25 MHz.

> **Tip**
>
> Here you find the descriptions of platform-specific functions and generic `RTLIB_TIC_XXX` macros. It is recommended to use the generic macros.

Where to go from here

Information in this section

Data Types for Time Measurement

Introduction

There is one specific data type used by the **`srtk_tic_count`**, **`srtk_tic_elapsed`**, **`srtk_tic_diff`** functions and their related macros.

rtlib_tic_t

This data type is used to specify the time base counter values. It is defined as ULong64 data type.

Example of Using Time Measurement Functions

Example

The following example shows the source code to measure the execution time of certain actions. Three actions are specified in the program, but only action 1 and action 3 are measured using the board-specific function names:

```
srtk_tic_start();   /* starts time measurement */
...
time = srtk_tic_read();
... action 1 ...
srtk_tic_halt(); /* start of the break */
... action 2 ...
srtk_tic_continue(); /* end of the break */
... action 3 ...
time = srtk_tic_read() - time;
/* second read and calculation of the action 1 and 3 period */
```

To measure the execution time of action 1 and action 3 using the standard macros:

```
RTLIB_TIC_START();   /* starts time measurement */
...
time = RTLIB_TIC_READ();
... action 1 ...
RTLIB_TIC_HALT();  /* start of the break */
... action 2 ...
RTLIB_TIC_CONTINUE(); /* end of the break */
... action 3 ...
time = RTLIB_TIC_READ() - time;
/* second read and calculation of the action 1 and 3 period */
```

srtk_tic_continue

Syntax

```
srtk_tic_continue()
```

Include file

`SrtkTick.h`

Purpose

To resume time measurement after it was paused by **`srtk_tic_halt`**.

Description

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value

None

Related topics

Examples

References

srtk_tic_count

Syntax	`rtlib_tic_t srtk_tic_count(void)`
Include file	`SrtkTick.h`
Purpose	To read the current counter value of the time base.
Description	Use **`srtk_tic_count`** in conjunction with **`srtk_tic_elapsed`** or **`srtk_tic_diff`** to perform execution time measurement in recursive functions.
Parameters	None
Return value	This function returns the current counter value of the time base as **`rtlib_tic_t`** data type.

Example

The following example shows how to calculate the time difference between two time base counter values.

```
void main(void)
{
   rtlib_tic_t timer_count1 = 0,
   rtlib_tic_t timer_count2 = 0;
   dsfloat exec_time = 0;

   init();

   timer_count1 = srtk_tic_count();
   …
   timer_count2 = srtk_tic_count();
   exec_time = srtk_tic_diff(timer_count1, timer_count2);
   …
}
```

Related topics

References

srtk_tic_delay

Syntax	`srtk_tic_delay(Float64 duration)`
Include file	`SrtkTick.h`
Purpose	To perform the specified time delay.
Parameters	**duration** Specifies the time delay in seconds. If you specify a duration that exceeds the maximum range of the timer, the function never stops.
Return value	None
Related topics	References

srtk_tic_diff

Syntax

```
dsfloat srtk_tic_diff(
      rtlib_tic_t tmr_cnt1,
      rtlib_tic_t tmr_cnt2)
```

Include file `SrtkTick.h`

Purpose To calculate the difference between two time base counter values.

Description Use **`srtk_tic_diff`** in conjunction with **`srtk_tic_count`** or **`srtk_tic_elapsed`** to perform execution time measurement in recursive functions.

Parameters

tmr_cnt1 Specifies the first time base counter value.

tmr_cnt2 Specifies the second time base counter value.

Return value

This function returns the time difference in seconds.

Example

The following example shows how to calculate the time difference between two time base counter values.

```
void main(void)
{
   rtlib_tic_t timer_count1 = 0, timer_count2 = 0;
   dsfloat exec_time = 0;

   init();

   timer_count1 = srtk_tic_count();
   …
   timer_count2 = srtk_tic_count();
   exec_time = srtk_tic_diff(timer_count1, timer_count2);
   …
}
```

Related topics

References

RTLIB_TIC_DIFF 38
srtk_tic_count 27
srtk_tic_elapsed 29

srtk_tic_elapsed

Syntax

```
dsfloat srtk_tic_elapsed(rtlib_tic_t tmr_cnt)
```

Include file

`SrtkTick.h`

Purpose

To calculate the difference between a previous time base counter value specified by `tmr_cnt` and the current time base value in seconds.

Description

Use **`srtk_tic_elapsed`** in conjunction with **`srtk_tic_count`** or **`srtk_tic_diff`** to perform execution time measurement in recursive functions.

Parameters	**tmr_cnt**	Specifies the previous counter value of the time base.

Return value

This function returns the elapsed time in seconds.

Example

The following example shows how to calculate the time difference between a previous time base counter value and the current time base value.

```
void main(void)
{
   rtlib_tic_t timer_count;
   dsfloat exec_time = 0;

   init();

   timer_count = srtk_tic_count();
   …
   exec_time = srtk_tic_elapsed(timer_count);
   …
}
```

Related topics

References

RTLIB_TIC_ELAPSED........39
srtk_tic_count........27
srtk_tic_diff........28

srtk_tic_halt

Syntax

```
srtk_tic_halt()
```

Include file

`SrtkTick.h`

Purpose

To pause time measurement.

Description

The break lasts until measurement is resumed by `srtk_tic_continue`.

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value None

Related topics

Examples

References

srtk_tic_read

Syntax

```
Float64 srtk_tic_read()
```

Include file `SrtkTick.h`

Purpose To read the time period since time measurement was started by **`srtk_tic_start`**, minus the breaks made from **`srtk_tic_halt`** to **`srtk_tic_continue`**.

Description Use **`srtk_tic_total_read`** to read the complete time period including the breaks.

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value This function returns the time duration in seconds.

Related topics

Examples

References

srtk_tic_start

Syntax

```
srtk_tic_start()
```

Include file

`SrtkTick.h`

Purpose

To start a time measurement.

Description

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value

None

Related topics

Examples

References

srtk_tic_total_read

Syntax

```
Float64 srtk_tic_total_read()
```

Include file

`SrtkTick.h`

Purpose

To read the complete time period since the time measurement was started by **`srtk_tic_start`**, including all breaks made from **`srtk_tic_halt`** to **`srtk_tic_continue`**.

Description

Use **`srtk_tic_read`** to read the time period minus the breaks made.

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value

This function returns the time duration in seconds.

Related topics

References

RTLIB_TIC_READ_TOTAL.......... 42
srtk_tic_continue.......... 26
srtk_tic_halt.......... 30
srtk_tic_read.......... 31

srtk_timebase_fltread

Syntax

```
Float64 srtk_timebase_fltread(void)
```

Include file

`SrtkTmr.h`

Purpose

To read the 64-bit value of the time base register and convert the result to a 64-bit float value (seconds).

Return value

This function returns the current value of the time base register in seconds.

Related topics

References

srtk_timebase_low_read

Syntax

```
UInt32 srtk_timebase_low_read(void)
```

Include file

`SrtkTmr.h`

Purpose

To read the lower 32 bits of the time base register.

Description

Use `srtk_timebase_read` to read the complete time base register.

Note

This function is provided for downward compatibility and should not be used on DS1007

The time base of the DS1007 has a resolution of 25 MHz. The lower time base register (TBRL) will wrap to zero after nearly three minutes. This causes problems if the TBRL is used for interval measurements or delays greater than three minutes.

Return value

This function returns the lower 32 bits of the current time base register.

Related topics

References

srtk_timebase_read

Syntax	`ULong64 srtk_timebase_read(void)`
Include file	`SrtkTmr.h`
Purpose	To read the 64 bits of the time base register.
Return value	This function returns the current value of the time base register.
Related topics	References

RTLIB_TIC_CONTINUE

Syntax	`RTLIB_TIC_CONTINUE()`
Include file	`SrtkStd.h`
Purpose	To resume time measurement after it was paused by **`RTLIB_TIC_HALT`**. This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.
Return value	None

Related topics	Examples	

Examples

Example of Using Time Measurement Functions.. 25

References

RTLIB_TIC_HALT.. 40
srtk_tic_continue.. 26

RTLIB_TIC_COUNT

Syntax

```
rtlib_tic_t RTLIB_TIC_COUNT(void)
```

Include file

`SrtkStd.h`

Purpose

To read the current counter value of the time base.

Description

Use `RTLIB_TIC_COUNT()` in conjunction with **`RTLIB_TIC_ELAPSED`** or **`RTLIB_TIC_DIFF`** to perform execution time measurement in recursive functions.

Parameters

None

Return value

This function returns the current counter value of the time base as **`rtlib_tic_t`** data type.

Example

The following example shows how to calculate the time difference between two time base counter values.

```
void main(void)
{
   rtlib_tic_t timer_count1 = 0,
   rtlib_tic_t timer_count2 = 0;
   dsfloat exec_time = 0;

   init();

   timer_count1 = RTLIB_TIC_COUNT();
   …
   timer_count2 = RTLIB_TIC_COUNT();
   exec_time = RTLIB_TIC_DIFF(timer_count1, timer_count2);
}
```

Related topics

References

RTLIB_TIC_DIFF 38
RTLIB_TIC_ELAPSED 39
srtk_tic_count 27

RTLIB_TIC_DELAY

Syntax

```
RTLIB_TIC_DELAY(Float64 duration)
```

Include file

`SrtkStd.h`

Purpose

To perform the specified time delay.

Parameters

duration Specifies the time delay in seconds. If you specify a duration that exceeds the maximum range of the timer, the function never stops.

Return value

None

Related topics

References

RTLIB_TIC_DIFF

Syntax

```
dsfloat RTLIB_TIC_DIFF(
      rtlib_tic_t tmr_cnt1,
      rtlib_tic_t tmr_cnt2)
```

Include file

`SrtkStd.h`

Purpose

To calculate the difference between two time base counter values.

Description

Use **`RTLIB_TIC_DIFF`** in conjunction with **`RTLIB_TIC_COUNT`** or **`RTLIB_TIC_ELAPSED`** to perform execution time measurement in recursive functions.

Parameters

tmr_cnt1	Specifies the first time base counter value.
tmr_cnt2	Specifies the second time base counter value.

Return value

This function returns the time difference in seconds.

Example

The following example shows how to calculate the time difference between two time base counter values.

```
void main(void)
{
   rtlib_tic_t timer_count1 = 0, timer_count2 = 0;
   dsfloat exec_time = 0;

   init();

   timer_count1 = RTLIB_TIC_COUNT();
   …
   timer_count2 = RTLIB_TIC_COUNT();
   exec_time = RTLIB_TIC_DIFF(timer_count1, timer_count2);
   …
}
```

Related topics

References

RTLIB_TIC_COUNT.. 36
RTLIB_TIC_ELAPSED.. 39
srtk_tic_diff.. 28

RTLIB_TIC_ELAPSED

Syntax

```
dsfloat RTLIB_TIC_ELAPSED(rtlib_tic_t tmr_cnt)
```

Include file

`SrtkStd.h`

Purpose

To calculate the difference between a previous time base counter value specified by `tmr_cnt` and the current time base value in seconds using a generic macro.

Description

Use **`RTLIB_TIC_ELAPSED`** in conjunction with **`RTLIB_TIC_COUNT`** or **`RTLIB_TIC_DIFF`** to perform execution time measurement in recursive functions.

Parameters

tmr_cnt	Specifies the previous counter value of the time base.

Return value

This function returns the elapsed time in seconds.

Example

The following example shows how to calculate the time difference between a previous time base counter value and the current time base value.

```
void main(void)
{
   rtlib_tic_t timer_count;
   dsfloat exec_time = 0;

   init();

   timer_count = RTLIB_TIC_COUNT();
   …
   exec_time = RTLIB_TIC_ELAPSED(timer_count);
   …
}
```

Related topics

References

RTLIB_TIC_HALT

Syntax

```
RTLIB_TIC_HALT()
```

Include file

`SrtkStd.h`

Purpose

To pause time measurement.

Description

The break lasts until measurement is resumed by **`RTLIB_TIC_CONTINUE`**.

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value

None

Related topics

Examples

References

RTLIB_TIC_READ

Syntax

```
RTLIB_TIC_READ()
```

Include file

`SrtkStd.h`

Purpose

To read the time period since time measurement was started by **`RTLIB_TIC_START`**, minus the breaks made from **`RTLIB_TIC_HALT`** to **`RTLIB_TIC_CONTINUE`**.

Description

Use **`RTLIB_TIC_READ_TOTAL`** to read the complete time period including the breaks made.

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value

This function returns the time duration in seconds.

Related topics

Examples

References

RTLIB_TIC_READ_TOTAL

Syntax

```
RTLIB_TIC_READ_TOTAL()
```

Include file

`SrtkStd.h`

Purpose

To read the complete time period since the time measurement was started by **`RTLIB_TIC_START`**, including all breaks made from **`RTLIB_TIC_HALT`** to **`RTLIB_TIC_CONTINUE`**.

Description

Use **`RTLIB_TIC_READ`** to read the time period minus the breaks made.

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value

This function returns the time duration in seconds.

Related topics

References

RTLIB_TIC_START

Syntax

```
RTLIB_TIC_START()
```

Include file

`SrtkStd.h`

Purpose

To start a time measurement.

Description

This function is not reentrant. It is recommended to use the time-stamping functions instead, refer to Time-Stamping Functions on page 47.

Return value None

Related topics

Examples

References

Time-Stamping

Introduction

The time-stamping module is used to take absolute time stamps from a highly accurate, absolute time base.

Where to go from here

Information in this section

General Information on Time-Stamping

Introduction

Gives you information on basic principles and implementation details of the time-stamping feature.

Where to go from here

Information in this section

Basic Principles of Time-Stamping

Introduction

The Time-Stamping module is used to take absolute time stamps from a highly accurate, absolute time base. The time base fulfills the following requirements:

Time stamp accuracy The exact resolution depends on the mode of the Time-Stamping module. See Modes of the Time-Stamping module on page 45 for the exact resolution.

Time stamp range The time base has a range of 64 bit. Combined with a resolution down to 40 ns, this is enough to measure highly accurate absolute times up to several years.

Principles of an Absolute Time in Single-Processor and Multiprocessor Systems

Introduction

The Time-Stamping module is the fundamental time base for real-time simulations. It provides sufficiently accurate samples of the independent variable time. Therefore, if data and events have been recorded together with the associated time stamps, it is possible to reconstruct their temporal order.

> **Note**
>
> The same information applies to multiprocessor applications running on a multicore system.

Synchronization of local clocks

Each processor has its own local time base (local clock). Due to manufacturing tolerances, which lead to clock drifts, the local clocks in a multiprocessor system have to be synchronized periodically. To keep the communication effort low, synchronization does not take place at every tick of the local clocks (microtick), but at a selected tick of a timing master. This selected tick is called macrotick.

In single-processor systems, no synchronization is required. In a single-processor system, one macrotick equals 2^{32} microticks with the microtick running at the speed of the CPU's time base. This means that macrotick and microtick are actually stored in a 64-bit value. The data type of the time stamp structure contains one counter for the microtick and one for the macrotick. Because of this, the time stamp structure meets the requirements of both single-processor and multiprocessor systems.

When a macrotick occurs, the number of microticks is set to zero and the number of macroticks is increased by 1 at each processor. Starting from this point in time, the absolute time t_{abs} is calculated as follows:

$t_{abs} = MAT \cdot P_{MAT} + MIT \cdot P_{MIT}$

In this equation, “MAT” denotes the number of macroticks, which is incremented in the entire system, whereas “MIT” is the number of microticks. “P_{MAT}” is the macrotick period, which is a system-wide constant. “P_{MIT}” denotes the microtick period that can differ from clock to clock.

Synchronization by interrupts

The macrotick is dispatched from the timing master to all other processors in the system. This is done by an interrupt line of the DS1007 Gigalinks. The dispatching mechanism and the macrotick event mechanism are implemented in the hardware and are therefore fully transparent for the applications.

Modes of the Time-Stamping module

The Time-Stamping module can operate in three different modes:

single mode This is the mode for single-processor systems (single-core applications).

The microtick (the tick of the local clock) is directly taken from the CPU-internal time-base register. It has a resolution of 40 ns (25 MHz) on the DS1007.

multi-master mode This is the mode of the timing master in a multiprocessor system (multicore application).

The microtick is generated by the synchronous time base unit (STBU), and driven by the bus clock of the DS1007, scaled by 2. For example, at a DS1007 with 100 MHz bus clock, the resolution of the Microtick Counter is 20 ns.

When the Microtick Counter reaches the macrotick period, a system-wide macrotick is generated.

multi-slave mode This is the mode of all other processors in a multiprocessor system (multicore application).

As on the master processor, the microtick is generated by the STBU. Processors in *Slave mode* receive their macrotick from the timing master.

Data Types and Global Variables for Time-Stamping

Introduction

Gives you basic information on data types and global variables used for time-stamping.

Data Types Used for Time-Stamping

Data types

The following data types are defined for time-stamping:

<table>
<tr><th>Data Type</th><th>Syntax</th></tr>
<tr><td>ts_timestamp_type</td><td><pre>typedef struct
{
     UInt32 mat;   /* 32 bit macrotick counter value */
     UInt32 mit;   /* 32 bit microtick counter value */
}ts_timestamp_type;</pre></td></tr>
<tr><td>ts_timestamp_ptr_type</td><td><code>typedef ts_timestamp_type *   ts_timestamp_ptr_type</code></td></tr>
</table>

Time-Stamping Functions

Introduction

Gives you information on the C functions available for the time-stamping feature.

Where to go from here

Information in this section

ts_init

Syntax

```
int ts_init(
      int mode,
      float mat_period)
```

Include file

dsts.h

Purpose

To initialize the Time-Stamping module and the hardware, and to reset the Microtick and the Macrotick Counter.

Description

- The function `ts_init` is called automatically during board initialization. The Time-Stamping module is set to the `TS_MODE_SINGLE` mode.
- The function `ts_init` is also called automatically by the multiprocessor initialization, which sets the Time-Stamping module to the `TS_MODE_MULTI_MASTER` mode at the processor core with ID 0 and to the `TS_MODE_MULTI_SLAVE` mode at the other core.
- When the Time-Stamping module is initialized with `TS_MODE_MULTI_MASTER` or `TS_MODE_MULTI_SLAVE`, the Synchronous Time Base Unit (STBU) is stopped. It can be started explicitly by calling `ts_reset`.

Parameters

mode Specifies the mode of the Time-Stamping module; the following symbols are predefined:

Predefined Symbol	Meaning
`TS_MODE_SINGLE`	single mode
`TS_MODE_MULTI_MASTER`	multi-master mode
`TS_MODE_MULTI_SLAVE`	multi-slave mode

mat_period Specifies the time in seconds of one macrotick period. In single-processor systems, this argument is ignored (can be 0.0).

Return value

This function returns the error code; the following symbols are predefined:

Predefined Symbol	Meaning
`TS_INIT_DONE`	Module initialization successful
`TS_INIT_FAILED`	Module initialization failed

Related topics

Basics

Basic Principles of Time-Stamping.. 44

References

ts_reset.. 50

ts_mat_period_get

Syntax

```
dsfloat ts_mat_period_get()
```

Include file

`dsts.h`

Purpose

To get the time for one macrotick period.

Return value

Returns the macrotick period in seconds.

Related topics

Basics

Basic Principles of Time-Stamping.. 44

References

ts_init.. 48
ts_mit_period_get.. 49

ts_mit_period_get

Syntax

```
dsfloat ts_mit_period_get()
```

Include file

`dsts.h`

Purpose	To get the time for one microtick period.
Description	The microtick depends on the frequency of the Time Base Counter.
Return value	Returns the microtick period in seconds.
Related topics	Basics

References

ts_reset

Syntax

```
void ts_reset()
```

Include file

`dsts.h`

Purpose

To reset the Time-Stamping module to the absolute time 0.

> **Note**
>
> The information, that the Time-stamping module performs a reset, is not transferred between processor boards/cores. Hence, if one processor/core performs a reset, all others must perform it too. To synchronize all processors/cores in a multiprocessor system at a specific location within the code the function **`dsgl_mp_synchronize`** can be used before calling `ts_reset` (see **`dsgl_mp_synchronize`** on page 261).

Return value

None

Related topics

Basics

Basic Principles of Time-Stamping........ 44

References

ts_init........ 48

ts_time_read

Syntax

```
double ts_time_read()
```

Include file

`dsts.h`

Purpose

To read the absolute time in seconds.

Return value

This function returns the absolute time in seconds since the initialization `ts_init` or the last reset `ts_reset`.

Related topics

Basics

Basic Principles of Time-Stamping........ 44

References

ts_timestamp_read........ 51

ts_timestamp_read

Syntax

```
void ts_timestamp_read(ts_timestamp_ptr_type ts)
```

Include file

`dsts.h`

Purpose	To read the absolute time and return it as time stamp structure.
Result	The absolute time is read and is written to the time stamp structure `ts` points to.
Parameters	**ts** Specifies the pointer to a time stamp structure for the read value.
Return value	None

Related topics

Basics

Basic Principles of Time-Stamping..44

References

ts_time_read..51

ts_timestamp_compare

Syntax

```
int ts_timestamp_compare(
      ts_timestamp_ptr_type ts1,
      ts_timestamp_ptr_type ts2,
      int operation)
```

Include file

`dsts.h`

Purpose

To compare two time stamps.

Parameters

ts1 Specifies the pointer to the first time stamp structure.

ts2 Specifies the pointer to the second time stamp structure.

operation Specifies the kind of operation; the following symbols are predefined:

Predefined Symbol	Meaning
`TS_COMPARE_LT`	less than
`TS_COMPARE_LE`	less than or equal to

Predefined Symbol	Meaning
TS_COMPARE_EQ	equal
TS_COMPARE_GE	greater than or equal to
TS_COMPARE_GT	greater than

Return value

This function returns the operation result; the following symbols are predefined:

Value	Meaning
= 0	Result is false
!= 0	Result is true

Related topics

Basics

Basic Principles of Time-Stamping.. 44

References

ts_timestamp_interval.. 53

ts_timestamp_interval

Syntax

```
double ts_timestamp_interval(
      ts_timestamp_ptr_type ts1,
      ts_timestamp_ptr_type ts2)
```

Include file

`dsts.h`

Purpose

To calculate the interval in seconds between time stamps 1 and 2.

Parameters

ts1 Specifies the pointer to the first time stamp structure.

ts2 Specifies the pointer to the second time stamp structure.

Return value

This function returns the interval between time stamps 1 and 2 in seconds.

Related topics

Basics

References

ts_time_offset

Syntax

```
void ts_time_offset(
      double reference_time,
      ts_timestamp_ptr_type ts1,
      ts_timestamp_ptr_type ts2,
      ts_timestamp_ptr_type ts_ta)
```

Include file

`dsts.h`

Purpose

To calculate the time offset.

Result

The interval between time stamps 1 and 2 is calculated and the difference between the time stamps is added to the reference time. The absolute time is returned as a time stamp.

Parameters

reference_time Specifies the reference time in seconds.

ts1 Specifies the pointer to the first time stamp structure.

ts2 Specifies the pointer to the second time stamp structure.

ts_ta Specifies the pointer to the time stamp structure for the calculated value.

Return value

None

Related topics

Basics

References

ts_timestamp_offset

Syntax

```
void ts_timestamp_offset(
      ts_timestamp_ptr_type ts_reference,
      ts_timestamp_ptr_type ts1,
      ts_timestamp_ptr_type ts2,
      ts_timestamp_ptr_type ts_ta)
```

Include file

`dsts.h`

Purpose

To calculate the time offset.

Result

The interval between time stamps 1 and 2 is calculated and the difference between the time stamps is added to the reference time stamp. The absolute time is returned as a time stamp.

Parameters

ts_reference Specifies the pointer to the time stamp structure holding the reference time.

ts1 Specifies the pointer to the first time stamp structure.

ts2 Specifies the pointer to the second time stamp structure.

ts_ta Specifies the pointer to the time stamp structure holding the absolute time in seconds.

Return value

None

Related topics

Basics

References

ts_time_calculate

Syntax

```
double ts_time_calculate(ts_timestamp_ptr_type ts)
```

Include file

`dsts.h`

Purpose

To convert a time stamp structure to a time value in seconds.

Parameters

ts	Specifies the pointer to a time stamp structure.

Return value

This function returns the time corresponding to the time stamp.

Related topics

Basics

References

ts_timestamp_calculate

Syntax

```
void ts_time_calculate(
      double time,
      ts_timestamp_ptr_type ts)
```

Include file	`dsts.h`
Purpose	To convert a time value in seconds to a time stamp structure.
Parameters	**time** Specifies the time in seconds. **ts** Specifies the pointer to a time stamp structure for the calculated value.
Return value	None
Related topics	Basics

References

Timer A

Introduction

Timer A is a down counter generating an interrupt whenever it reaches zero. The period value is then reloaded automatically. Timer A is also used by the `RTLIB_SRT_PERIOD` standard macro as the default sampling rate timer.

For further information on Timer A, refer to Timer A and Timer D (DS1007 Features 🕮).

Where to go from here

Information in this section

Information in other sections

Example of Using Timer A Functions

Example

The following example demonstrates how to use Timer A functions.

```
#include <Brtenv.h>
#define DT 1.0e-4          /* 100 µs simulation step size */
```

```
/* ++ variables for host PC +++++++++++++++++++++++++++ */
Float64 exec_time, timeA;     /* execution time */
void ad_routine(void)
{
   ts_timestamp_type ts;
   Float64 old_timeA;
   RTLIB_SRT_ISR_BEGIN();           /* overrun check TimerA */
   srtk_timerA_read(&old_timeA);

   ts_timestamp_read(&ts);
   DsDaq_Service(0, 0, 1,
      (DsDaqSTimestampStruct *)&ts);   /* data acquisition service */
   /* +++ do something +++ */
   srtk_timerA_read(&timeA);
   exec_time = old_timeA - timeA; /* exec time with Timer A */
   RTLIB_SRT_ISR_END();           /* overrun check TimerA */
}
void main(void)
{
   /* init processor board */
   init();
   /* set period of timerA */
   timeA = DT;
   srtk_timerA_period_set(timeA);
   /* periodic event in ISR */
   RTLIB_SRT_START(timeA, ad_routine);
   /* Background tasks */
   while(1)
   {
      RTLIB_BACKGROUND_SERVICE();    /* host PC service */
   }
}
```

RTLIB_SRT_PERIOD

Syntax

```
RTLIB_SRT_PERIOD(Float64 time)
```

Include file

`SrtkStd.h`

Purpose

To set a new period of Timer A and restart it immediately.

Description

The new value is loaded immediately: Timer A is stopped, the new value is set, and Timer A is started again.

Parameters

time Specifies the period in seconds.

Return value	None
Related topics	References

srtk_timerA_period_set

Syntax	`void srtk_timerA_period_set(Float64 time)`
Include file	`SrtkTmr.h`
Purpose	To set the period of Timer A.
Description	If the timer is not running, the new value is loaded immediately. If the timer is running, the new value is loaded the next time the timer reaches zero.
Parameters	**time** Specifies the period in seconds.
Return value	None
Related topics	Examples References

srtk_timerA_period_reload

Syntax	`void srtk_timerA_period_reload(Float64 time)`

Include file	`SrtkTmr.h`
Purpose	To set a new period of Timer A and restart it immediately.
Description	The new value is loaded immediately: Timer A is stopped, the new value is set, and Timer A is started again.
Parameters	**time** Specifies the period in seconds.
Return value	None
Related topics	References

srtk_timerA_read

Syntax	`void srtk_timerA_read(Float64 *time)`
Include file	`SrtkTmr.h`
Purpose	To read the current value of Timer A.
Parameters	**time** Specifies the pointer to the current value of Timer A. The value is stated in seconds.
Return value	None

srtk_timerA_start

Syntax	`void srtk_timerA_start(void)`
Include file	`SrtkTmr.h`
Purpose	To start Timer A.
Description	If no period is set, the counter starts with the highest counter value (0xFFFF FFFF). **Tip** Use **`srtk_timerA_period_set`** to set the period.
Return value	None
Related topics	References srtk_timerA_period_set 60

srtk_timerA_stop

Syntax	`void srtk_timerA_stop(void)`
Include file	`SrtkTmr.h`
Purpose	To stop Timer A. **Tip** Use **`srtk_timerA_start`** to resume from the current value.

Return value

None

Related topics

References

Timer B

Introduction

Timer B is a counter generating an interrupt when it reaches its compare value and continues counting. Thus, Timer B is designed only for single timer events. If your model requires periodic timer events, use Timer A (refer to Timer A on page 58). If Timer A is already used, use Timer B and set its compare value periodically (function srtk_timerB_compare_set_periodically).

For further information on Timer B, refer to Timer B (DS1007 Features 📖).

Where to go from here

Information in this section

Information in other sections

Example of Using Timer B Functions

Example

The following example demonstrates how to use Timer B functions.

```
#include <Brtenv.h>
#define DT 1e-4              /* 100 µs simulation step size */
```

```
/* ++ variables for execution time profiling +++++++++++++++++ */
Float64 exec_time;                        /* execution time */
Float64 timerB;                        /* timerB read value */
/* ++ adjust values for timerB ++++++++++++++++++++++++++++ */
Float64 upc_period = .001;       /* upcounter period in sec */
UInt16 scale_value = 2;        /* set the scaling of timerB */
/* ++ counter for Interrupt service functions +++++++++++++ */
Int32 timerB_counter = 0;
void isr_timerB(void)
{

   ts_timestamp_type ts;
   srtk_begin_isr_timerB();              /* overrun check */
   RTLIB_TIC_START();
   timerB_counter++;       /* counter for timerB interrupts */
   srtk_timerB_read(&timerB);              /* read timerB */
   ts_timestamp_read(&ts);
   DsDaq_Service(0, 0, 1,
      (DsDaqSTimestampStruct *)&ts);   /* data acquisition service */
   exec_time = RTLIB_TIC_READ();
   srtk_end_isr_timerB();                /* overrun check */
}
void main(void)
{
   /* init processor board */
   init();
   /* periodic event with TimerB */
   srtk_start_isr_timerB(scale_value,
                         upc_period,
                         isr_timerB);
   /* Background task */
   while(1)
   {
      RTLIB_BACKGROUND_SERVICE();    /* host PC service */
   }
}
```

srtk_timerB_init

Syntax

```
void srtk_timerB_init(UInt16 scale)
```

Include file

`SrtkTmr.h`

Purpose

To initialize Timer B.

Parameters

scale Specifies a value within the range 0 … 7 that defines the prescaler setting of Timer B as a function of the I/O bus clock. The I/O bus clock runs at a speed of 100 MHz (10 ns).

Scale Value	Timer B Clock/ Bus Clock	(I/O bus clock is 100 MHz)	
		Timer B Clock	Prescaler Period
0	1/4	25.0 MHz	40 ns
1	1/8	12.5 MHz	80 ns
2	1/16	6.25 MHz	160 ns
3	1/32	3.125 MHz	320 ns
4	1/64	1.5625 MHz	640 ns
5	1/128	0.78125 MHz	1280 ns
6	1/256	0.390625 MHz	2560 ns
7	1/512	0.1953125 MHz	5120 ns

Return value

None

Example

The DS1007 I/O bus clock has a resolution of 10 ns. To achieve a timer period of 160 ns, the prescaler must be set to 1/16:

```
srtk_timerB_init(SRTK_TIMERB_1_16_BCLK);
```

Related topics

References

srtk_timerB_compare_set 66

srtk_timerB_compare_set

Syntax

```
void srtk_timerB_compare_set(Float64 delta_time)
```

Include file

`SrtkTmr.h`

Purpose

To set the new compare value.

Description

The compare value to be written to the Timer B compare register is calculated by adding the `delta_time` to the current timer value. When the counter value matches the value of the compare register, Timer B generates an interrupt.

To make the Timer B interrupt available, refer to Timer Interrupt Control on page 75 and Interrupt Handling on page 87.

If you want to generate a Timer B interrupt periodically, use the function **`srtk_timerB_compare_set_periodically`**.

Parameters

delta_time Specifies the period in seconds.

Return value

None

Related topics

References

srtk_timerB_compare_set_periodically

Syntax

```
void srtk_timerB_compare_set_periodically(
     Float64 delta_time)
```

Include file

`SrtkTmr.h`

Purpose

To periodically set a new compare value.

Description

This function is used in the Timer B interrupt service routine to make Timer B a periodic timer. The new compare value to be written to the Timer B compare register is calculated by adding the **`delta_time`** to the old compare value.

When the counter value matches the value of the compare register, Timer B generates an interrupt.

This function is automatically called in your interrupt service routine when using **`srtk_begin_isr_timerB`**.

To make the Timer B interrupt available, refer to Timer Interrupt Control on page 75 and Interrupt Handling on page 87.

Parameters

delta_time Specifies the period in seconds.

Return value

None

Related topics

References

srtk_begin_isr_timerA 77
srtk_timerB_compare_set 66

srtk_timerB_read

Syntax

```
void srtk_timerB_read(Float64 *time)
```

Include file

`SrtkTmr.h`

Purpose

To read the current value of Timer B.

Parameters

time Specifies the pointer to the current value of Timer B. The value is given in seconds.

Return value

None

Related topics

Examples

Example of Using Timer B Functions 64

srtk_timerB_start

Syntax

```
void srtk_timerB_start(void)
```

Include file

`SrtkTmr.h`

Purpose

To start Timer B.

> **Tip**
>
> Use `srtk_timerB_compare_set` to set the compare value.

Return value

None

Related topics

References

srtk_timerB_stop

Syntax

```
void srtk_timerB_stop(void)
```

Include file

`SrtkTmr.h`

Purpose

To stop Timer B.

> **Tip**
>
> Use `srtk_timerB_start` to continue.

Return value

None

Related topics

References

Timer D

Introduction

Timer D is functionally identical to Timer A. Timer D is a down counter generating an interrupt whenever it reaches zero. The period value is then reloaded automatically.

For further information on Timer D, refer to Timer A and Timer D (DS1007 Features 📖).

Where to go from here

Information in this section

Information in other sections

For information on handling Timer D interrupts.

Example of Using Timer D functions

Example

The following example demonstrates how to use Timer D functions.

```
#include <Brtenv.h>
#define DT 1.0e-4            /* 100 µs simulation step size */
/*-- variables for host PC ---------------------------*/
Float64 exec_time, timeD;      /* execution time */
```

```
void ad_routine(void)
{
   ts_timestamp_type ts;
   Float64 old_timeD;
   srtk_begin_isr_timerD();    /* overrun check TimerD */
   srtk_timerD_read(&old_timeD);
   ts_timestamp_read(&ts);
   DsDaq_Service(0, 0, 1,
      (DsDaqSTimestampStruct *)&ts);   /* data acquisition service */
   /*--- do something ---*/
   srtk_timerD_read(&timeD);
   exec_time = old_timeD - timeD; /* exec time with Timer D */
   srtk_end_isr_timerD();         /* overrun check TimerD */
}
void main(void)
{
   /* init processor board */
   init();
   /* set period of timerD */
   timeD = DT;
   srtk_timerD_period_set(timeD);
   /* periodic event in ISR */
   srtk_start_isr_timerD(timeD, ad_routine);
   /* Background tasks */
   while(1)
   {
      RTLIB_BACKGROUND_SERVICE();    /* host PC service */
   }
}
```

srtk_timerD_period_set

Syntax	`void srtk_timerD_period_set(Float64 time)`
Include file	`SrtkTmr.h`
Purpose	To define the period of Timer D.
Description	If the timer is not running, the new value is loaded immediately. If the timer is running, the new value is loaded the next time the timer reaches zero.
Parameters	**time** Specifies the period in seconds.
Return value	None

Related topics

Examples

References

srtk_timerD_period_reload

Syntax

```
void srtk_timerD_period_reload(Float64 time)
```

Include file

`SrtkTmr.h`

Purpose

To set a new period of Timer D and restart it immediately.

Description

The new value is loaded immediately. Timer D is stopped, the new value is set, and Timer D is started again.

Parameters

time	Specifies the period in seconds.

Return value

None

Related topics

References

srtk_timerD_read

Syntax

```
void srtk_timerD_read(Float64 *time)
```

Include file

`SrtkTmr.h`

Purpose

To read the current value of Timer D.

Parameters

time Specifies the pointer to the current value of Timer D. The value is stated in seconds.

Return value

None

Related topics

Examples

Example of Using Timer D functions.. 70

srtk_timerD_start

Syntax

```
void srtk_timerD_start(void)
```

Include file

`SrtkTmr.h`

Purpose

To start Timer D.

Description

If no period is set, the counter starts with the highest counter value (0xFFFF FFFF).

> **Tip**
>
> Use **`srtk_timerD_period_set`** to set the period.

Return value

None

Related topics

References

srtk_timerD_period_set.. 71

srtk_timerD_stop

Syntax	`void srtk_timerD_stop(void)`
Include file	`SrtkTmr.h`
Purpose	To stop Timer D. **Tip** Use **`srtk_timerD_start`** to resume from the current value.
Return value	None
Related topics	References

Timer Interrupt Control

Introduction

These functions are used to install interrupt service routines for the available timers and to perform overrun checks for the defined interrupt service routines.

> **Tip**
>
> Here you find the descriptions of platform-specific functions and generic `RTLIB_SRT_XXX` macros. It is recommended to use the generic macros if available.

Where to go from here

Information in this section

Information in other sections

srtk_begin_isr_timerA

Syntax

```
srtk_begin_isr_timerA()
```

Include file

`SrtkTmrInt.h`

Purpose

To check for an overrun in the interrupt service routine assigned by **`srtk_start_isr_timerA`**.

Description

When the execution time of the interrupt service routine exceeds the interrupt period (overrun), the interrupt is stopped, and an appropriate error message is generated.

Return value

None

Example

This example shows an interrupt service routine with overrun check:

```
void timerA_interrupt(void)
{
   srtk_begin_isr_timerA();
   /* interrupt service routine */
   srtk_end_isr_timerA();
}
```

Related topics

References

RTLIB_SRT_ISR_BEGIN 84
srtk_end_isr_timerA 79

srtk_begin_isr_timerB

Syntax

```
srtk_begin_isr_timerB()
```

Include file

`SrtkTmrInt.h`

Purpose

To check for an overrun in the interrupt service routine assigned by **`srtk_start_isr_timerB`** and to reload the compare value.

Description

When the execution time of the interrupt service routine exceeds the interrupt period (overrun), the interrupt is stopped, and an appropriate error message is generated.

Return value

None

Example

This example shows an interrupt service routine with overrun check:

```
void timerB_interrupt(void)
{
   srtk_begin_isr_timerB();
   /* interrupt service routine */
   srtk_end_isr_timerB();
}
```

Related topics

References

srtk_begin_isr_timerD

Syntax

```
srtk_begin_isr_timerD()
```

Include file

`SrtkTmrInt.h`

Purpose

To check for an overrun in the interrupt service routine assigned by **`srtk_start_isr_timerD`**.

Description

When the execution time of the interrupt service routine exceeds the interrupt period (overrun), the interrupt is stopped, and an appropriate error message is generated.

Return value

None

Example

This example shows an interrupt service routine with overrun check:

```
void timerD_interrupt(void)
{
   srtk_begin_isr_timerD();
   /* interrupt service routine */
   srtk_end_isr_timerD();
}
```

Related topics

References

srtk_end_isr_timerA

Syntax

```
srtk_end_isr_timerA()
```

Include file

`SrtkTmrInt.h`

Purpose

To check for an overrun in the interrupt service routine assigned by **`srtk_start_isr_timerA`**.

Return value

None

Related topics

References

srtk_end_isr_timerB

Syntax

```
srtk_end_isr_timerB()
```

Include file	`SrtkTmrInt.h`
Purpose	To check for an overrun in the interrupt service routine assigned by **`srtk_start_isr_timerB`**.
Return value	None
Related topics	References

srtk_end_isr_timerD

Syntax	`srtk_end_isr_timerD()`
Include file	`SrtkTmrInt.h`
Purpose	To check for an overrun in the interrupt service routine assigned by **`srtk_start_isr_timerD`**.
Return value	None
Related topics	References

srtk_start_isr_timerA

Syntax

```
void srtk_start_isr_timerA(
      Float64 sampling_period,
      Srtk_Int_Handler_Type isr_function_name)
```

Include file

`SrtkTmrInt.h`

Purpose

To install `isr_function_name` as an interrupt service routine for Timer A.

Description

The function sets the period of Timer A, installs the specified routine as interrupt handler, and starts Timer A.

If the execution time of the interrupt service routine exceeds the interrupt period, an overrun occurs. Use **`srtk_begin_isr_timerA`** and **`srtk_end_isr_timerA`** in your interrupt service routine to install an overrun check.

Parameters

sampling_period Specifies the period in seconds.

isr_function_name Specifies the name of the function to be assigned to the Timer A interrupt. This function must not have an input parameter or a return value, i.e., `void isr_function_name(void)`.

Return value

None

Example

This example installs the function `timerA_interrupt`, which is called when the Timer A interrupt occurs, namely every 20 µs:

```
srtk_start_isr_timerA(20e-6, timerA_interrupt);
```

Related topics

References

srtk_start_isr_timerB

Syntax

```
void srtk_start_isr_timerB(
     UInt32 scale,
     Float64 sampling_period,
     Srtk_Int_Handler_Type isr_function_name)
```

Include file

`SrtkTmrInt.h`

Purpose

To install `isr_function_name` as an interrupt service routine for Timer B and initialize Timer B.

Description

The function sets the compare value of Timer B, installs the specified routine as interrupt handler, and starts Timer B. Because Timer B is not a periodic timer, you must use **`srtk_begin_isr_timerB`** and **`srtk_end_isr_timerB`** in your interrupt service routine to reload the compare value. In addition, you install an overrun check that prevents the execution time of the interrupt service routine from exceeding the interrupt period.

Parameters

scale Specifies a value within the range 0 … 7 that defines the prescaler setting of Timer B as a function of the I/O bus clock. The I/O bus clock runs at a speed of 100 MHz (10 ns).

Scale Value	Timer B Clock/ Bus Clock	(I/O bus clock is 100 MHz)	
		Timer B Clock	Prescaler Period
0	1/4	25.0 MHz	40 ns
1	1/8	12.5 MHz	80 ns
2	1/16	6.25 MHz	160 ns
3	1/32	3.125 MHz	320 ns
4	1/64	1.5625 MHz	640 ns
5	1/128	0.78125 MHz	1280 ns
6	1/256	0.390625 MHz	2560 ns
7	1/512	0.1953125 MHz	5120 ns

sampling_period Specifies the period in seconds.

isr_function_name Specifies the name of the function to be assigned to the Timer B interrupt. This function must not have an input parameter or a return value, i.e., `void isr_function_name(void)`.

Return value

None

Example

This example installs the function **`timerB_interrupt`**, which is called when the Timer B interrupt occurs, namely every 100 µs:

```
srtk_start_isr_timerB(0, 100e-6, timerB_interrupt)
```

Related topics

References

srtk_start_isr_timerD

Syntax

```
void srtk_start_isr_timerD(
      Float64 sampling_period,
      Srtk_Int_Handler_Type isr_function_name)
```

Include file

`SrtkTmrInt.h`

Purpose

To install **`isr_function_name`** as an interrupt service routine for Timer D.

Description

The function sets the period of Timer D, installs the specified routine as interrupt handler, and starts Timer D.

If the execution time of the interrupt service routine exceeds the interrupt period, an overrun occurs. Use **`srtk_begin_isr_timerD`** and **`srtk_end_isr_timerD`** in your interrupt service routine to install an overrun check.

Parameters

sampling_period Specifies the period in seconds.

isr_function_name Specifies the name of the function to be assigned to the Timer D interrupt. This function must not have an input parameter or a return value, i.e., **`void isr_function_name(void)`**.

Return value

None

Example

This example installs the function **`timerD_interrupt`**, which is called when the Timer D interrupt occurs, namely every 20 µs:

```
srtk_start_isr_timerD(20e-6, timerD_interrupt);
```

Related topics

References

RTLIB_SRT_ISR_BEGIN

Syntax

```
RTLIB_SRT_ISR_BEGIN()
```

Include file

`SrtkStd.h`

Purpose

To check for an overrun in the interrupt service routine assigned by **`RTLIB_SRT_START`**.

Description

When the execution time of the interrupt service routine exceeds the interrupt period (overrun), the interrupt is stopped, and an appropriate error message is generated.

Return value

None

Example

This example shows an interrupt service routine with overrun check:

```
void timerA_interrupt(void)
{
   RTLIB_SRT_ISR_BEGIN();
   /* interrupt service routine */
   RTLIB_SRT_ISR_END();
}
```

Related topics

References

RTLIB_SRT_ISR_END

Syntax

```
RTLIB_SRT_ISR_END()
```

Include file

`SrtkStd.h`

Purpose

To check for an overrun in the interrupt service routine assigned by **`RTLIB_SRT_START`**.

Return value

None

Related topics

References

RTLIB_SRT_START

Syntax

```
RTLIB_SRT_START(
      Float64 sampling_period,
      Srtk_Int_Handler_Type isr_function_name)
```

Include file

`SrtkStd.h`

Purpose

To install `isr_function_name` as an interrupt service routine for Timer A.

Description

The function sets the period of Timer A, installs the specified routine as interrupt handler, and starts Timer A.

If the execution time of the interrupt service routine exceeds the interrupt period, an overrun occurs. Use **`RTLIB_SRT_ISR_BEGIN`** and **`RTLIB_SRT_ISR_END`** in your interrupt service routine to install an overrun check.

Parameters

sampling_period Specifies the period in seconds.

isr_function_name Specifies the name of the function to be assigned to the Timer A interrupt. This function must not have an input parameter or a return value, i.e., `void isr_function_name(void)`.

Return value

None

Example

This example installs the function `timerA_interrupt`, which is called when the Timer A interrupt occurs, namely every 20 µs:

```
RTLIB_SRT_START(20e-6, timerA_interrupt);
```

Related topics

References

srtk_start_isr_timerA.. 81

Interrupt Handling

Introduction

Use the interrupt handling functions to make interrupts available as trigger sources. If you want to use an interrupt, you have to install an appropriate handler and enable interrupt handling. The interrupt handling uses the interrupt identification (IntId) to identify the interrupt handler that has been installed for this interrupt. Whether or not an interrupt has been generated is indicated by the interrupt flag.

> **Note**
>
> For examples of the installation of interrupt service routines for the Timer A, Timer B and Timer D interrupts, refer to srtk_set_interrupt_vector on page 99 and Timer Interrupt Control on page 75.

For further information on the interrupt handling, refer to Interrupt Controller (DS1007 Features).

Interrupt service routine type

The interrupt service routine type is defined as follows:

```
typedef void (*Srtk_Int_Handler_Type)(void)
```

Where to go from here

Information in this section

srtk_disable_hardware_int

Syntax

```
void srtk_disable_hardware_int(UInt32 IntID)
```

Include file

`SrtkInt.h`

Purpose

To disable the specified hardware interrupt when the interrupts are still globally enabled (see **`RTLIB_INT_ENABLE`**).

Description

This function sets the corresponding bit of the Interrupt Mask Register (IMR).

Parameters

IntID Specifies the interrupt that is to be disabled.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_PHS_7	PHS-bus interrupt line 7
SRTK_INT_TIMER_A	Timer A interrupt
SRTK_INT_TIMER_B	Timer B interrupt
SRTK_INT_TIMER_D	Timer D interrupt
SRTK_INT_SERIAL_UART	Serial UART interrupt
SRTK_INT_MACROTICK	Macrotick interrupt
SRTK_INT_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_GL_3	Gigalink 3 interrupt
SRTK_INT_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

None

Related topics

References

srtk_disable_hardware_int_bm

Syntax

```
void srtk_disable_hardware_int_bm(UInt32 IntMask)
```

Include file

`SrtkInt.h`

Purpose

To disable several hardware interrupts when the interrupts are still globally enabled (see **`RTLIB_INT_ENABLE`**).

Description

This function sets the corresponding bit of the Interrupt Mask Register (IMR).

Parameters

IntMask Specifies the interrupts that are to be disabled. To specify more than one interrupt, you can combine the predefined symbols using the logical operator OR.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_MASK_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_MASK_PHS_7	PHS-bus interrupt line 7
SRTK_INT_MASK_TIMER_A	Timer A interrupt
SRTK_INT_MASK_TIMER_B	Timer B interrupt
SRTK_INT_MASK_TIMER_D	Timer D interrupt
SRTK_INT_MASK_SERIAL_UART	Serial UART interrupt
SRTK_INT_MASK_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_MASK_GL_3	Gigalink 3 interrupt
SRTK_INT_MASK_MACROTICK	Macrotick interrupt
SRTK_INT_MASK_FWD_TIMER_A	Forwarded Timer A interrupt

Predefined Symbol	Meaning
SRTK_INT_MASK_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_MASK_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

None

Related topics

References

RTLIB_INT_ENABLE........102
srtk_enable_hardware_int_bm........92

srtk_enable_hardware_int

Syntax

```
void srtk_enable_hardware_int(UInt32 IntID)
```

Include file

`SrtkInt.h`

Purpose

To enable the specified hardware interrupt.

Description

This function only clears the corresponding bit of the Interrupt Mask Register (IMR). However, the specified hardware interrupt is available only when the interrupts are globally enabled (see **`RTLIB_INT_ENABLE`**).

Parameters

IntID Specifies the interrupt that is to be enabled.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_PHS_7	PHS-bus interrupt line 7

Predefined Symbol	Meaning
SRTK_INT_TIMER_A	Timer A interrupt
SRTK_INT_TIMER_B	Timer B interrupt
SRTK_INT_TIMER_D	Timer D interrupt
SRTK_INT_SERIAL_UART	Serial UART interrupt
SRTK_INT_MACROTICK	Macrotick interrupt
SRTK_INT_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_GL_3	Gigalink 3 interrupt
SRTK_INT_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

None

Related topics

References

RTLIB_INT_ENABLE ... 102
srtk_disable_hardware_int ... 89
srtk_enable_hardware_int_bm ... 92

srtk_enable_hardware_int_bm

Syntax

```
void srtk_enable_hardware_int_bm(UInt32 IntMask)
```

Include file

`SrtkInt.h`

Purpose

To enable several hardware interrupts.

Description

This function clears the corresponding bits of the Interrupt Mask Register (IMR). However, the specified hardware interrupts are available only when the interrupts are globally enabled (see **RTLIB_INT_ENABLE**).

Parameters

IntMask Specifies the interrupts that are to be enabled. To specify more than one interrupt, you can combine the predefined symbols by using the logical operator OR.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_MASK_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_MASK_PHS_7	PHS-bus interrupt line 7
SRTK_INT_MASK_TIMER_A	Timer A interrupt
SRTK_INT_MASK_TIMER_B	Timer B interrupt
SRTK_INT_MASK_TIMER_D	Timer D interrupt
SRTK_INT_MASK_SERIAL_UART	Serial UART interrupt
SRTK_INT_MASK_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_MASK_GL_3	Gigalink 3 interrupt
SRTK_INT_MASK_MACROTICK	Macrotick interrupt
SRTK_INT_MASK_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_MASK_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_MASK_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

None

Related topics

References

RTLIB_INT_ENABLE..102
srtk_disable_hardware_int_bm..90

srtk_get_interrupt_flag

Syntax

```
int srtk_get_interrupt_flag(UInt32 IntID)
```

Include file

`SrtkInt.h`

Purpose

To get the interrupt flag for the specified interrupt.

Description

The interrupt flag indicates whether or not the specified interrupt has been generated.

Parameters

IntID Specifies the interrupt whose interrupt flag is to be read.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_PHS_7	PHS-bus interrupt line 7
SRTK_INT_TIMER_A	Timer A interrupt
SRTK_INT_TIMER_B	Timer B interrupt
SRTK_INT_TIMER_D	Timer D interrupt
SRTK_INT_SERIAL_UART	Serial UART interrupt
SRTK_INT_MACROTICK	Macrotick interrupt
SRTK_INT_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_GL_3	Gigalink 3 interrupt
SRTK_INT_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

This function returns the value of the interrupt flag:

Value	Meaning
0	Interrupt has not been generated
1	Interrupt has been generated

Related topics

References

srtk_get_interrupt_flag_bm

Syntax

```
int srtk_get_interrupt_flag_bm(UInt32 flag)
```

Include file

`SrtkInt.h`

Purpose

To get the interrupt flag for several interrupts.

Description

The interrupt flag indicates whether or not one of the specified interrupts has been generated.

Parameters

flag Specifies a bitmask of interrupts that are to be checked. To specify more than one interrupt, you can combine the predefined symbols using the logical operator OR.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_MASK_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_MASK_PHS_7	PHS-bus interrupt line 7
SRTK_INT_MASK_TIMER_A	Timer A interrupt
SRTK_INT_MASK_TIMER_B	Timer B interrupt
SRTK_INT_MASK_TIMER_D	Timer D interrupt
SRTK_INT_MASK_SERIAL_UART	Serial UART interrupt
SRTK_INT_MASK_GL_0	Gigalink 0 interrupt
...	...

Predefined Symbol	Meaning
SRTK_INT_MASK_GL_3	Gigalink 3 interrupt
SRTK_INT_MASK_MACROTICK	Macrotick interrupt
SRTK_INT_MASK_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_MASK_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_MASK_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

This function returns the value of the interrupt flag:

Value	Meaning
0	Interrupt has not been generated
1	At least one interrupt has been generated

Related topics

References

srtk_get_interrupt_flag.. 94

srtk_get_interrupt_vector

Syntax

```
Srtk_Int_Handler_Type srtk_get_interrupt_vector(
     UInt32 IntID)
```

Include file

`SrtkInt.h`

Purpose

To get the address of the interrupt service routine related to the given interrupt.

Description

Use this function to retrieve the interrupt service routine installed for a given interrupt source. If no user handler has been installed, the default handler will be returned.

Parameters

IntID Specifies the interrupt source for which the installed handler is to be returned.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_PHS_7	PHS-bus interrupt line 7
SRTK_INT_TIMER_A	Timer A interrupt
SRTK_INT_TIMER_B	Timer B interrupt
SRTK_INT_TIMER_D	Timer D interrupt
SRTK_INT_SERIAL_UART	Serial UART interrupt
SRTK_INT_MACROTICK	Macrotick interrupt
SRTK_INT_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_GL_3	Gigalink 3 interrupt
SRTK_INT_IO_ETH	I/O Ethernet interrupt

Return value

This function returns the address of the interrupt service routine that is installed for this interrupt.

srtk_reset_interrupt_flag

Syntax

```
void srtk_reset_interrupt_flag(UInt32 IntID)
```

Include file

`SrtkInt.h`

Purpose

To reset the interrupt flag for the specified interrupt.

Parameters

IntID Specifies the interrupt for which the interrupt flag is to be reset.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_PHS_7	PHS-bus interrupt line 7
SRTK_INT_TIMER_A	Timer A interrupt
SRTK_INT_TIMER_B	Timer B interrupt
SRTK_INT_TIMER_D	Timer D interrupt
SRTK_INT_SERIAL_UART	Serial UART interrupt
SRTK_INT_MACROTICK	Macrotick interrupt
SRTK_INT_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_GL_3	Gigalink 3 interrupt
SRTK_INT_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

None

Related topics

References

srtk_reset_interrupt_flag_bm 98

srtk_reset_interrupt_flag_bm

Syntax

```
void srtk_reset_interrupt_flag_bm(UInt32 flag)
```

Include file

`SrtkInt.h`

Purpose

To reset the interrupt flag for several interrupts.

Parameters

flag Specifies the bitmask of interrupts whose interrupt flag is to be reset. To specify more than one interrupt, you can combine the predefined symbols using the logical operator OR.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_MASK_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_MASK_PHS_7	PHS-bus interrupt line 7
SRTK_INT_MASK_TIMER_A	Timer A interrupt
SRTK_INT_MASK_TIMER_B	Timer B interrupt
SRTK_INT_MASK_TIMER_D	Timer D interrupt
SRTK_INT_MASK_SERIAL_UART	Serial UART interrupt
SRTK_INT_MASK_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_MASK_GL_3	Gigalink 3 interrupt
SRTK_INT_MASK_MACROTICK	Macrotick interrupt
SRTK_INT_MASK_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_MASK_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_MASK_IO_ETH	I/O Ethernet interrupt

Note

Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Return value

None

Related topics

References

srtk_reset_interrupt_flag.. 97

srtk_set_interrupt_vector

Syntax

```
Srtk_Int_Handler_Type srtk_set_interrupt_vector(
      UInt32 IntID,
      Srtk_Int_Handler_Type Handler)
```

Include file `SrtkInt.h`

Purpose To install an interrupt service routine for the selected interrupt.

> **Note**
>
> - When you want to migrate your code written for DS1005 or DS1006 to DS1007, you have to note that the `SaveRegs` parameter is no more available.
> - Use `RTLIB_INT_ENABLE` to enable interrupts.
> - The installation of interrupt service routines for the Timer A, Timer B, and Timer D interrupts is different from that of other interrupts. Refer to the example below and to Timer Interrupt Control on page 75.

Parameters

IntID Identifies the interrupt that the handler is to be installed for.

The following symbols are predefined:

Predefined Symbol	Meaning
SRTK_INT_PHS_0	PHS-bus interrupt line 0
...	...
SRTK_INT_PHS_7	PHS-bus interrupt line 7
SRTK_INT_TIMER_A	Timer A interrupt
SRTK_INT_TIMER_B	Timer B interrupt
SRTK_INT_TIMER_D	Timer D interrupt
SRTK_INT_SERIAL_UART	Serial UART interrupt
SRTK_INT_MACROTICK	Macrotick interrupt
SRTK_INT_FWD_TIMER_A	Forwarded Timer A interrupt
SRTK_INT_FWD_TIMER_B	Forwarded Timer B interrupt
SRTK_INT_GL_0	Gigalink 0 interrupt
...	...
SRTK_INT_GL_3	Gigalink 3 interrupt
SRTK_INT_IO_ETH	I/O Ethernet interrupt

> **Note**
>
> Level-triggered interrupts (serial UART interrupts) have to be acknowledged in the interrupt service routine before they are enabled globally again.

Handler Specifies the pointer to the interrupt service routine.

Return value

This function returns the address of the interrupt service routine that was previously installed for this interrupt.

Example

The Timer A interrupt is to call the function `timera_interrupt` (see also srtk_start_isr_timerA).

First write the interrupt service routine `timera_interrupt`:

```
void timera_interrupt(void)
{
...
}
```

Then install the interrupt vector at the beginning of your application:

```
srtk_set_interrupt_vector(
   SRTK_INT_TIMER_A,
   (Srtk_Int_Handler_Type) timera_interrupt);
```

Related topics

References

RTLIB_INT_DISABLE

Syntax

```
RTLIB_INT_DISABLE()
```

Include file

`SrtkStd.h`

Purpose

To globally disable the interrupts.

> **Note**
>
> Use this macro only in conjunction with **`RTLIB_INT_ENABLE`**.

Return value

None

Related topics

References

RTLIB_INT_ENABLE 102
srtk_disable_hardware_int 89
srtk_disable_hardware_int_bm 90

RTLIB_INT_ENABLE

Syntax

```
RTLIB_INT_ENABLE()
```

Include file

`SrtkStd.h`

Purpose

To globally enable the interrupts.

Description

The only hardware interrupts that are available are the ones that are also enabled by srtk_enable_hardware_int.

Note

Use this macro only in conjunction with **`RTLIB_INT_DISABLE`**.

Return value

None

Related topics

References

RTLIB_INT_DISABLE 101
srtk_enable_hardware_int 91
srtk_enable_hardware_int_bm 92

RTLIB_INT_RESTORE

Syntax

```
void RTLIB_INT_RESTORE(UInt32 var_name)
```

Include file	`SrtkStd.h`
Purpose	To restore the previous interrupt state after calling **`RTLIB_INT_SAVE_AND_DISABLE`**. **Note** Use this macro only in conjunction with **`RTLIB_INT_SAVE_AND_DISABLE`**.
Parameters	**var_name** Returns the value of the previously executed macro **`RTLIB_INT_SAVE_AND_DISABLE`**.
Return value	None
Related topics	References

RTLIB_INT_SAVE_AND_DISABLE

Syntax	`RTLIB_INT_SAVE_AND_DISABLE(UInt32 var_name)`
Include file	`SrtkStd.h`
Purpose	To save the current interrupt status and globally disable the interrupts. **Note** Use this macro only in conjunction with **`RTLIB_INT_RESTORE`**.
Parameters	**var_name** Specifies the variable to store the interrupt status.

Return value

None

Example

```
void restore(void)
{
   UInt32 msr_state;

   RTLIB_INT_SAVE_AND_DISABLE(msr_state);
/* Save the value of the EE bit in MSR and disable interrupts*/
   ...
   RTLIB_INT_RESTORE(msr_state);
   /* Restore the EE bit in MSR at the end of the function*/
}
```

Related topics

References

RTLIB_INT_RESTORE 102

RTLIB_SRT_DISABLE

Syntax

`RTLIB_SRT_DISABLE()`

Include file

`SrtkStd.h`

Purpose

To disable the hardware interrupt for the sampling rate timer when the interrupts are still globally enabled (see **`RTLIB_INT_ENABLE`**).

Description

Timer A is used as the sampling rate timer for a DS1007 board. This function sets the corresponding bit of the Interrupt Mask Register (IMR).

Return value

None

Related topics

References

RTLIB_INT_ENABLE 102
RTLIB_SRT_ENABLE 105

RTLIB_SRT_ENABLE

Syntax	`RTLIB_SRT_ENABLE()`
Include file	`SrtkStd.h`
Purpose	To enable the hardware interrupt for the sampling rate timer.
Description	Timer A is used as the sampling rate timer for a DS1007 board. This function only clears the corresponding bit of the Interrupt Mask Register (IMR). However, the hardware interrupt for Timer A is available only when the interrupts are globally enabled (see **`RTLIB_INT_ENABLE`**).
Return value	None
Related topics	References RTLIB_INT_ENABLE .... 102 RTLIB_SRT_DISABLE 104

Subinterrupt Handling

Introduction

Subinterrupt handling provides functions to extend one hardware interrupt to multiple software subinterrupts.

Where to go from here

Information in this section

Basic Principles of Subinterrupt Handling

Introduction

In dSPACE multiprocessor systems, interrupts can be dispatched between processors. Typically, there is only one hardware line between processors. To allow multiple different interrupt signals to be sent from a sender to a receiver, a subinterrupt handling is provided which introduces logical interrupt sources. The subinterrupt handling meets the following goals:

- To trigger and handle multiple subinterrupts using a single hardware interrupt line.
- To allow that multiple different subinterrupts are pending at the receiver.
- To transmit and dispatch interrupts between several processors.
- To define interrupt senders/receivers to transmit subinterrupts.
- To use multiple senders and receivers at one processor.
- To get a point-to-point interrupt connection between two processors using a combination of sender and receiver.
- To make priority-based interrupt arbitration available (optional).
- Subinterrupts stay pending if they are disabled at the moment they occur.

Method

The following steps are necessary to program a subinterrupt handling between two applications:

1. Install a subinterrupt sender in your application that sends an interrupt.
2. Write an interrupt handler in your application that receives the interrupt.
3. Install a subinterrupt receiver in your application that receives the interrupt.

Example

See the following examples for more information:

- Example of Using a Subinterrupt Sender on page 107
- Example of Using a Subinterrupt Handler on page 108
- Example of Using a Subinterrupt Receiver on page 109

Example of Using a Subinterrupt Sender

Example

The following example shows the source code for the interrupt sender. It is defined for 16 subinterrupts. Every time the background loop is interrupted by timer 0, the subinterrupt 3 is sent to the receiver. The dual-port memory width is 16 bit and the accesses are direct.

```
#include <Brtenv.h>
#include <Defxxxx.h>       /* xxxx stands for the dSPACE */
#include <Mydefs.h>        /* board, e.g., 1401 for DS1401 */
dssint_sender_type *sender;
```

```
void isr_t0()
{
   dssint_interrupt(sender, 3);
}
void main()
{
   sender = dssint_define_int_sender_1(
      16,                       /* number of subinterrupts*/
      SUBINT_ADDR,              /* start address of int. info */
      ACK_ADDR,                 /* start address of ack. info */
      SENDER_ADDR,              /* trigger address */
      DPM_TARGET_DIRECT,        /* e.g., PHS bus base address */
      16,                       /* dual-port memory width */
      DPM_ACCESS_DIRECT,        /* pointer to write function */
      DPM_ACCESS_DIRECT);        /* pointer to read function */
   /* ... initialize timer 0 ... */
   global_enable();
   while(1);
}
```

Related topics

Basics

Examples

Example of Using a Subinterrupt Handler

Example

The example shows an interrupt handler for the dSPACE real-time kernel.

When the interrupt is triggered, the processor dispatches it to `my_handler`, where it is acknowledged by calling `dssint_acknowledge`. The function `dssint_decode` is called repetitively and returns the according subinterrupt number for every pending subinterrupt. For every subinterrupt, one task is registered by calling `rtk_register_task`.

`rtk_register_task` sets the task state for the according task to 'ready' when the task priority is not the highest of all registered tasks. The function internally stores the task registered with the highest priority and returns a pointer to it. `rtk_register_task` does not schedule tasks.

Once all tasks are registered, the “task” pointer holds the one with the highest priority. This task can be of a lower, equal or higher priority than the currently running task. Via the “task” pointer the scheduler is called – this is the reason why the state of the task registered with the highest priority must not be set to 'ready'.

The scheduler clears the stored information about the task registered with the highest priority.

```
void my_handler()
{
   rtk_p_task_control_block task = 0;
   int sub_int;
   dssint_acknowledge(receiver); /* interrupt acknowledge */
  /* Register tasks */
   do {
      if ( (sub_int = dssint_decode(receiver)) >= 0)
         task = rtk_register_task(S_MYSERVICE, sub_int);
   } while(subint >= 0);
  /* Call the scheduler */
   if (task)
      rtk_scheduler(task);
}
```

Related topics

Basics

Examples

Example of Using a Subinterrupt Receiver

Example

In this example, a receiver with 16 subinterrupts is defined. It is assumed that the kernel installs the function `my_handler` (refer to the Example of Using a Subinterrupt Handler on page 108) as an interrupt service routine for subinterrupts. The `main` function enables interrupts and enters the background task after creating and binding the tasks to the subinterrupts.

```
#include <Brtenv.h>
#include <Defxxxx.h>       /* xxxx stands for the dSPACE */
                           /* board, e.g. 1401 for DS1401 */
void slave0_task(void)
{
   /*...*/
};
dssint_receiver_type receiver;
```

```
void main()
{
   rtk_p_task_control_block task;
   receiver = dssint_define_int_receiver_1(
      16,                     /* number of subinterrupts*/
      SUBINT_ADDR,            /* start address of int. info */
      ACK_ADDR,               /* start address of ack. info */
      RECEIVER_ADDR,          /* receiver address */
      DPM_TARGET_DIRECT,      /* e.g. PHS bus base address */
      16,                     /* dual-port memory width */
      DPM_ACCESS_DIRECT,      /* pointer to write function */
      DPM_ACCESS_DIRECT);     /* pointer to read function */
   /* ... */
  task = rtk_create_task((rtk_task_fcn_type)slave0_task, 1,
            ovc_queue, rtk_default_overrun_fcn, 10,0);
   rtk_bind_interrupt(S_SLAVE, 0, task, 0.0, C_LOCAL, 0, 0);
   /*...*/
   global_enable();
   while(1);
}
```

Related topics

Basics

Examples

Data Types for Subinterrupt Handling

dssint_sender_type

```
typedef struct{
   unsigned int     nr_sint;     /* number of subinterrupts */
   unsigned long    sint_addr;   /* start address of the */
                                 /* interrupt info */
   unsigned long    ack_addr;    /* start address of the */
                                 /* acknowledge info */
   unsigned long    sender_addr; /* writing to this address */
                                 /* triggers interrupt */
   unsigned int     nr_words;    /* number of words */
                                 /* needed for nr_sint */
   unsigned long*   request;     /* pointer to local copy */
                                 /* of sint_addr */
   long             target;      /* e.g. PHS bus base address */
   unsigned int     sint_mem_width;
                                 /* width of the*/
                                 /* dual-port memory */
   dpm_write_fcn_t write_fcn;    /* pointer to write function */
   dpm_read_fcn_t   read_fcn;    /* pointer to read function */
   unsigned int     sint_mem_shift;
                                 /* internal performance */
                                 /* improvement */
}dssint_sender_type;
```

dssint_receiver_type

```
typedef struct{
   unsigned int      nr_sint;     /* number of subinterrupts */
   unsigned long     sint_addr;   /* start address of the */
                                  /* interrupt info */
   unsigned long     ack_addr;    /* start address of the */
                                  /* acknowledge info */
   unsigned long     receiver_addr;
                                  /* reading from this address */
                                  /* performs hardware ack of */
                                  /* interrupt */
   unsigned int       nr_words;   /* number of words */
                                  /* needed for nr_sint */
   unsigned long*    acknowledge;
                                  /* pointer to local copy */
                                  /* of ack_addr */
   unsigned long*    state;       /* pointer to state info */
   long              target;      /* e.g. PHS bus base address */
   unsigned int      sint_mem_width;
                                  /* width of the */
                                  /* dual-port memory */
   unsigned int      state_position;
                                  /* decode position in state */
   dpm_write_fcn_t   write_fcn;   /* pointer to write function */
   dpm_read_fcn_t    read_fcn;    /* pointer to read function */
   unsigned int      sint_mem_shift;
                                  /* internal performance */
                                  /* improvement */
   unsigned long*    enable_flag; /* for pending interrupts */
   dssint_ack_fcn_t  ack_fcn;     /* pointer to interrupt acknowledge function */
}dssint_receiver_type;
```

Related topics

Basics

Examples

dssint_define_int_sender

Syntax

```
dssint_sender_type* dssint_define_int_sender(
      unsigned int nr_subinterrupts,
      unsigned long subint_addr,
      unsigned long ack_addr,
      unsigned long sender_addr,
      long target,
      unsigned int sint_mem_width,
      dpm_write_fcn_t write_fcn,
      dpm_read_fcn_t read_fcn)
```

Include file

`dssint.h`

Purpose

To define the sender of a subinterrupt.

Description

The function defines an interrupt sender and returns a handle to it. A sender processor can have multiple receiver processors to pass interrupts to. The handle identifies where to send an interrupt. The function initializes all memory locations in the dual-port memory used for the subinterrupt handling with 0.

The functions `dssint_define_int_sender` and `dssint_define_int_receiver` define the sender and receiver of a subinterrupt in the following way:

When subinterrupts are sent before the receiver is initialized, these interrupts are stored. After the receiver is initialized these interrupts are passed to the receiver and processed.

Note

- The behavior described above can cause overflows. To avoid this, use the functions `dssint_define_int_sender_1` and `dssint_define_int_receiver_1` instead.
- If you define a sender of a subinterrupt via the function `dssint_define_int_sender`, you must define the receiver via the function `dssint_define_int_receiver`.

Parameters

nr_subinterrupts Specifies the number of different subinterrupts to be transferred. This is necessary to define the width of the memory portion which passes the subinterrupt information. The number of subinterrupts must be equal for sender and receiver.

subint_addr Specifies the memory location the subinterrupt information is passed to.

ack_addr Specifies the memory location the acknowledgment information from the receiver is passed to.

sender_addr Specifies the pointers to the memory location that triggers the interrupt by writing to it (hardware trigger). This address can be the same as subint_addr.

target Specifies the address of the target memory, for example, a PHS bus address or COM port number. This parameter is meaningless for direct access.

sint_mem_width Specifies the width of the dual-port memory.

write_fcn Specifies the address of a function that performs a write access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

read_fcn Specifies the address of a function that performs a read access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

Return value

This function returns the handle to an interrupt sender. The function returns 0 if an error occurred.

Example

See Example of Using a Subinterrupt Sender on page 107.

Related topics

Basics

Examples

References

dssint_define_int_sender_1

Syntax

```
dssint_sender_type* dssint_define_int_sender_1(
      unsigned int nr_subinterrupts,
      unsigned long subint_addr,
      unsigned long ack_addr,
      unsigned long sender_addr,
      long target,
      unsigned int sint_mem_width,
      dpm_write_fcn_t write_fcn,
      dpm_read_fcn_t read_fcn)
```

Include file

`dssint.h`

Purpose

To define the sender of a subinterrupt.

Description

The function defines an interrupt sender and returns a handle to it. A sender processor can have multiple receiver processors to pass interrupts to. The handle identifies where to send an interrupt. The function initializes all memory locations in the dual-port memory used for the subinterrupt handling with 0.

The functions `dssint_define_int_sender_1` and `dssint_define_int_receiver_1` define the sender and receiver of a subinterrupt in the following way:

When subinterrupts are sent before the receiver is initialized, these interrupts are not stored to avoid overflows.

Note

If you define a sender of a subinterrupt via the function `dssint_define_int_sender_1`, you have to define the receiver via the function `dssint_define_int_receiver_1`.

Parameters

nr_subinterrupts Specifies the number of different subinterrupts to be transferred. The number of subinterrupts must be equal for sender and receiver. See dssint_define_int_sender on page 112.

subint_addr Specifies the memory location the subinterrupt information is passed to.

ack_addr Specifies the memory location the acknowledgment information from the receiver is passed to.

sender_addr Specifies the pointers to the memory location that triggers the interrupt by writing to it (hardware trigger). This address can be the same as subint_addr.

target Specifies the address of the target memory, for example, a PHS bus address or COM port number. This parameter is meaningless for direct access.

sint_mem_width Specifies the width of the dual-port memory.

write_fcn Specifies the address of a function that performs a write access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

read_fcn Specifies the address of a function that performs a read access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

Return value

This function returns the handle to an interrupt sender. The function returns 0 if an error occurred.

Example

See Example of Using a Subinterrupt Sender on page 107.

Related topics

Basics

Examples

References

dssint_define_int_receiver

Syntax

```
dssint_receiver_type *dssint_define_int_receiver(
      unsigned int nr_subinterrupts,
      unsigned long subint_addr,
      unsigned long ack_addr,
      unsigned long receiver_addr,
      long target,
      unsigned int sint_mem_width,
      dpm_write_fcn_t write_fcn,
      dpm_read_fcn_t read_fcn)
```

Include file

`dssint.h`

Purpose

To define the receiver of a subinterrupt.

Description

The function reads from the `receiver_addr` to enable interrupt triggering by the sender. It defines an interrupt receiver and returns a handle to it. A receiver processor can have multiple sender processors from which interrupts are retrieved. The handle identifies the appropriate subinterrupt vector and receiving information table for a specific sender.

The functions `dssint_define_int_receiver` and `dssint_define_int_sender` define the receiver and sender of a subinterrupt in the following way:

When subinterrupts are sent before the receiver is initialized, these interrupts are stored. After the receiver is initialized, these interrupts are passed to the receiver and processed.

Note

- The behavior described above can cause overflows. To avoid this, use the functions `dssint_define_int_sender_1` and `dssint_define_int_receiver_1` instead.
- If you define a receiver of a subinterrupt via the function `dssint_define_int_receiver`, you have to define the sender via the function `dssint_define_int_sender`.

Parameters

nr_subinterrupts Specifies the number of different subinterrupts to be transferred. The number of subinterrupts must be equal for sender and receiver. See dssint_define_int_sender on page 112.

subint_addr Specifies the memory location the subinterrupt information is passed to.

ack_addr Specifies the memory location the acknowledgment information from the receiver is passed to.

receiver_addr Specifies the pointers to the memory location that acknowledges the interrupt by reading it (hardware acknowledge).

target Specifies the address of the target memory, for example, a PHS bus address or COM port number. This parameter is meaningless for direct access.

sint_mem_width Specifies the width of the dual-port memory.

write_fcn Specifies the address of a function that performs a write access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

read_fcn Specifies the address of a function that performs a read access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

Return value

This function returns the address of an interrupt receiver. The function returns 0 if an error occurred.

Example

See Example of Using a Subinterrupt Receiver on page 109.

Related topics

Basics

Examples

References

dssint_define_int_receiver_1

Syntax

```
dssint_receiver_type *dssint_define_int_receiver_1(
      unsigned int nr_subinterrupts,
      unsigned long subint_addr,
      unsigned long ack_addr,
      unsigned long receiver_addr,
      long target,
      unsigned int sint_mem_width,
      dpm_write_fcn_t write_fcn,
      dpm_read_fcn_t read_fcn)
```

Include file

`dssint.h`

Purpose

To define the receiver of a subinterrupt.

Description

The function reads from the `receiver_addr` to enable interrupt triggering by the sender. It defines an interrupt receiver and returns a handle to it. A receiver processor can have multiple sender processors from which interrupts are retrieved. The handle identifies the appropriate subinterrupt vector and receiving information table for a specific sender.

The functions `dssint_define_int_receiver_1` and `dssint_define_int_sender_1` define the receiver and sender of a subinterrupt in the following way:

When subinterrupts are sent before the receiver is initialized, these interrupts will not be stored to avoid overflows.

Note

If you define a receiver of a subinterrupt via the function `dssint_define_int_receiver_1`, you must define the sender via the function `dssint_define_int_sender_1`.

Parameters

nr_subinterrupts Specifies the number of different subinterrupts to be transferred. The number of subinterrupts must be equal for sender and receiver. See dssint_define_int_sender on page 112.

subint_addr Specifies the memory location the subinterrupt information is passed to.

ack_addr Specifies the memory location the acknowledgment information from the receiver is passed to.

receiver_addr Specifies the pointers to the memory location that acknowledges the interrupt by reading it (hardware acknowledge).

target Specifies the address of the target memory, for example, a PHS bus address or COM port number. This parameter is meaningless for direct access.

sint_mem_width Specifies the width of the dual-port memory.

write_fcn Specifies the address of a function that performs a write access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

read_fcn Specifies the address of a function that performs a read access to the dual-port memory. Set this parameter to 0 for direct access (if all supported memories that are accessed directly have the same width).

Return value

This function returns the address of an interrupt receiver. The function returns 0 if an error occurred.

Example

See Example of Using a Subinterrupt Receiver on page 109.

Related topics

Basics

Examples

References

dssint_subint_disable

Syntax

```
void dssint_subint_disable(
     dssint_receiver_type *receiver,
     unsigned int subinterrupt)
```

Include file

`dssint.h`

Purpose

To disable a subinterrupt.

Description

After initialization, all subinterrupts are enabled. You must disable the subinterrupt explicitly via this function.

Parameters

receiver Specifies the receiver handler the subinterrupt is located in.

subinterrupt Specifies the subinterrupt to reset.

Example

```
...
dssint_subint_disable(my_receiver, 5);
...
```

Related topics

Basics

References

dssint_subint_enable

Syntax

```
void dssint_subint_enable(
     dssint_receiver_type *receiver,
     unsigned int subinterrupt)
```

Include file

`dssint.h`

Purpose

To enable a subinterrupt.

Description

After initialization, all subinterrupts are enabled. Use this function if you disabled a subinterrupt via `dssint_subint_disable` before.

Parameters

receiver Specifies the receiver handler the subinterrupt is located in.

subinterrupt Specifies the subinterrupt to reset.

Example

```
...
dssint_subint_enable(my_receiver, 5);
...
```

Related topics

Basics

References

dssint_interrupt

Syntax

```
void dssint_interrupt(
      dssint_sender_type *sender,
      unsigned int sub_interrupt)
```

Include file

`dssint.h`

Purpose

To write the subinterrupt information to the specified memory location and to trigger the interrupt.

Parameters

sender Specifies the handle of the interrupt sender.

sub_interrupt Specifies the subinterrupt to be triggered. Values are within the range 0 … `nr_subinterrupts`. Parameter `nr_subinterrupts` is defined by `dssint_define_int_sender` (or `dssint_define_int_sender_1`) and `dssint_define_int_receiver` (or `dssint_define_int_receiver_1`).

Example

See Example of Using a Subinterrupt Sender on page 107.

Related topics

Basics

Examples

References

dssint_decode

Syntax

```
int dssint_decode(dssint_receiver_type *receiver)
```

Include file	`dssint.h`
Purpose	To identify the pending interrupts.
Description	This function is called repetitively within an interrupt handler. It processes the interrupt information of the receiver data structure that was given by `dssint_acknowledge`, determines the pending subinterrupt with the highest priority and returns it to the handler. The pending subinterrupt with the highest priority is the one with the smallest subinterrupt number.
Parameters	**receiver** Specifies the receiver handler the subinterrupt is located in.
Return value	This function returns the number of the pending subinterrupt with highest priority. If there is no pending subinterrupt left, the function returns SINT_NO_SUBINT ("–1").
Example	See Example of Using a Subinterrupt Handler on page 108.

Related topics

Basics

Examples

References

dssint_acknowledge

Syntax

```
void dssint_acknowledge(dssint_receiver_type *receiver)
```

Include file `dssint.h`

Purpose

To acknowledge pending subinterrupts.

Description

This function acknowledges the interrupt by reading `receiver->receiver_addr` (hardware acknowledge), and copies the subinterrupt information to the receiver data structure. Then it performs the software acknowledgment for every pending subinterrupt.

For information on the receiver data structure, refer to the type definition given in Data Types for Subinterrupt Handling on page 111.

Parameters

receiver	Specifies the receiver handler the subinterrupt is located in.

Example

See Example of Using a Subinterrupt Handler on page 108.

Related topics

Basics

Examples

References

dssint_subint_reset

Syntax

```
void dssint_subint_reset(
     dssint_receiver_type *receiver,
     unsigned int subinterrupt)
```

Include file

`dssint.h`

Purpose

To clear a pending subinterrupt.

Parameters

receiver Specifies the receiver handler the subinterrupt is located in.

subinterrupt Specifies the subinterrupt to reset.

Example

```
...
dssint_subint_reset(my_receiver, 5);
...
```

Related topics

Basics

Basic Principles of Subinterrupt Handling........107

References

Data Types for Subinterrupt Handling........111
dssint_subint_disable........120
dssint_subint_enable........121

Message Handling

Purpose

To configure and generate messages.

Where to go from here

Information in this section

Basic Principles of Message Handling

Introduction

The Message module provides functions to generate error, warning, and information messages to be displayed by the dSPACE experiment software. Messages are generated by the processor board, written to a message buffer, and sent to the host PC. On the host PC, the dSPACE experiment software displays the messages in the log window and writes them to the log file. Each message consists of a message number and the message string. To use the message module, you have to initialize the board via the initialization function `init()`.

Message characteristics

There are two predefined symbols that define the message buffer. The symbol `MSG_STRING_LENGTH` specifies the maximum length of a generated message. If a message exceeds the given length, it is truncated. The symbol `MSG_BUFFER_LENGTH` specifies the maximum number of messages that can be stored to the reserved memory. The behavior of the message buffer is controlled by the `msg_mode_set` function. The values of the message and buffer lengths are defined in `MsgXXXX.h` (`XXXX` denotes the relevant dSPACE board) or `StrkMsg.h` when you use DS1007 or MicroLabBox.

For the DS1007 PPC Processor Board, the following values are used:

Predefined Symbol	Default Value
MSG_STRING_LENGTH	480 characters
MSG_BUFFER_LENGTH	256 messages

Message types

There are four message types:

Type	Representation in the dSPACE Experiment Software
ERROR	Dialog box containing the message text and entry in the Log window beginning with ERROR
WARNING	Entry in the Log window beginning with WARNING

Type	Representation in the dSPACE Experiment Software
INFO	Entry in the Log window
LOG	Entry in the Log file only

The following table gives examples for the three message types ERROR, WARNING, and INFO:

Module	Message Type	Board Name	Submodule	Message Text
`Platform:`	`ERROR`			`Board is not present or expansion box is off.`
`DataKernel:`	`WARNING`			`Data connection not valid!`
`Real-Time Processor:`		`#1 DS1007 -`	`RTLib:`	`System started. (0)`

Data Types and Symbols for Message Handling

Data types

The following data types are defined:

msg_string_type

```
typedef char msg_string_type;
```

msg_no_type

```
typedef Int32 msg_no_type;
```

msg_class_type

```
typedef enum msg_class_type;
```

msg_dialog_type

```
typedef enum msg_dialog_type;
```

msg_submodule_type

```
typedef UInt32 msg_submodule_type;
```

msg_hookfcn_type

```
typedef int (*msg_hookfcn_type)(msg_submodule_type, msg_no_type);
```

The following symbols are defined:

Predefined Symbol	Message refers to ...
`MSG_SM_NONE`	No specific module (default)
`MSG_SM_USER`	User messages
`MSG_SM_CAN1401`	RTLib: CAN (DS1401)
`MSG_SM_CAN2202`	RTLib: CAN (DS2202)
`MSG_SM_CAN2210`	RTLib: CAN (DS2210)

Predefined Symbol	Message refers to ...
MSG_SM_CAN2211	RTLib: CAN (DS2211)
MSG_SM_CAN4302	RTLib: CAN (DS4302)
MSG_SM_DIO1401	RTLib: Digital I/O (DS1401)
MSG_SM_DS1104SLVLIB	RTLib: Slave DSP (DS1104)
MSG_SM_DS4501	RTLib: DS4501 functions
MSG_SM_DS4502	RTLib: DS4502 functions
MSG_SM_DSBYPASS	RTI: Bypass Blockset
MSG_SM_DSCAN	RTLib: CAN support
MSG_SM_DSETH	RTI: RTI Ethernet Blockset
MSG_SM_DSFR	RTLib: FlexRay support
MSG_SM_DSJ1939	J1939 Support in RTI CAN MultiMessage Blockset
MSG_SM_DSSER	RTLib: Serial interface
MSG_SM_ECU_POD	ECU PODs (DS5xx)
MSG_SM_ECU1401	RTLib: ECU interface (DS1401)
MSG_SM_HOSTSERV	Host services
MSG_SM_LIN	RTLib: LIN support
MSG_SM_REALMOTION	RealMotion / MotionDesk
MSG_SM_RTI	Real-Time Interface
MSG_SM_RTICAN	RTI: CAN Blockset
MSG_SM_RTICAN1401	RTI: CAN Blockset (DS1401)
MSG_SM_RTICAN2202	RTI: CAN Blockset (DS2202)
MSG_SM_RTICAN2210	RTI: CAN Blockset (DS2210)
MSG_SM_RTICAN2211	RTI: CAN Blockset (DS2211)
MSG_SM_RTICAN4302	RTI: CAN Blockset (DS4302)
MSG_SM_RTICANMM	RTI: CAN MultiMessage Blockset
MSG_SM_RTIFLEXRAY	RTI: FlexRay Blockset
MSG_SM_RTIFLEXRAYCONFIG	RTI: FlexRay Configuration Blockset
MSG_SM_RTILINMM	RTI: LIN MultiMessage Blockset
MSG_SM_RTIMP	RTI-MP (Real-Time Interface for multiprocessor systems)
MSG_SM_RTKERNEL	Real-Time Kernel
MSG_SM_RTLIB	Real-Time Board Library
MSG_SM_RTOSAL	RTOS Abstractionlayer
MSG_SM_RTPYTHON	RTPythoninterpreter
MSG_SM_SIMENG	RTI: Simulation engine

msg_error_set

Syntax

```
void msg_error_set(
   msg_submodule_type module,
   msg_no_type msg_no,
   msg_string_type *msg)
```

Include file

`dsmsg.h`

Purpose

To generate an error message.

> **Note**
>
> If there is a hook function installed (see **`msg_error_hook_set`**), the hook function is called before the error message is generated.

Parameters

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \dots 2^{31}-1$ defined by the user.

msg Specifies the message string (for information on the maximum length, see Message characteristics on page 127).

Return value

None

Related topics

Basics

References

msg_warning_set

Syntax

```
void msg_warning_set(
   msg_submodule_type module,
   msg_no_type msg_no,
   msg_string_type *msg)
```

Include file

`dsmsg.h`

Purpose

To generate a warning message.

Parameters

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \dots 2^{31}-1$ defined by the user.

msg Specifies the message string (for information on the maximum length, see Message characteristics on page 127).

Return value

None

Related topics

Basics

References

msg_info_set

Syntax

```
void msg_info_set(
   msg_submodule_type module,
   msg_no_type msg_no,
   msg_string_type *msg)
```

Include file `dsmsg.h`

Purpose To generate an information message.

Parameters

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \ldots 2^{31}-1$ defined by the user.

msg Specifies the message string (for information on the maximum length, see Message characteristics on page 127).

Return value None

Related topics

Basics

References

msg_set

Syntax

```
void msg_set(
   msg_class_type msg_class,
   msg_dialog_type msg_dialog,
   msg_submodule_type module,
   msg_no_type msg_no,
   msg_string_type *msg)
```

Include file `dsmsg.h`

Purpose To generate a message of the defined message class.

Description

This function issues an error, information, or warning message that is displayed by the dSPACE experiment software, or a message that only appears in the log file. In addition to the other `msg_xxx_set` functions, the user can adjust the type of the message dialogs.

Parameters

msg_class Specifies the type of the message. The following symbols are predefined:

Predefined Symbol	Meaning
`MSG_MC_ERROR`	Error message
`MSG_MC_INFO`	Information message
`MSG_MC_WARNING`	Warning message
`MSG_MC_LOG`	Message appears only in the log file

msg_dialog Specifies the type of the dialog. The following types are predefined:

Predefined Symbol	Meaning
`MSG_DLG_NONE`	No dialog, silent mode
`MSG_DLG_OKCANCEL`	OK/Cancel dialog
`MSG_DLG_DEFAULT`	Dialog type specified by **`msg_default_dialog_set`**

Note

If you use a DS1007 PPC Processor Board, displaying messages in a dialog in ControlDesk is not supported. This parameter is not used and only provided for backward compatibility. All messages are displayed in the standard log.

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \ldots 2^{31}-1$ defined by the user.

msg Specifies the message string (for information on the maximum length, see Message characteristics on page 127).

Return value

None

Example

The following example issues an error message without a dialog.

```
msg_set(
      MSG_MC_ERROR,
      MSG_DLG_NONE,
      MSG_SM_USER,
      1,
      "This is an error message.");
```

Related topics

Basics

References

msg_error_printf

Syntax

```
int msg_error_printf(
   msg_submodule_typemodule,
   msg_no_typemsg_no,
   char *format,
   arg1, arg2, etc.)
```

Include file

`dsmsg.h`

Purpose

To generate an error message with arguments using the `printf` format (see a standard C documentation).

Result

`printf` builds the message string with the standard C command arguments of `printf(char *format, arg1, arg2, etc.)`. The string is then automatically given to `msg_error_set` to generate the message.

> **Note**
>
> If there is a hook function installed (see msg_error_hook_set on page 146), the hook function is called before the error message is generated.

Parameters

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \ldots 2^{31}-1$ defined by the user.

format Specifies the string using the `printf` format.

arg1, arg2, etc. Specifies the optional arguments for the format string (see a standard C documentation).

> **Note**
>
> The length of the format string is not restricted, but the default value of the maximum length is specified by `MSG_STRING_LENGTH`, see Message characteristics on page 127. Longer messages are truncated.

Return value

This function returns the number of characters which were printed to the message buffer.

Example

This example shows how to generate an error message with the printf format:

```
#include <Brtenv.h>
/* An example integer value */
int num = 13;
void main()
{
   /* Initialization of the board */
   init();
   /* Write an error message to the message buffer using the printf format */
   msg_error_printf(MSG_SM_USER, 1, "The value of num is %i", num);
}
```

Related topics

Basics

References

msg_warning_printf

Syntax

```
int msg_warning_printf(
   msg_submodule_typemodule,
   msg_no_typemsg_no,
   char *format,
   arg1, arg2, etc.)
```

Include file

`dsmsg.h`

Purpose

To generate a warning message with arguments using the `printf` format (see a standard C documentation).

Result

`printf` builds the message string with the standard C command arguments of `printf(char *format, arg1, arg2, etc.)`. The string is then automatically passed to `msg_warning_set` to generate the message.

Parameters

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of -2^{31} … 2^{31}-1 defined by the user.

format Specifies the string using the `printf` format.

arg1, arg2, etc. Specifies the optional arguments for the format string (see a standard C documentation).

Note

The length of the format string is not restricted, but the default value of the maximum length is specified by `MSG_STRING_LENGTH`, see Message characteristics on page 127. Longer messages are truncated.

Return value

This function returns the number of characters which were printed to the message buffer.

Related topics

Basics

References

msg_info_printf

Syntax

```
int msg_info_printf(
   msg_submodule_typemodule,
   msg_no_typemsg_no,
   char *format,
   arg1, arg2, etc.)
```

Include file

`dsmsg.h`

Purpose

To generate an information message with arguments using the `printf` format (see a standard C documentation).

Result

`printf` builds the message string with the standard C command arguments of `printf(char *format, arg1, arg2, etc.)`. The string is then automatically given to `msg_info_set` to generate the message.

Parameters

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \ldots 2^{31}-1$ defined by the user.

format Specifies the string using the `printf` format.

arg1, arg2, etc. Specifies the optional arguments for the format string (see a standard C documentation).

Note

The length of the format string is not restricted, but the default value of the maximum length is specified by `MSG_STRING_LENGTH`, see Message characteristics on page 127. Longer messages are truncated.

Return value

This function returns the number of characters which were printed to the message buffer.

Related topics

Basics

References

msg_printf

Syntax

```
int msg_printf(
   msg_class_typemsg_class,
   msg_dialog_typemsg_dialog,
   msg_submodule_typemodule,
   msg_no_typemsg_no,
   char *format,
   arg1, arg2, etc.)
```

Include file

`dsmsg.h`

Purpose

To generate a message of the specified class with arguments using the `printf` format (see a standard C documentation).

Result

`printf` builds the message string with the standard C command arguments of `printf(char *format, arg1, arg2, etc.)`. The string is then automatically given to **`msg_set`** to generate the message.

Parameters

msg_class Specifies the type of the message. The following symbols are predefined:

Predefined Symbol	Meaning
`MSG_MC_ERROR`	Error message
`MSG_MC_INFO`	Information message
`MSG_MC_WARNING`	Warning message
`MSG_MC_LOG`	Message appears only in the log file

msg_dialog Specifies the type of the dialog. The following types are predefined:

Predefined Symbol	Meaning
`MSG_DLG_NONE`	No dialog, silent mode
`MSG_DLG_OKCANCEL`	OK/Cancel dialog
`MSG_DLG_DEFAULT`	Dialog type specified by **`msg_default_dialog_set`**

> **Note**
>
> If you use a DS1007 PPC Processor Board, displaying messages in a dialog in ControlDesk is not supported. This parameter is not used and only provided for backward compatibility. All messages are displayed in the standard log.

module Specifies the predefined symbol of the application module generating the message. Use the module type `MSG_SM_USER` only for handcoded programs. For a list of all predefined symbols, refer to Data Types and Symbols for Message Handling on page 128.

msg_no Specifies the number of the message within the range of $-2^{31} \dots 2^{31}-1$ defined by the user.

format Specifies the string using the `printf` format.

arg1, arg2, etc. Specifies the optional arguments for the format string (see a standard C documentation).

> **Note**
>
> The length of the format string is not restricted, but the default value of the maximum length is specified by `MSG_STRING_LENGTH`, see Message characteristics on page 127. Longer messages are truncated.

Return value

This function returns the number of characters which were printed to the message buffer.

Related topics

Basics

References

msg_default_dialog_set

Syntax

```
void msg_default_dialog_set(
   msg_class_type msg_class,
   msg_dialog_type msg_dialog)
```

Include file

`dsmsg.h`

Purpose

To specify the default dialog type for the selected message class.

Note

If you use a DS1007 PPC Processor Board, displaying messages in a dialog in ControlDesk is not supported. This function is not used and only provided for backward compatibility. All messages are displayed in the standard log.

Result

The message module functions `msg_xxx_set` and `msg_xxx_printf` always use the specified default dialog type. The dialog type of the functions **`msg_set`** and **`msg_printf`** is set to the default type when they are calling with the `msg_dialog` argument `MSG_DLG_DEFAULT`.

Parameters

msg_class Specifies the type of the message. The following symbols are predefined:

Predefined Symbol	Meaning
`MSG_MC_ERROR`	Error message
`MSG_MC_INFO`	Information message

Predefined Symbol	Meaning
MSG_MC_WARNING	Warning message
MSG_MC_LOG	Message appears only in the log file

msg_dialog Specifies the type of the dialog. The following types are predefined:

Predefined Symbol	Meaning
MSG_DLG_NONE	No dialog, silent mode
MSG_DLG_OKCANCEL	OK/Cancel dialog

Return value

None

Related topics

Basics

Basic Principles of Message Handling.. 127

msg_mode_set

Syntax

```
void msg_mode_set(UInt32 mode)
```

Include file

`dsmsg.h`

Purpose

To set the mode of the message buffer.

Description

This function specifies the behavior of the message buffer if the number of messages exceeds the maximum buffer length. On start-up, the overwrite mode is active.

Parameters

mode Specifies the mode of the message buffer. The following symbols are predefined:

Predefined Symbol	Meaning
MSG_BLOCKING	The message buffer will be filled to the maximum number of entries. Any further messages will be lost.
MSG_OVERWRITE	The message buffer will be filled cyclically. The oldest message will be overwritten when the buffer is full.

Return value	None
Related topics	Basics

msg_reset

Syntax

```
void msg_reset()
```

Include file `dsmsg.h`

Purpose To reset the message buffer and clear the values of the last error (see `msg_error_clear`).

Description The next message will be the first entry in the message buffer. Nevertheless, the message number will be incremented.

Return value None

Related topics

Basics

References

msg_last_error_number

Syntax

```
msg_no_type msg_last_error_number()
```

Include file	`dsmsg.h`
Purpose	To read the number of the last generated error message.
Description	Independently of the order of the messages in the message buffer, this function returns the number of the last error message. On start-up, the value is set to 0. **Note** Warning and information messages do not change this number.
Return value	This function returns the number of the last generated error message.
Related topics	Basics References

msg_last_error_submodule

Syntax	`msg_submodule_type msg_last_error_submodule()`
Include file	`dsmsg.h`
Purpose	To read the submodule of the last generated error message.

Description

On start-up, the value is set to `MSG_SM_NONE` (see table below).

Note

Warning and information messages do not change this value.

Return value

This function returns the submodule of the last generated error message. The following symbols are defined:

Predefined Symbol	Message refers to ...
MSG_SM_NONE	No specific module (default)
MSG_SM_USER	User messages
MSG_SM_CAN1401	RTLib: CAN (DS1401)
MSG_SM_CAN2202	RTLib: CAN (DS2202)
MSG_SM_CAN2210	RTLib: CAN (DS2210)
MSG_SM_CAN2211	RTLib: CAN (DS2211)
MSG_SM_CAN4302	RTLib: CAN (DS4302)
MSG_SM_DIO1401	RTLib: Digital I/O (DS1401)
MSG_SM_DS1104SLVLIB	RTLib: Slave DSP (DS1104)
MSG_SM_DS4501	RTLib: DS4501 functions
MSG_SM_DS4502	RTLib: DS4502 functions
MSG_SM_DSBYPASS	RTI: Bypass Blockset
MSG_SM_DSCAN	RTLib: CAN support
MSG_SM_DSETH	RTI: RTI Ethernet Blockset
MSG_SM_DSFR	RTLib: FlexRay support
MSG_SM_DSJ1939	J1939 Support in RTI CAN MultiMessage Blockset
MSG_SM_DSSER	RTLib: Serial interface
MSG_SM_ECU_POD	ECU PODs (DS5xx)
MSG_SM_ECU1401	RTLib: ECU interface (DS1401)
MSG_SM_HOSTSERV	Host services
MSG_SM_LIN	RTLib: LIN support
MSG_SM_REALMOTION	RealMotion / MotionDesk
MSG_SM_RTI	Real-Time Interface
MSG_SM_RTICAN	RTI: CAN Blockset
MSG_SM_RTICAN1401	RTI: CAN Blockset (DS1401)
MSG_SM_RTICAN2202	RTI: CAN Blockset (DS2202)
MSG_SM_RTICAN2210	RTI: CAN Blockset (DS2210)
MSG_SM_RTICAN2211	RTI: CAN Blockset (DS2211)
MSG_SM_RTICAN4302	RTI: CAN Blockset (DS4302)

Predefined Symbol	Message refers to ...
`MSG_SM_RTICANMM`	RTI: CAN MultiMessage Blockset
`MSG_SM_RTIFLEXRAY`	RTI: FlexRay Blockset
`MSG_SM_RTIFLEXRAYCONFIG`	RTI: FlexRay Configuration Blockset
`MSG_SM_RTILINMM`	RTI: LIN MultiMessage Blockset
`MSG_SM_RTIMP`	RTI-MP (Real-Time Interface for multiprocessor systems)
`MSG_SM_RTKERNEL`	Real-Time Kernel
`MSG_SM_RTLIB`	Real-Time Board Library
`MSG_SM_RTOSAL`	RTOS Abstractionlayer
`MSG_SM_RTPYTHON`	RTPythoninterpreter
`MSG_SM_SIMENG`	RTI: Simulation engine

Related topics

Basics

References

msg_error_clear

Syntax

```
void msg_error_clear()
```

Include file

`dsmsg.h`

Purpose

To set the number of the last generated error to 0 and the submodule of the last generated error message to `MSG_SM_NONE` (refer to Data Types and Symbols for Message Handling on page 128).

Return value

None

Related topics

Basics

References

msg_error_hook_set

Syntax

```
void msg_error_hook_set(msg_hookfcn_type hook)
```

Include file

`dsmsg.h`

Purpose

To install a hook function.

Description

The hook function is activated when an error message is generated (see **`msg_error_set`** and **`msg_error_printf`**) and before the message is displayed.

Use the hook function to:

- React to an error (for example, to implement an error correction function)
- Suppress the error message

The hook function is activated for all errors. To react only for certain submodules or message numbers, you have to manage restrictions within your handcoded function (see example below).

Parameters

hook Specifies the pointer to the hook function.

Return value

This function returns one of the following values:

Value	Meaning
1	The error message is displayed.
0	The error message is not displayed.

Example

This example shows how to use a hook function:

```
#include <Brtenv.h>
int error_hook_function(msg_submodule_type sm, msg_no_type no)
{
   if ((sm == MSG_SM_RTI) && (no == 1))
   {
      /* suppress error message */
      return(0);
   } else
   {
      /* display error message */
   return(1);
   }
}
void main()
{
   /* Initialization of the board */
   init();
   /* Announce the hook function to the message module */
   msg_error_hook_set(error_hook_function);
  /* Write an error message to the message buffer */
   msg_error_set(MSG_SM_USER, 1, "user error message");
   /* This error message will be suppressed by the
      hook function */
   msg_error_set(MSG_SM_RTI, 1, "RTI error message");
}
```

Related topics

Basics

Basic Principles of Message Handling..127

msg_init

Syntax

```
void msg_init(void)
```

Include file

`dsmsg.h`

Purpose

To initialize the message handling.

Description

This function is called automatically during the board initialization. The mode is set to MSG_OVERWRITE, counter and indices are set to 0. The buffer and string lengths are set according to the values of `MSG_BUFFER_LENGTH` and `MSG_STRING_LENGTH` defined in `SrtkMsg.h.`

Return value None

Related topics

Basics

Basic Principles of Message Handling 127

References

msg_mode_set 141

Serial Interface Communication

Introduction

This section contains the generic functions for communication via a serial interface.

The generic functions use a receive and transmit buffer to buffer the data. Because they do not have direct access to the UART, they are hardware-independent and can be used for different I/O boards. These generic functions are described in this chapter.

Where to go from here

Information in this section

Basic Principles of Serial Communication

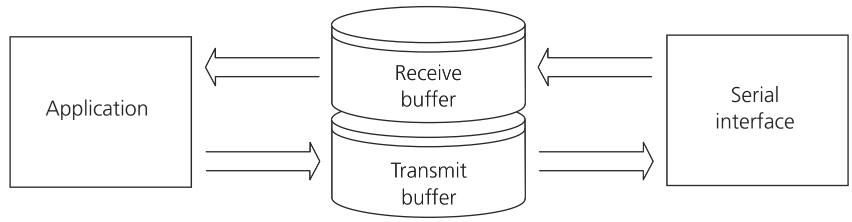
Where to go from here

Information in this section

Software FIFO Buffer

Introduction

The software FIFO buffer is a memory section that provides the UART with additional space for data storage and ensures that the generic functions are hardware-independent.



The software FIFO buffer stores data that will be written to (transmit buffer) or has been read by (receive buffer) the UART.

Buffer size

The buffer size must be a power of two (2^n) and at least 64 bytes great. The maximum size depends on the available memory.

Transmit buffer

The transmit buffer is filled with data to be sent as long as free space is available. It cannot be overwritten. You can write data to the transmit buffer with the function `dsser_transmit`.

Receive buffer

The receive buffer is filled with data received by the UART as long as free space is available. If an overflow occurs, old data in the receive buffer is overwritten or new data is rejected. This depends on the mode of the FIFO. You can access the receive buffer by using the functions `dsser_receive` and `dsser_receive_term`.

Related topics

Basics

Trigger Levels 150

References

dsser_receive 169
dsser_receive_term 170
dsser_transmit 167

Trigger Levels

Introduction

Two different trigger levels can be configured.

UART trigger level

The UART trigger level is hardware-dependent. After the specified number of bytes is received, the UART generates an interrupt and the bytes are copied into the receive buffer.

User trigger level

The user trigger level is hardware-independent and can be adjusted in smaller or larger steps than the UART trigger level. After a specified number of bytes is received in the receive buffer, the subinterrupt handler is called.

Related topics

Basics

HowTos

How to Handle Subinterrupts in Serial Communication

Introduction

The interrupt functions must be used only in handcoded applications. Using them in Simulink applications (user code or S-functions) conflicts with the internal interrupt handling.

The following subinterrupts can be passed to your application:

Subinterrupt	Meaning
`DSSER_TRIGGER_LEVEL_SUBINT`	Generated when the receive buffer is filled with the number of bytes specified as the trigger level (see Trigger Levels on page 150).
`DSSER_TX_FIFO_EMPTY_SUBINT`	Generated when the transmit buffer has no data.
`DSSER_RECEIVER_LINE_SUBINT`	Line status interrupt provided by the UART.
`DSSER_MODEM_STATE_SUBINT`	Modem status interrupt provided by the UART.
`DSSER_NO_SUBINT`	Generated after the last subinterrupt. This subinterrupt tells your application that no further subinterrupts were generated.

Method

To install a subinterrupt handler within your application

1 Write a function that handles your subinterrupt, such as:

```
void my_subint_handler(dsserChannel* serCh, Int32 subint)
{
   switch (subint)
   {
      case DSSER_TRIGGER_LEVEL_SUBINT:
         /* do something */
         break;
      case DSSER_TX_FIFO_EMPTY_SUBINT:
         /* do something */
         break;
      case DSSER_NO_SUBINT:
         /* no further subinterrupts */
         break;
      default:
      break;
   }
}
```

2 Initialize your subinterrupt handler:

```
dsser_subint_handler_inst(serCh,
      (dsser_subint_handler_t) my_subint_handler);
```

3 Enable the required subinterrupts:

```
dsser_subint_enable(serCh,
                     DSSER_TRIGGER_LEVEL_SUBINT_MASK |
                     DSSER_TX_FIFO_EMPTY_SUBINT_MASK);
```

Related topics

Basics

Trigger Levels........150

References

dsser_subint_enable........181
dsser_subint_handler_inst........180
dsser_subint_handler_t........158
dsserChannel........159

Example of a Serial Interface Communication

Example

The serial interface is initialized with 9600 baud, 8 data bits, 1 stop bit and no parity. The receiver FIFO generates a subinterrupt when it received 32 bytes and the subinterrupt handler `callback` is called. The subinterrupt handler `callback` reads the received bytes and sends the bytes back immediately.

```
#include <brtenv.h>
void callback(dsserChannel* serCh, UInt32 subint)
{
   UInt32 count;
   UInt8 data[32];
   switch (subint)
   {
      case DSSER_TRIGGER_LEVEL_SUBINT:
         msg_info_set(0,0,"DSSER_TRIGGER_LEVEL_SUBINT");
         dsser_receive(serCh,32,data,&count);
         dsser_transmit(serCh,count,data,&count);
         break;
      case DSSER_TX_FIFO_EMPTY_SUBINT:
         msg_info_set(0,0,"DSSER_TX_FIFO_EMPTY_SUBINT");
         break;
      default:
         break;
   }
}
main()
{
   dsserChannel* serCh;
   RTLIB_INIT();

/* allocate a new 1024 byte SW-FIFO */
   serCh = dsser_init(DSSER_ONBOARD, 0, 1024);
   dsser_subint_handler_inst(serCh,
         (dsser_subint_handler_t)callback);
   dsser_subint_enable(serCh,
         DSSER_TRIGGER_LEVEL_SUBINT_MASK |
         DSSER_TX_FIFO_EMPTY_SUBINT_MASK);
/* config and start the UART */
   dsser_config(serCh, DSSER_FIFO_MODE_OVERWRITE,
         9600, 8, DSSER_1_STOPBIT, DSSER_NO_PARITY,
         DSSER_14_BYTE_TRIGGER_LEVEL, 32, DSSER_RS232);
   RTLIB_INT_ENABLE();
   for(;;)
   {
      RTLIB_BACKGROUND_SERVICE();
   }
}
```

Data Types for Serial Communication

Introduction

There are some specific data structures specified for the serial communication interface.

Where to go from here

Information in this section

dsser_ISR

Syntax

```
typedef union
{
   UInt32    Byte;
   struct
   {
      unsigned dummy : 24;
      unsigned DSSER_FIFO_STATUS_BIT1 : 1;
      unsigned DSSER_FIFO_STATUS_BIT0 : 1;
      unsigned DSSER_BIT5 : 1;
      unsigned DSSER_BIT4 : 1;
      unsigned DSSER_INT_PRIORITY_BIT2 : 1;
      unsigned DSSER_INT_PRIORITY_BIT1 : 1;
      unsigned DSSER_INT_PRIORITY_BIT0 : 1;
      unsigned DSSER_INT_STATUS : 1;
   }Bit;
}dsser_ISR;
```

Include file

`dsserdef.h`

Description

The structure `dsser_ISR` provides information about the interrupt identification register (IIR). Call **`dsser_status_read`** to read the status register.

Note

The data type contains the value of the UART's register.
The register conforms to a standard 16550 UART such as the TEXAS INSTRUMENTS PC16550D. For further information, refer to http://www.ti.com.

Members

The structure provides the following members:

Member	Description
`DSSER_INT_STATUS`	0 if interrupt pending
`DSSER_INT_PRIORITY_BIT0`	Interrupt ID bit 1
`DSSER_INT_PRIORITY_BIT1`	Interrupt ID bit 2
`DSSER_INT_PRIORITY_BIT2`	Interrupt ID bit 3
`DSSER_BIT4`	Not relevant
`DSSER_BIT5`	Not relevant
`DSSER_FIFO_STATUS_BIT0`	UART FIFOs enabled
`DSSER_FIFO_STATUS_BIT1`	UART FIFOs enabled

For more information about the predefined constants, refer to the datasheet of the *TEXAS INSTRUMENTS, PC16550D*.

Related topics

References

dsser_status_read..177

dsser_LSR

Syntax

```
typedef union
{
   UInt32    Byte;
   struct
   {
      unsigned dummy : 24;
      unsigned DSSER_FIFO_DATA_ERR : 1;
      unsigned DSSER_THR_TSR_STATUS : 1;
      unsigned DSSER_THR_STATUS : 1;
      unsigned DSSER_BREAK_STATUS : 1;
      unsigned DSSER_FRAMING_ERR : 1;
      unsigned DSSER_PARITY_ERR : 1;
      unsigned DSSER_OVERRUN_ERR : 1;
      unsigned DSSER_RECEIVE_DATA_RDY : 1;
   }Bit;
} dsser_LSR;
```

Include file

`dsserdef.h`

Description

The structure `dsser_LSR` provides information about the status of data transfers. Call **`dsser_status_read`** to read the status register.

Note

The data type contains the value of the UART's register.
The register conforms to a standard 16550 UART such as the TEXAS INSTRUMENTS PC16550D. For further information, refer to http://www.ti.com.

Members

The structure provides the following members.

Member	Description
DSSER_RECEIVE_DATA_RDY	Data ready (DR) indicator
DSSER_OVERRUN_ERR	Overrun error (OE) indicator
DSSER_PARITY ERR	Parity error (PE) indicator
DSSER_FRAMING_ERR	Framing error (FE) indicator
DSSER_BREAK_STATUS	Break interrupt (BI) indicator
DSSER_THR_STATUS	Transmitter holding register empty (THRE)
DSSER_THR_TSR_STATUS	Transmitter empty (TEMT) indicator
DSSER_FIFO_DATA_ERR	Error in receiver FIFO

For more information about the predefined constants, refer to the datasheet of the *TEXAS INSTRUMENTS, PC16550D*.

Related topics

References

dsser_MSR

Syntax

```
typedef union
{
   UInt32    Byte;
   struct
   {
      unsigned dummy : 24;
      unsigned DSSER_OP2_STATUS : 1;
      unsigned DSSER_OP1_STATUS : 1;
      unsigned DSSER_DTR_STATUS : 1;
      unsigned DSSER_RTS_STATUS : 1;
      unsigned DSSER_CD_STATUS : 1;
      unsigned DSSER_RI_STATUS : 1;
      unsigned DSSER_DSR_STATUS : 1;
      unsigned DSSER_CTS_STATUS : 1;
   }Bit;
}dsser_MSR;
```

Include file

`dsserdef.h`

Description

The structure `dsser_MSR` provides information about the state of the control lines. Call **`dsser_status_read`** to read the status register.

Note

The data type contains the value of the UART's register.
The register conforms to a standard 16550 UART such as the TEXAS INSTRUMENTS PC16550D. For further information, refer to http://www.ti.com.

Members

The structure provides the following members.

Member	Description
DSSER_CTS_STATUS	Clear-to-send (CTS) changed state
DSSER_DSR_STATUS	Data-set-ready (DSR) changed state
DSSER_RI_STATUS	Ring-indicator (RI) changed state
DSSER_CD_STATUS	Data-carrier-detect (CD) changed state
DSSER_RTS_STATUS	Complement of CTS
DSSER_DTR_STATUS	Complement of DSR
DSSER_OP1_STATUS	Complement of RI
DSSER_OP2_STATUS	Complement of DCD

For more information about the predefined constants, refer to the datasheet of the *TEXAS INSTRUMENTS, PC16550D*.

Related topics

References

dsser_subint_handler_t

Syntax

```
typedef void (*dsser_subint_handler_t) (void* serCh, Int32 subint)
```

Include file

`dsserdef.h`

Description

You must use this type definition if you install a subinterrupt handler (see How to Handle Subinterrupts in Serial Communication on page 151 or dsser_subint_handler_inst on page 180).

Members

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

subint Identification number of the related subinterrupt. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_TRIGGER_LEVEL_SUBINT	Interrupt triggered when the user trigger level is reached (see Trigger Levels on page 150).

Predefined Symbol	Meaning
DSSER_TX_FIFO_EMPTY_SUBINT	Interrupt triggered when the transmit buffer is empty.
DSSER_RECEIVER_LINE_SUBINT	Line status interrupt of the UART.
DSSER_MODEM_STATE_SUBINT	Modem status interrupt of the UART.
DSSER_NO_SUBINT	Flag that is sent after the last triggered subinterrupt.

Related topics

Basics

Trigger Levels........ 150

References

dsser_init........ 162

dsserChannel

Syntax

```
typedef struct
{
/*--- public ----------------------------*/
   /* interrupt status register */
   dsser_ISR intStatusReg;
   /* line status register */
   dsser_LSR lineStatusReg;
   /* modem status register */
   dsser_MSR modemStatusReg;
/*--- protected ----------------------------*/
   /*--- serial channel allocation ---*/
   UInt32 module;
   UInt32 channel;
   Int32 board_bt;
   UInt32 board;
   UInt32 fifo_size;
   UInt32 frequency;
```

```
    /*--- serial channel configuration ---*/
    UInt32 baudrate;
    UInt32 databits;
    UInt32 stopbits;
    UInt32 parity;
    UInt32 rs_mode;
    UInt32 fifo_mode;
    UInt32 uart_trigger_level;
    UInt32 user_trigger_level;
    dsser_subint_handler_t subint_handler;
    dsserService* serService;
    dsfifo_t* txFifo;
    dsfifo_t* rxFifo;
    UInt32 queue;
    UInt8 isr;
    UInt8 lsr;
    UInt8 msr;
    UInt32 interrupt_mode;
    UInt8 subint_mask;
    Int8 subint;
}dsserChannel
```

Include file

`dsserdef.h`

Description

This structure provides information about the serial channel. You can call `dsser_status_read` to read the values of the status registers. All protected variables are only for internal use.

Members

intStatusReg Interrupt status register. Refer to `dsser_ISR` on page 154.

lineStatusReg Line status register. Refer to `dsser_LSR` on page 156.

modemStatusReg Modem status register. Refer to `dsser_MSR` on page 157.

Related topics

References

dsser_status_read 177

Generic Serial Interface Communication Functions

Where to go from here

Information in this section

dsser_init

Syntax

```
dsserChannel* dsser_init(
      UInt32 base,
      UInt32 channel,
      UInt32 fifo_size)
```

Include file

`dsser.h`

Purpose

To initialize the serial interface and install the interrupt handler.

Note

Pay attention to the initialization sequence. First, initialize the processor board, then the I/O boards, and then the serial interface.

Parameters

base Specifies the base address of the serial interface. This value has to be set to DSSER_ONBOARD.

channel Specifies the number of the channel to be used for the serial interface. The permitted value is 0.

fifo_size Specifies the size of the transmit and receive buffer in bytes. The size must be a power of two (2^n) and at least 64 bytes. The maximum size depends on the available memory.

Return value

This function returns the pointer to the serial channel structure.

Messages

The following messages are defined (x = base address of the I/O board, y = number of the channel):

ID	Type	Message	Description
100	Error	x, ch=y, Board not found!	I/O board was not found.
101	Warning	x, ch=y, Mixed usage of high and low level API!	It is not allowed to use the generic functions (high-level access functions) and the low-level access functions of the serial interface on the same channel. It is recommended to use only the generic functions.
501	Error	x, ch=y, memory: Allocation error on master.	Memory allocation error. No free memory on the master.
508	Error	x, ch=y, channel: out of range!	The `channel` parameter is out of range.
700	Error	x, ch=y, Buffersize: Illegal	The `fifo_size` parameter is out of range.

Related topics

Basics

Basic Principles of Serial Communication ... 149

Examples

Example of a Serial Interface Communication ... 152

References

Data Types for Serial Communication ... 153
dsser_config ... 164
dsser_free ... 163

dsser_free

Syntax

```
Int32 dsser_free(dsserChannel*serCh)
```

Include file

`dsser.h`

Purpose

To close a serial interface.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation. The specified serial interface is closed. Its memory for the buffer is freed and the interrupts are released. A serial interface can be created again using the `dsser_init` function.
DSSER_TX_FIFO_NOT_EMPTY	The serial interface is not closed, because the transmit buffer is not empty.
DSSER_CHANNEL_INIT_ERROR	There is no serial interface to be closed (serCh == NULL).

Related topics

Basics

Basic Principles of Serial Communication.......... 149

References

dsser_init.......... 162

dsser_config

Syntax

```
void dsser_config(
      dsserChannel* serCh,
      const UInt32 fifo_mode,
      const UInt32 baudrate,
      const UInt32 databits,
      const UInt32 stopbits,
      const UInt32 parity,
      const UInt32 uart_trigger_level,
      const Int32  user_trigger_level,
      const UInt32 uart_mode)
```

Include file

`dsser.h`

Purpose

To configure and start the serial interface.

Note

- This function starts the serial interface. Therefore, all dSPACE real-time boards must be initialized and the interrupt vector must be installed before calling this function.
- Calling this function again reconfigures the serial interface.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

fifo_mode Specifies the mode of the receive buffer (see Software FIFO Buffer on page 149):

Value	Mode	Meaning
DSSER_FIFO_MODE_BLOCKED	Blocked mode	If the receive buffer is full, new data is rejected.
DSSER_FIFO_MODE_OVERWRITE	Overwrite mode	If the receive buffer is full, new data replaces the oldest data in the buffer.

baudrate Specifies the baud rate in bits per second:

Mode	Baud Rate Range
RS232	5 … 230,400 baud

For further information, refer to Serial Interface of the DS1007 (DS1007 Features 🕮).

databits Specifies the number of data bits. Values are: 5, 6, 7, 8.

stopbits Specifies the number of stop bits. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_1_STOPBIT	1 stop bit
DSSER_2_STOPBIT	The number of stop bits depends on the number of the specified data bits: 5 data bits: 1.5 stop bits 6 data bits: 2 stop bits 7 data bits: 2 stop bits 8 data bits: 2 stop bits

parity Specifies whether and how parity bits are generated. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_PARITY	No parity bits
DSSER_ODD_PARITY	Parity bit is set so that there is an odd number of "1" bits in the byte, including the parity bit.
DSSER_EVEN_PARITY	Parity bit is set so that there is an even number of "1" bits in the byte, including the parity bit.
DSSER_FORCED_PARITY_ONE	Parity bit is forced to a logic 1.
DSSER_FORCED_PARITY_ZERO	Parity bit is forced to a logic 0.

uart_trigger_level Sets the UART trigger level (see Trigger Levels on page 150). The following symbols are predefined:

Predefined Symbol	Meaning
`DSSER_1_BYTE_TRIGGER_LEVEL`	1-byte trigger level
`DSSER_4_BYTE_TRIGGER_LEVEL`	4-byte trigger level
`DSSER_8_BYTE_TRIGGER_LEVEL`	8-byte trigger level
`DSSER_14_BYTE_TRIGGER_LEVEL`	14-byte trigger level

Note

Use the highest UART trigger level possible to generate fewer interrupts.

user_trigger_level Sets the user trigger level within the range of 1 … (`fifo_size` - 1) for the receive interrupt (see Trigger Levels on page 150):

Value	Meaning
`DSSER_DEFAULT_TRIGGER_LEVEL`	Synchronizes the UART trigger level and the user trigger level.
1 … (`fifo_size` - 1)	Sets the user trigger level.
`DSSER_TRIGGER_LEVEL_DISABLE`	No receive subinterrupt handling for the serial interface

uart_mode Sets the mode of the UART transceiver.

The following symbols are predefined:

Predefined Symbol	Meaning
`DSSER_RS232`	RS232 mode
`DSSER_AUTOFLOW_DISABLE`	Transfer without HW handshake (RTS/CTS)
`DSSER_AUTOFLOW_ENABLE`	Transfer with HW handshake (RTS/CTS)

Messages

The following messages are defined (x = base address of the I/O board, y = number of the channel):

ID	Type	Message	Description
101	Warning	x, ch=y, Mixed usage of high and low level API!	It is not allowed to use the generic functions (high-level access functions) and the low-level access functions of the serial interface on the same channel. It is recommended to use only the generic functions.
601	Error	x, serCh: The UART channel was not initialized.	The `dsser_config` function was called before the serial interface was initialized with `dsser_init`.
602	Error	x, ch=y, baudrate: Illegal!	The `baudrate` parameter is out of range.
603	Error	x, ch=y, databits: Use range 5 … 8 bits!	The `databits` parameter is out of range.

ID	Type	Message	Description
604	Error	x, ch=y, stopbits: Illegal number (1-2 bits allowed)!	The `stopbits` parameter is out of range.
605	Error	x, ch=y, parity: Illegal parity!	The `parity` parameter is out of range.
606	Error	x, ch=y, trigger_level: Illegal UART trigger level!	The `uart_trigger_level` parameter is out of range.
607	Error	x, ch=y, trigger_level: Illegal user trigger level!	The `user_trigger_level` parameter is out of range.
608	Error	x, ch=y, fifo_mode: Use range 0 … (fifo_size-1) bytes!	The `uart_mode` parameter is out of range.
609	Error	x, ch=y, uart_mode: Transceiver not supported!	The selected UART mode does not exist for this serial interface.
611	Error	x, ch=y, uart_mode: Autoflow is not supported!	Autoflow does not exist for this serial interface.

Related topics

Basics

Basic Principles of Serial Communication..149

Examples

Example of a Serial Interface Communication..152

References

dsser_init..162

dsser_transmit

Syntax

```
Int32 dsser_transmit(
      dsserChannel* serCh,
      UInt32 datalen,
      UInt8* data,
      UInt32* count)
```

Include file

`dsser.h`

Purpose

To transmit data through the serial interface.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

datalen Specifies the number of bytes to be transmitted.

data Specifies the pointer to the data to be transmitted.

count Specifies the pointer to the number of transmitted bytes. When this function is finished, the variable contains the number of bytes that were transmitted. If the function was able to send all the data, the value is equal to the value of the `datalen` parameter.

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_FIFO_OVERFLOW	The FIFO is filled or not all the data could be copied to the FIFO.
DSSER_COMMUNICATION_FAILED	The function failed with no effect on the input or output data. No data is written to the FIFO. The communication between the real-time processor and the UART is might be overloaded. Do not poll this function because it may cause an endless loop.

Example

This example shows how to check the transmit buffer for sufficient free memory before transmitting data.

```
UInt32 count;
UInt8 block[5] = {1, 2, 3, 4, 5};
if(dsser_transmit_fifo_level(serCh) < serCh->fifo_size - 5)
{
   dsser_transmit(serCh, 5, block, &count);
}
```

Related topics

Basics

Examples

References

dsser_receive

Syntax

```
Int32 dsser_receive(
      dsserChannel* serCh,
      UInt32 datalen,
      UInt8* data,
      UInt32* count)
```

Include file

`dsser.h`

Purpose

To receive data through the serial interface.

> **Tip**
>
> It is better to receive a block of bytes instead of several single bytes because the processing speed is faster.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

datalen Specifies the number of data bytes to be read. The value must not be greater than the FIFO size defined with `dsser_init`.

data Specifies the pointer to the destination buffer.

count Specifies the pointer to the number of received bytes. When this function is finished, the variable contains the number of bytes that were received.

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
`DSSER_NO_ERROR`	No error occurred during the operation.
`DSSER_NO_DATA`	No new data is read from the FIFO.
`DSSER_FIFO_OVERFLOW`	The FIFO is filled. The behavior depends on the `fifo_mode` adjusted with `dsser_config`: ▪ `fifo_mode = DSSER_FIFO_MODE_BLOCKED` Not all new data could be placed in the FIFO. ▪ `fifo_mode = DSSER_FIFO_MODE_OVERWRITE` The old data is rejected.
`DSSER_COMMUNICATION_FAILED`	The function failed with no effect on the input or output data. No data is read from the FIFO. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Example

The following example shows how to receive 4 bytes.

```
UInt8 data[4];
UInt32 count;
Int32 error;
/* receive four bytes over serCh */
error = dsser_receive(serCh, 4, data, &count);
```

Related topics

Basics

Examples

References

dsser_receive_term

Syntax

```
Int32 dsser_receive_term(
      dsserChannel* serCh,
      UInt32 datalen,
      UInt8* data,
      UInt32* count,
      const UInt8 term)
```

Include file

`dsser.h`

Purpose

To receive data through the serial interface.

Description

This function is terminated when the character `term` is received. The character `term` is stored as the last character in the buffer, so you can check if the function was completed.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

datalen Specifies the number of data bytes to be read. The value must not be greater than the FIFO size defined with `dsser_init`.

data Specifies the pointer to the destination buffer.

count Specifies the pointer to the number of received bytes. When this function is finished, the variable contains the number of bytes that were received.

term Specifies the character that terminates the reception of bytes.

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_NO_DATA	No new data is read from the FIFO.
DSSER_FIFO_OVERFLOW	The FIFO is filled. The behavior depends on the `fifo_mode` adjusted with `dsser_config`: ▪ `fifo_mode = DSSER_FIFO_MODE_BLOCKED` Not all new data could be placed in the FIFO. ▪ `fifo_mode = DSSER_FIFO_MODE_OVERWRITE` The old data is rejected.
DSSER_COMMUNICATION_FAILED	The function failed with no effect on the input or output data. No data is read from the FIFO. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Example

The following example shows how to receive a maximum of 4 bytes via the serial channel until the terminating character '\r' occurs:

```
UInt8 data[4];
UInt32 count;
Int32 error;
error = dsser_receive_term(serCh, 4, data, &count, '\r');
```

Related topics

Basics

Basic Principles of Serial Communication..149

References

dsser_init..162

dsser_fifo_reset

Syntax

```
Int32 dsser_fifo_reset(dsserChannel* serCh)
```

Include file

`dsser.h`

Purpose

To reset the serial interface.

Description

The channel is disabled and the transmit and receive buffers are cleared.

> **Note**
>
> If you want to continue to use the serial interface, the channel has to be enabled with `dsser_enable`.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
`DSSER_NO_ERROR`	No error occurred during the operation.
`DSSER_COMMUNICATION_FAILED`	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Related topics

Basics

Basic Principles of Serial Communication........ 149

References

dsser_enable........ 173
dsser_init........ 162

dsser_enable

Syntax

```
Int32 dsser_enable(const dsserChannel* serCh)
```

Include file

`dsser.h`

Purpose

To enable the serial interface.

Description

The UART interrupt is enabled, the serial interface starts transmitting and receiving data.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_COMMUNICATION_FAILED	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Related topics

Basics

References

dsser_disable

Syntax

```
Int32 dsser_disable(const dsserChannel* serCh)
```

Include file `dsser.h`

Purpose To disable the serial interface.

Description The serial interface stops transmitting data, incoming data is no longer stored in the receive buffer and the UART subinterrupts are disabled.

Parameters **serCh** Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_COMMUNICATION_FAILED	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Related topics

Basics

References

dsser_error_read

Syntax

```
Int32 dsser_error_read(const dsserChannel* serCh)
```

Include file `dsser.h`

Purpose To read an error flag of the serial interface.

Description

Because only one error flag is returned, you have to call this function as long as the value `DSSER_NO_ERROR` is returned to get all error flags.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value

This function returns an error flag.

The following symbols are predefined:

Predefined Symbol	Meaning
`DSSER_NO_ERROR`	No error flag set
`DSSER_FIFO_OVERFLOW`	Too many bytes for the buffer

Related topics

Basics

Basic Principles of Serial Communication......149

References

dsser_config......164
dsser_init......162

dsser_transmit_fifo_level

Syntax

```
Int32 dsser_transmit_fifo_level(const dsserChannel* serCh)
```

Include file

`dsser.h`

Purpose

To get the number of bytes in the transmit buffer.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value

This function returns the number of bytes in the transmit buffer.

Related topics

Basics

References

dsser_receive_fifo_level

Syntax

```
Int32 dsser_receive_fifo_level(const dsserChannel* serCh)
```

Include file

`dsser.h`

Purpose

To get the number of bytes in the receive buffer.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

Return value

This function returns the number of bytes in the receive buffer.

Related topics

Basics

References

dsser_status_read

Syntax

```
Int32 dsser_status_read(
      dsserChannel*serCh,
      const UInt8 register_type)
```

Include file

`dsser.h`

Purpose

To read the value of one or more status registers and to store the values in the appropriate fields of the channel structure.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

register_type Specifies the register that is read. You can combine the predefined symbols with the logical operator OR to read several registers. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_STATUS_IIR_FCR	Interrupt status register, see `dsser_ISR` data type.
DSSER_STATUS_LSR	Line status register, see `dsser_ISR` data type.
DSSER_STATUS_MSR	Modem status register, see `dsser_ISR` data type.

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_COMMUNICATION_FAILED	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Example

This example shows how to check if the clear-to-send bit has changed:

```
UInt8 cts;
dsser_status_read(serCh, DSSER_STATUS_MSR);
cts = serCh->modemStatusReg.Bit.DSSER_CTS_STATUS;
```

Related topics

Basics

References

dsser_handle_get

Syntax

```
dsserChannel* dsser_handle_get(
    UInt32 base,
    UInt32 channel)
```

Include file

`dsser.h`

Purpose

To check whether the serial interface is in use.

Parameters

base Specifies the base address of the serial interface. This value has to be set to DSSER_ONBOARD.

channel Specifies the number of the channel to be used for the serial interface. The permitted value is 0.

Return value

This function returns:

- NULL if the specified serial interface is not used.
- A pointer to the serial channel structure of the serial interface that has been created by using the `dsser_init` function.

Related topics

Basics

References

dsser_set

Syntax

```
Int32 dsser_set(
      dsserChannel *serCh,
      UInt32 type,
      const void *value_p)
```

Include file

`dsser.h`

Purpose

To set a property of the UART.

Description

The DS1007 board is delivered with a standard quartz working with the frequency of $1.8432 \cdot 10^6$ Hz. You can replace this quartz with another one with a different frequency. Then you have to set the new quartz frequency using `dsser_set` followed by executing `dsser_config`.

> **Note**
>
> You must execute `dsser_config` after `dsser_set`; otherwise `dsser_set` has no effect.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

type Specifies the property to be changed (`DSSER_SET_UART_FREQUENCY`).

value_p Specifies the pointer to a UInt32-variable with the new value, for example, a variable which contains the quartz frequency.

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
`DSSER_NO_ERROR`	No error occurred during the operation.
`DSSER_COMMUNICATION_FAILED`	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Example

This example sets a new value for the frequency.

```
UInt32 freq = 1843200;            /* 1.8432 MHz */
Int32 error;
error = dsser_set(serCh, DSSER_SET_UART_FREQUENCY, &freq);
```

Related topics

Basics

References

dsser_subint_handler_inst

Syntax

```
dsser_subint_handler_t dsser_subint_handler_inst(
        dsserChannel* serCh,
        dsser_subint_handler_t subint_handler)
```

Include file

`dsser.h`

Purpose

To install a subinterrupt handler for the serial interface.

Description

After installing the handler, the specified subinterrupt type must be enabled (see dsser_subint_enable on page 181).

Note

The interrupt functions must be used only in handcoded applications. Using them in Simulink applications (user code or S-functions) conflicts with the internal interrupt handling.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

subint_handler Specifies the pointer to the subinterrupt handler.

Return value

This function returns the pointer to the previously installed subinterrupt handler.

Related topics

Basics

Examples

References

dsser_subint_enable

Syntax

```
Int32 dsser_subint_enable(
      dsserChannel* serCh,
      const UInt8 subint)
```

Include file

`dsser.h`

Purpose

To enable one or several subinterrupts of the serial interface.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

subint Specifies the subinterrupts to be enabled. You can combine the predefined symbols with the logical operator OR to enable several subinterrupts. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_TRIGGER_LEVEL_SUBINT_MASK	Interrupt triggered when the user trigger level is reached (see Trigger Levels on page 150)
DSSER_TX_FIFO_EMPTY_SUBINT_MASK	Interrupt triggered when the transmit buffer is empty

Predefined Symbol	Meaning
DSSER_RECEIVER_LINE_SUBINT_MASK	Line status interrupt of the UART
DSSER_MODEM_STATE_SUBINT_MASK	Modem status interrupt of the UART

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_COMMUNICATION_FAILED	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Related topics

Basics

Examples

References

dsser_subint_disable

Syntax

```
Int32 dsser_subint_disable(
      dsserChannel* serCh,
      const UInt8 subint)
```

Include file

`dsser.h`

Purpose

To disable one or several subinterrupts of the serial interface.

Parameters

serCh Specifies the pointer to the serial channel structure (see dsser_init on page 162).

subint Specifies the subinterrupts to be disabled. You can combine the predefined symbols with the logical operator OR to disable several subinterrupts. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_TRIGGER_LEVEL_SUBINT_MASK	Interrupt triggered when the user trigger level is reached (see Trigger Levels on page 150)
DSSER_TX_FIFO_EMPTY_SUBINT_MASK	Interrupt triggered when the transmit buffer is empty
DSSER_RECEIVER_LINE_SUBINT_MASK	Line status interrupt of the UART
DSSER_MODEM_STATE_SUBINT_MASK	Modem status interrupt of the UART

Return value

This function returns an error code. The following symbols are predefined:

Predefined Symbol	Meaning
DSSER_NO_ERROR	No error occurred during the operation.
DSSER_COMMUNICATION_FAILED	The function failed. The communication between the real-time processor and the UART might be overloaded. Do not poll this function because it might cause an endless loop.

Related topics

Basics

References

dsser_word2bytes

Syntax

```
UInt8* dsser_word2bytes(
      const UInt32* word,
      UInt8* bytes,
      const int bytesInWord)
```

Include file

dsser.h

Purpose

To convert a word (max. 4 bytes long) into a byte array.

Parameters

word Specifies the pointer to the input word.

bytes Specifies the pointer to the byte array. The byte array must have enough memory for `bytesInWord` elements.

bytesInWord Specifies the number of elements in the byte array. Possible values are 2, 3, 4.

Return value

This function returns the pointer to a byte array.

Example

The following example shows how to write a processor-independent function that transmits a 32-bit value:

```
void word_transmit(dsserChannel* serCh, UInt32* word, UInt32* count)
{
   UInt8    bytes[4];
   UInt8*   data_p;
   if(dsser_transmit_fifo_level(serCh) < serCh->fifo_size - 4)
   {
      data_p = dsser_word2bytes(word, bytes, 4);
      dsser_transmit(serCh, 4, data_p, count);
   }
   else
   {
      *count = 0;
   }
}
```

Use of the function:

```
UInt32 word = 0x12345678;
UInt32 count;
word_transmit(serCh, &word, &count);
```

Related topics

Basics

References

dsser_bytes2word

Syntax

```
UInt32* dsser_bytes2word(
      UInt8* bytes_p,
      UInt32* word_p,
      const int bytesInWord)
```

Include file

`dsser.h`

Purpose

To convert a byte array with a maximum of 4 elements into a single word.

Parameters

bytes_p Specifies the pointer to the input byte array.

word_p Specifies the pointer to the converted word.

bytesInWord Specifies the number of elements in the byte array. Possible values are 2, 3, 4.

Return value

This function returns the pointer to the converted word.

Example

The following example shows how to write a processor-independent function that receives a 32-bit value:

```
void word_receive(dsserChannel* serCh, UInt32* word_p, UInt32* count)
{
   UInt8 bytes[4];
   if(dsser_receive_fifo_level(serCh) > 3)
   {
      dsser_receive(serCh, 4, bytes, count);
      word_p = dsser_bytes2word(bytes, word_p, 4);
   }
   else
   {
      *count = 0;
   }
}
```

Use of the function:

```
UInt32 word;
UInt32 count;
word_receive(serCh, &word, &count);
```

Related topics

Basics

Basic Principles of Serial Communication 149

References

dsser_receive 169
dsser_receive_fifo_level 176
dsser_word2bytes 183

USB Flight Recorder

Purpose

With the USB Flight Recorder, you can perform long-term data acquisition. During the simulation, the values of selectable variables are written to the connected USB mass storage device. The available storage size is only restricted by the USB mass storage device.

Introduction to the USB Flight Recorder

Purpose

With the USB Flight Recorder, you can perform long-term data acquisition. During the simulation, the values of selectable variables are written to the connected USB mass storage device.

Description

Any standard USB mass storage device can be used, such as a USB memory stick or an external USB hard drive with or without separate power supply. The USB device must be formatted with the Microsoft FAT32 file system and must be directly connected to DS1007 PPC Processor Board. Connection via USB hubs is not supported.

The recorded data is written to a sequence of files stored in the root directory of the USB device. The file names are generated automatically and contain the name of the real-time model as well as the creation date and time of the file. Only one file is written at a time. The file grows until it reaches a user-defined maximum size (refer to **`dsflrec_usb_initialize`** (USB Flight Recorder RTLib Reference 🕮)). After that, the file is closed and a new output file is created. As long as the USB Flight Recording session runs, new files are generated until the maximum number of files has been reached. The maximum file number is given by the size of the USB device divided by the maximum file size.

When the maximum number of files is reached, either the oldest file is deleted or the USB Flight Recording session is stopped (refer to **`dsflrec_usb_initialize`** (USB Flight Recorder RTLib Reference 🕮)).

On multicore platforms such as the DS1007 PPC Processor Board, the USB Flight Recorder is separately configured for each real-time application running on the board. Each instantiated USB Flight Recorder generates its own output files.

The real-time model continues to run, even if the USB Flight Recording session is stopped.

> **Note**
>
> To avoid data loss, use the **`dsflrec_usb_eject`** function before unplugging the USB memory stick or hard drive.

Characteristics

For information on the USB Flight Recorder's characteristics, such as the maximum data rate or the maximum number of variables, refer to USB Flight Recorder (DS1007 Features 📖). The section also contains instructions on how to use the USB Flight Recorder.

Nonvolatile Data Handling (NVDATA)

Purpose

With the nonvolatile data handling (NVDATA), you can write data to the board's nonvolatile memory and read data from the memory.

Where to go from here

Information in this section

Information in other sections

Nonvolatile Data Handling (NVDATA) (DS1007 Features 📖)
The DS1007 provides access to the board's nonvolatile memory by implementing the access in a real-time application or by using the board's web interface.

Using the Web Interface for Nonvolatile Data Handling (DS1007 Features 📖)
The web interface of your DS1007 provides a configuration page which lets you manage the data sets stored in the board's nonvolatile memory.

NvData_apply

Syntax

```
Int32 NvData_apply(
   NvDataTDrv *pNvDataDrv)
```

Include file

`NvDataRP.h`

Purpose

To apply the initialization settings to the NvData driver object.

Description

Before you can use the nonvolatile data handling, you must apply the settings to the NvData driver object. The `NvData_apply` function is intended to be called at the end of the initialization phase of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller and a relevant error message is output.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using **`NvData_create`**.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_DRIVER_NULL	A NULL driver object was passed to the function.
NVDATA_ERR_DUPLICATE_APPLY	The function was called more than once.
NVDATA_ERR_DUPLICATE_ENTRIES	Data sets with the same name but different settings were detected.
NVDATA_ERR_INTERNAL	An internal error was detected. Check the message log.

Related topics

References

NvData_create 191

NvData_create

Syntax

```
Int32 NvData_create(
   NvDataTDrv **ppNvDataDrv)
```

Include file

`NvDataRP.h`

Purpose

To create the driver object for nonvolatile data handling.

Description

This function performs all the steps necessary to create a driver object for nonvolatile data handling. You can create only one driver object per application.

The function is intended to be called during the initialization phase of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller and a relevant error message is output.

Parameters

ppNvDataDrv Lets you specify the address of a variable which holds the address of the created driver object.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_MAX_INSTANCE	The function was called more than once.

NvData_createDataSet

Syntax

```
Int32 NvData_createDataSet(
   NvDataTDrv *pNvDataDrv,
   UInt32 *phDataSet)
```

Include file

`NvDataRP.h`

Purpose

To create a data set for nonvolatile data handling.

Description

This function creates a handle for a data set. You have to completely configure the instantiated data set by specifying its name, the number of elements to be transferred, and the type of the elements.

You can create up to 64 data sets in your real-time application. In a multicore application, the sum of the data sets allocated by the subapplications also must not exceed 64.

The available memory for data sets is limited to a total of 64 KB.

Note

The board's nonvolatile memory is not automatically cleared upon application start. Data sets from previous sessions might still exist in the memory and decrease the available memory size.

- Use the board's web interface to check and manage the content of the nonvolatile memory.

The function is intended to be called during the initialization phase of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller and a relevant error message is output.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using `NvData_create`.

phDataSet Lets you specify the address to the created data set handler.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_OUT_OF_MEMORY	The data set cannot be created because either the number of data sets is exceeded or there is not enough free nonvolatile memory available.
NVDATA_ERR_DRIVER_NULL	The function was called with a NULL driver object.

Related topics

References

NvData_create..191

NvData_read

Syntax

```
Int32 NvData_read(
   NvDataTDrv *pNvDataDrv,
   UInt32 hDataSet,
   void *pDestBuffer,
   int *DataValid)
```

Include file

`NvDataRP.h`

Purpose

To read a data set from the board's nonvolatile memory.

Description

You can access a data set via its handler that you created by using the **`NvData_createDataSet`** function. To read the data from the specified data set, you have to provide a destination buffer with at least the memory size of the specified data set. The required memory size depends on the number of elements in the data set specified by using the **`NvData_setDimension`** function and the data types of the contained elements specified by using the **`NvData_setType`** function.

The `DataValid` parameter can be used to check, if the data set has previously been written. If the data set has been successfully written at least once, `DataValid` will return 1. Otherwise it will return 0.

The function is intended to be called during the run time of the real-time application. If an error is detected, the very first instance of the error is returned to the caller.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using **`NvData_create`**.

hDataSet Lets you specify the data set to be accessed. The data set must be created before by using **`NvData_createDataSet`**.

pDestBuffer Lets you specify the address of the buffer in which the read data is being stored.
The size of the buffer must match the size of the data set to be read as the result of the number of elements in the data set and their data types.

DataValid Returns a flag that shows if the specified data set is valid or not.
There must be one initial write access to the data set before a read access will be valid.

- 0: Data set has not been previously written, i.e., it is still uninitialized.
- 1: Data set has been written at least once.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_DRIVER_NULL	A NULL driver object was passed to the function.
NVDATA_ERR_INVALID_HANDLE	The `hDataSet` handle is invalid.

Related topics

References

NvData_setDimension

Syntax

```
Int32 NvData_setDimension(
   NvDataTDrv *pNvDataDrv,
   UInt32 hDataSet,
   UInt32 Dimension)
```

Include file

`NvDataRP.h`

Purpose

To specify the number of elements in the data set.

Description

The memory size of a data set results from its specified dimension (number of elements) and the specified data type of the elements (see **`NvData_setType`**).

The function is intended to be called during the initialization phase of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using **`NvData_create`**.

hDataSet Lets you specify the data set to be accessed. The data set must be created before by using **`NvData_createDataSet`**.

Dimension Lets you specify the number of elements in the current data set in the range 1 … 64.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_DRIVER_NULL	A NULL driver object was passed to the function.
NVDATA_ERR_INVALID_SIZE	The specified dimension is 0.
NVDATA_ERR_TOO_MANY_ELEMENTS	The specified dimension is greater than 64.

Related topics

References

NvData_create 191
NvData_createDataSet 191
NvData_setType 197

NvData_setName

Syntax

```
Int32 NvData_setName(
   NvDataTDrv *pNvDataDrv,
   UInt32 hDataSet,
   const char *Name)
```

Include file

`NvDataRP.h`

Purpose

To specify the name of the data set.

Description

Each data set in the nonvolatile memory is uniquely identified by its name. For each data set handle that you created by using **`NvData_createDataSet`**, its name, type and dimension must be specified.

For further information on handling the NVDATA via the board's web interface, refer to Nonvolatile Data Handling (NVDATA) (DS1007 Features 📖).

The function is intended to be called during the initialization phase of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using `NvData_create`.

hDataSet Lets you specify the data set to be accessed. The data set must be created before by using `NvData_createDataSet`.

Name Lets you specify a unique name for the data set with a maximum length of 63 characters.

Note

A valid data set name is a character string of letters, digits, and underscores. There are the following naming restrictions for the data set name:

- The first character must be a letter.
- The name must not be a keyword, such as `while` or `if`.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_DRIVER_NULL	A NULL driver object was passed to the function.
NVDATA_ERR_INVALID_NAME	The specified name is invalid: ▪ The name is not yet specified or has a length of 0. ▪ The maximum size of 63 characters is exceeded. ▪ The specified name already exists.

Related topics

References

NvData_create 191
NvData_createDataSet 191

NvData_setType

Syntax

```
Int32 NvData_setType(
   NvDataTDrv *pNvDataDrv,
   UInt32 hDataSet,
   NvDataEType DataType)
```

Include file

`NvDataRP.h`

Purpose

To specify the data type of the elements in a data set.

Description

The memory size of a data set results from its specified dimension (number of elements) set by **`NvData_setDimension`** and the specified data type of the elements.

The function is intended to be called during the initialization phase of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using **`NvData_create`**.

hDataSet Lets you specify the data set to be accessed. The data set must be created before by using **`NvData_createDataSet`**.

DataType Lets you specify the data type of the elements contained in the data set. All elements in a data set must have the same data type.
The following data types are available in the **`NvDataEType`** enumeration.

Data Type	Meaning
NVDATA_TYPE_DOUBLE_FLOAT	Specifies 64-bit float values (8 bytes for one element)
NVDATA_TYPE_SINGLE_FLOAT	Specifies 32-bit float values (4 bytes for one element)
NVDATA_TYPE_UINT32	Specifies 32-bit unsigned integer values (4 bytes for one element)
NVDATA_TYPE_INT32	Specifies 32-bit signed integer values (4 bytes for one element)
NVDATA_TYPE_UINT16	Specifies 16-bit unsigned integer values (2 bytes for one element)
NVDATA_TYPE_INT16	Specifies 16-bit signed integer values (2 bytes for one element)

Data Type	Meaning
NVDATA_TYPE_UINT8	Specifies 8-bit unsigned integer values (1 byte for one element)
NVDATA_TYPE_INT8	Specifies 8-bit signed integer values (1 byte for one element)

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_DRIVER_NULL	A NULL driver object was passed to the function.
ERR_NVTABLE_INVALID_PARAM	The specified data type is invalid. Refer to the description of the `DataType` parameter for the valid values.

Related topics

References

NvData_create... 191
NvData_createDataSet.. 191
NvData_setDimension.. 194

NvData_write

Syntax

```
Int32 NvData_write(
   NvDataTDrv *pNvDataDrv,
   UInt32 hDataSet,
   const void *pSrcBuffer)
```

Include file

`NvDataRP.h`

Purpose

To write a data set to the board's nonvolatile memory.

Description

You can access a data set via its handler that you created by using the **`NvData_createDataSet`** function. The specified data set has to provide at least the memory size of the specified source buffer.

The memory size of the data set depends on the specified number of elements and the data types of the contained elements, refer to **NvData_setDimension** and **NvData_setType**.

The function is intended to be called during run time of the real-time application.

If an error is detected, the very first instance of the error is returned to the caller.

Parameters

pNvDataDrv Lets you specify the address of the driver object created before by using **NvData_create**.

hDataSet Lets you specify the data set to be accessed. The data set must be created before by using **NvData_createDataSet**.

pSrcBuffer Lets you specify the address of the buffer providing the data that you want to write to the board's nonvolatile memory.

Return value

The function returns an error code.

Error Code	Meaning
0	The function was successfully completed.
NVDATA_ERR_DRIVER_NULL	A NULL driver object was passed to the function.
NVDATA_ERR_INVALID_HANDLE	The `hDataSet` handle is invalid.

Related topics

References

NvData_create..191
NvData_createDataSet..191
NvData_setDimension..194
NvData_setType..197

Example of Implementing Access to the Nonvolatile Data

Introduction

The following example code shows you the order in which the NvData functions are to be used.

Using NvData functions

The example code shows you how to create and configure two data sets.

```
void NvData_Example()
{
   NvDataTDrv *pNVDrv;
   UInt32     hDataSet1, hDataSet2;
   char       Zero[256];

   // Create driver object
   NvData_create(&pNVDrv);

   // Configure first data set 'Seat_Position' with an array of 4 doubles
   NvData_createDataSet(pNVDrv, &hDataSet1);
   NvData_setName(pNVDrv, hDataSet1, "Seat_Position");
   NvData_setType(pNVDrv, hDataSet1, NVDATA_TYPE_DOUBLE_FLOAT);
   NvData_setDimension(pNVDrv, hDataSet1, 4);

   // Configure second data set 'Error_History' with an array of 32 unsigned
integers
   NvData_createDataSet(pNVDrv, &hDataSet2);
   NvData_setName(pNVDrv, hDataSet2, "Error_History");
   NvData_setType(pNVDrv, hDataSet2, NVDATA_TYPE_UINT32);
   NvData_setDimension(pNVDrv, hDataSet2, 32);

   // Apply and verify previously set configuration
   NvData_apply(pNVDrv);

   // Do an initial write of both data sets
   memset(Zero, 0, 256);
   NvData_write(pNVDrv, hDataetS1, Zero);
   NvData_write(pNVDrv, hDataSet2, Zero);
}
```

Special Processor Functions

Purpose

To ensure proper operation of the PowerPC.

Where to go from here

Information in this section

RTLIB_FORCE_IN_ORDER

Syntax

```
void RTLIB_FORCE_IN_ORDER(void)
```

Include file

`SrtkStd.h`

Purpose

To force the processor to execute the I/O accesses in order.

Description

This macro ensures that the PowerPC executes I/O accesses in the right order. For example, when two I/O accesses are performed sequentially, the PowerPC can change their order. If the `RTLIB_FORCE_IN_ORDER` macro is executed between the two accesses, they are executed in the specified order.

Return value

None

Related topics

References

RTLIB_SYNC........202

RTLIB_SYNC

Syntax	`void RTLIB_SYNC(void)`
Include file	`SrtkStd.h`
Purpose	To force the PowerPC to perform all pending memory accesses.
Description	This macro ensures that the PowerPC performs all memory accesses that were issued before the macro was called.
Return value	None
Related topics	References RTLIB_FORCE_IN_ORDER 201

Conversion Functions

Introduction	Use these macros to convert floating-point values to other formats. Conversion is necessary because the PowerPC and the TI slave processors on the I/O boards use different floating-point formats. The PowerPC uses the IEEE floating-point format, and the TI slave processor uses the TI floating-point format. TI floating-point values are stored as UInt32 because they are usually transferred to the external hardware through 32-bit I/O registers.

Where to go from here

Information in this section

RTLIB_CONV_FLOAT32_FROM_IEEE32

Syntax

```
Float32 RTLIB_CONV_FLOAT32_FROM_IEEE32(UInt32 ieee_32)
```

Include file

`SrtkStd.h`

Purpose

To convert a value in IEEE floating-point format to native floating-point format.

Parameters

ieee_32 Specifies the value in IEEE floating-point format.

Return value

This function returns the value in native floating-point format.

Related topics

References

RTLIB_CONV_FLOAT32_TO_IEEE32 204

RTLIB_CONV_FLOAT32_FROM_TI32

Syntax	`Float32 RTLIB_CONV_FLOAT32_FROM_TI32(UInt32 ti_32)`
Include file	`SrtkStd.h`
Purpose	To convert a value in TI floating-point format to IEEE floating-point format.
Parameters	**ti_32** Specifies the value in TI floating-point format.
Return value	This function returns the value in IEEE floating-point format.
Related topics	References

RTLIB_CONV_FLOAT32_TO_TI32 205

RTLIB_CONV_FLOAT32_TO_IEEE32

Syntax	`UInt32 RTLIB_CONV_FLOAT32_TO_IEEE32(Float32 val_32)`
Include file	`SrtkStd.h`
Purpose	To convert a value in native floating-point format to IEEE floating-point format.
Parameters	**val_32** Specifies the value in float32 format.
Return value	This function returns the value in IEEE floating-point format.
Related topics	References

RTLIB_CONV_FLOAT32_FROM_IEEE32 203

RTLIB_CONV_FLOAT_TO_SATURATED_INT32

Syntax	`Int32 RTLIB_CONV_FLOAT_TO_SATURATED_INT32(double fp_value)`
Include file	`SrtkStd.h`
Purpose	To convert a value in floating-point format to signed integer format.
Parameters	**fp_value** Specifies the value in floating-point format (float or double).
Return value	This function returns the value in signed integer format, possibly saturated.
Related topics	References

RTLIB_CONV_FLOAT32_FROM_IEEE32 203
RTLIB_CONV_FLOAT32_TO_IEEE32 204

RTLIB_CONV_FLOAT32_TO_TI32

Syntax	`UInt32 RTLIB_CONV_FLOAT32_TO_TI32(Float32 ieee_32)`
Include file	`SrtkStd.h`
Purpose	To convert a value in IEEE floating-point format to TI floating-point format.
Parameters	**ieee_32** Specifies the value in IEEE floating-point format.

Return value	This function returns the value in TI floating-point format.
Related topics	References

Standard Macros

Introduction	The include file `SrtkStd.h` defines several macros that can be used to program board-independent applications. For further information about the functionality of a macro, see either this topic or the description of the corresponding function.
Initialization	There is a macro to call the board-specific initialization routines. ▪ **`RTLIB_INIT`** on page 209
Application background	There is a macro that can be used to start all board-specific background functions. There are also standard functions for calling hook functions that are to run in the background of the application. ▪ **`RTLIB_BACKGROUND_SERVICE`** on page 20 ▪ **`rtlib_background_hook`** on page 20
End of application	There is a macro that can be used to terminate the application, for example, because if critical exceptions. Do not use it for a normal stop of an application. ▪ **`RTLIB_TERMINATE`** on page 210
Registering hook functions	There are macros that you can use to register functions that are to be called in the background service, the termination phase, or when you unload an application. ▪ **`RTLIB_REGISTER_BACKGROUND_HANDLER`** on page 211 ▪ **`RTLIB_REGISTER_TERMINATE_HANDLER`** on page 212 ▪ **`RTLIB_REGISTER_UNLOAD_HANDLER`** on page 213
Reading the board's serial number	There is a macro that you can use to get the serial number of your board. ▪ **`RTLIB_GET_SERIAL_NUMBER()`** on page 211
Interrupt handling	There are macros that can be used to enable or disable the interrupts globally. ▪ **`RTLIB_INT_ENABLE`** on page 102 ▪ **`RTLIB_INT_DISABLE`** on page 101
Sampling rate timer	There are macros to handle the default sampling rate timer. This is usually Timer A. ▪ **`RTLIB_SRT_START`** on page 85 ▪ **`RTLIB_SRT_PERIOD`** on page 59 ▪ **`RTLIB_SRT_ISR_BEGIN`** on page 84

- **RTLIB_SRT_ISR_END** on page 85
- **RTLIB_SRT_ENABLE** on page 105
- **RTLIB_SRT_DISABLE** on page 104

Time interval measurement

There are macros to be used for time interval measurement.

- **RTLIB_TIC_START** on page 42
- **RTLIB_TIC_READ** on page 41
- **RTLIB_TIC_READ_TOTAL** on page 42
- **RTLIB_TIC_HALT** on page 40
- **RTLIB_TIC_CONTINUE** on page 35
- **RTLIB_TIC_DELAY** on page 37
- **RTLIB_TIC_COUNT** on page 36
- **RTLIB_TIC_DIFF** on page 38
- **RTLIB_TIC_ELAPSED** on page 39

Floating-point conversion

There are macros to be used when converting floating-point values transferred from or to a TI slave processor of an I/O Board:

- **RTLIB_CONV_FLOAT32_TO_TI32** on page 205
- **RTLIB_CONV_FLOAT32_FROM_TI32** on page 204
- **RTLIB_CONV_FLOAT32_TO_IEEE32** on page 204
- **RTLIB_CONV_FLOAT32_FROM_IEEE32** on page 203
- **RTLIB_CONV_FLOAT_TO_SATURATED_INT32** on page 205

Memory allocation

There are macros to handle memory allocation that is protected against interrupt activities.

- **RTLIB_MALLOC_PROT** on page 214
- **RTLIB_CALLOC_PROT** on page 215
- **RTLIB_REALLOC_PROT** on page 215
- **RTLIB_FREE_PROT** on page 216

Processor functions

There are macros to handle the following Assembler commands.

- **RTLIB_FORCE_IN_ORDER** on page 201
- **RTLIB_SYNC** on page 202

init()

Purpose

To initialize the required hardware and software modules for a specific hardware system.

> **Note**
>
> It is recommended to use `RTLIB_INIT` for new applications to avoid naming conflicts with `init` functions in other software modules.

Syntax

```
void init(void)
```

Include file

`Brtenv.h`

Description

This macro calls the internal initialization functions of the hardware system.

> **Note**
>
> - I/O boards used within a PHS-bus-based system, like DS1006, or DS1007 are not initialized by calling `init()`.
> - The initialization function `init()` must be executed at the beginning of each application. It can only be invoked once. Further calls to `init()` are ignored.
> - When you use RTI, this function is called automatically in the simulation engine. Hence, you do not need to call `init()` in S-functions. If you need to initialize single components that are not initialized by `init()`, use the specific initialization functions that are described at the beginning of the function references.

Related topics

References

RTLIB_INIT 209

RTLIB_INIT

Purpose

To initialize the required hardware and software modules for a specific hardware system.

Syntax	`void RTLIB_INIT(void)`
Include file	`SrtkStd.h`

Description

This macro calls the internal initialization functions of the specified hardware system.

> **Note**
>
> - I/O boards used within a DS1007 modular board system are not initialized by calling `RTLIB_INIT()`.
> - The initialization function `RTLIB_INIT()` must be executed at the beginning of each application. It can only be invoked once. Further calls to `RTLIB_INIT()` are ignored.
> - When you use RTI, this function is called automatically in the simulation engine. Hence, you do not need to call `RTLIB_INIT()` in S-functions. If you need to initialize single components that are not initialized by `RTLIB_INIT()`, use the specific initialization functions that are described at the beginning of the function references.

RTLIB_TERMINATE

Purpose	To terminate the application.
Syntax	`void RTLIB_TERMINATE(void)`
Include file	`SrtkStd.h`

Description

Usually, a real-time application is stopped by the host PC. You can handle this behavior by implementing termination code, but only if a critical error occurs during run time of the application.

A stopped application remains in the memory (RAM or flash) of the hardware and can be started again immediately. A terminated application also remains in the memory of the hardware, you must first reload it to restart it.

The `RTLIB_TERMINATE` macro behaves in the same way as the `exit` function, or a `main` function that finishes with a `return` statement. The exit code or the return code are ignored.

Note

To register a function that is to be executed in the termination phase, you must use **`RTLIB_REGISTER_TERMINATE_HANDLER`**. Do not use `atexit`.

Functions that are to be executed when you unload an application can be registered by using **`RTLIB_REGISTER_UNLOAD_HANDLER`**.

Related topics

References

RTLIB_GET_SERIAL_NUMBER()

Purpose

To get the serial number of the processor board.

Syntax

```
RTLIB_GET_SERIAL_NUMBER()
```

Include file

`SrtkStd.h`

Description

This macro returns the serial number as UInt32 data type.

RTLIB_REGISTER_BACKGROUND_HANDLER

Purpose

To register a function that is called by the background service.

Syntax

```
Int32 RTLIB_REGISTER_BACKGROUND_HANDLER(SrtkTApp_Handler pHandler)
```

Include file

`SrtkStd.h`

Description

You can use multiple calls of this macro to register more than one function. The functions are then executed in their registration order.

The registered background function should not contain endless loops or functions that might block the application.

Parameters

pHandler Specifies the pointer to the handler of the function to be registered. The handler function must be of `SrtkTApp_Handler` type, i.e., `void MyBackgroundHandler(void)`.

Return value

This function returns an error code.

Symbol	Meaning
SRTK_ERROR	The registration failed.
SRTK_NO_ERROR	The registration was performed without error.

Related topics

References

RTLIB_REGISTER_TERMINATE_HANDLER

Purpose

To register a function that is called in the termination phase.

Syntax

```
Int32 RTLIB_REGISTER_TERMINATE_HANDLER(SrtkTApp_Handler pHandler)
```

Include file

`SrtkStd.h`

Description

You can use multiple calls of this macro to register more than one function. The functions are then executed in the reverse order of their registration.

The registered termination function should not contain endless loops or functions that might block the application.

Parameters

pHandler Specifies the pointer to the handler of the function to be registered. The handler function must be of `SrtkTApp_Handler` type, i.e., `void MyBackgroundHandler(void)`.

Return value

This function returns an error code.

Symbol	Meaning
SRTK_ERROR	The registration failed.
SRTK_NO_ERROR	The registration was performed without error.

Related topics

References

RTLIB_REGISTER_BACKGROUND_HANDLER..211
RTLIB_REGISTER_UNLOAD_HANDLER..213

RTLIB_REGISTER_UNLOAD_HANDLER

Purpose

To register a function that is called when you unload an application.

Syntax

```
Int32 RTLIB_REGISTER_UNLOAD_HANDLER(SrtkTApp_Handler pHandler)
```

Include file

`SrtkStd.h`

Description

You can use multiple calls of this macro to register more than one function. The functions are then executed in the reverse order of their registration.

The registered termination function should not contain endless loops or functions that might block the application.

Parameters

pHandler Specifies the pointer to the handler of the function to be registered. The handler function must be of `SrtkTApp_Handler` type, i.e. `void MyBackgroundHandler(void)`.

Return value

This function returns an error code.

Symbol	Meaning
SRTK_ERROR	The registration failed.
SRTK_NO_ERROR	The registration was performed without error.

Related topics

References

RTLIB_REGISTER_BACKGROUND_HANDLER........211
RTLIB_REGISTER_TERMINATE_HANDLER........212

RTLIB_MALLOC_PROT

Purpose

To allocate memory with protection against interrupts by using the `malloc` routine of the standard C library.

Tip

This macro is provided for backward compatibility only. On the DS1007, use the routines for memory allocation (`malloc`, `calloc`, `realloc` and `free`).

Syntax

```
RTLIB_MALLOC_PROT(void *pointer, UInt32 size)
```

Include file

`SrtkStd.h`

Parameters

pointer Specifies the address of the allocated buffer.

size Specifies the memory size to be allocated.

Related topics

References

RTLIB_CALLOC_PROT........215
RTLIB_FREE_PROT........216
RTLIB_REALLOC_PROT........215

RTLIB_CALLOC_PROT

Purpose

To allocate memory for an array with protection against interrupts by using the `calloc` routine of the standard C library.

> **Tip**
>
> This macro is provided for backward compatibility only. On the DS1007, use the routines for memory allocation (`malloc`, `calloc`, `realloc` and `free`).

Syntax

```
RTLIB_CALLOC_PROT(void *pointer, UInt32 nobj, UInt32 size)
```

Include file

`SrtkStd.h`

Parameters

pointer Specifies the address of the allocated buffer.

nobj Specifies the number of elements.

size Specifies the size of one element.

Related topics

References

RTLIB_FREE_PROT 216
RTLIB_MALLOC_PROT 214
RTLIB_REALLOC_PROT 215

RTLIB_REALLOC_PROT

Purpose

To change the memory size with protection against interrupts by using the `realloc` routine of the standard C library.

> **Tip**
>
> This macro is provided for backward compatibility only. On the DS1007, use the routines for memory allocation (`malloc`, `calloc`, `realloc` and `free`).

Syntax

```
RTLIB_REALLOC_PROT(void *pointer, UInt32 size)
```

Include file

`SrtkStd.h`

Parameters

pointer Specifies the address of the allocated buffer.

size Specifies the memory size to be allocated.

Related topics

References

RTLIB_FREE_PROT

Purpose

To free the allocated memory with protection against interrupts by using the `free` routine of the standard C library.

Tip

This macro is provided for backward compatibility only. On the DS1007, use the routines for memory allocation (`malloc`, `calloc`, `realloc` and `free`).

Syntax

```
RTLIB_FREE_PROT(void *pointer)
```

Include file

`SrtkStd.h`

Parameters

pointer Specifies the address of the buffer to be freed.

Related topics

References

I/O Modules

Introduction

A PHS-bus-based system consists of a processor board and one or more I/O boards. Here you get an overview on the available I/O boards for a DS1007 modular system and the functions to handle the PHS bus that connects the processor board with the installed I/O boards.

Where to go from here

Information in this section

I/O Boards

Where to go from here

Information in this section

A/D Conversion

Board overview

The following dSPACE boards can be controlled by the DS1007 board to perform A/D conversion. See:

- DS2001 RTLib Reference
  for the DS2001 High-Speed A/D Converter Board
- DS2002 RTLib Reference
  for the DS2002 Multi-Channel A/D Converter Board
- DS2003 RTLib Reference
  for the DS2003 Multi-Channel A/D Converter Board
- DS2004 RTLib Reference
  for the DS2004 High-Speed A/D Converter Board
- DS2201 RTLib Reference
  for the DS2201 Multi-I/O Board
- DS2202 RTLib Reference
  for the DS2202 HIL I/O Board

D/A Conversion

Board overview

The following dSPACE boards can be controlled by the DS1007 board to perform D/A conversion. See:

- DS2101 RTLib Reference
  for the DS2101 D/A Converter Board
- DS2102 RTLib Reference
  for the DS2102 High-Resolution D/A Converter Board
- DS2103 RTLib Reference
  for the DS2103 Multi-Channel D/A Converter Board
- DS2201 RTLib Reference
  for the DS2201 Multi-I/O Board
- DS2202 RTLib Reference
  for the DS2202 HIL I/O Board

Automotive Signal Generation and Measurement

Board overview

The following dSPACE boards can be controlled by the DS1007 board for autonomous signal generation. See:

- DS2202 RTLib Reference
  for the DS2202 HIL I/O Board
- DS2210 RTLib Reference
  for the DS2210 HIL I/O Board
- DS2211 RTLib Reference
  for the DS2211 HIL I/O Board
- DS2302 RTLib Reference
  for the DS2302 Direct Digital Synthesis Board

Bit I/O

Board overview

The following dSPACE boards can be controlled by the DS1007 board to perform bit I/O. See:

- DS2201 RTLib Reference
  for the DS2201 Multi-I/O Board
- DS2202 RTLib Reference
  for the DS2202 HIL I/O Board

- DS2301 RTLib Reference
  for the DS2301 Direct Digital Synthesis Board
- DS2302 RTLib Reference
  for the DS2302 Direct Digital Synthesis Board
- DS4001 RTLib Reference
  for the DS4001 Timing and Digital I/O Board
- DS4002 RTLib Reference
  for the DS4002 Timing and Digital I/O Board
- DS4003 RTLib Reference
  for the DS4003 Digital I/O Board

Timing I/O

Board overview

The following dSPACE boards can be controlled by the DS1007 board to perform timing I/O, such as the generation of various pulse patterns including PWM or the capture of digital frequency signals. See:

- DS2201 RTLib Reference
  for the DS2201 Multi-I/O Board
- DS2301 RTLib Reference
  for the DS2301 Direct Digital Synthesis Board
- DS2302 RTLib Reference
  for the DS2302 Direct Digital Synthesis Board
- DS4001 RTLib Reference
  for the DS4001 Timing and Digital I/O Board
- DS4002 RTLib Reference
  for the DS4002 Timing and Digital I/O Board
- DS5001 RTLib Reference
  for the DS5001 Digital Waveform Capture Board
- DS5101 RTLib Reference
  for the DS5101 Digital Waveform Output Board

Interface Boards

Board overview

The following dSPACE boards can be controlled by the DS1007 board to integrate more specialized custom devices into the dSPACE real-time system. See:

- DS3001 RTLib Reference
  for the DS3001 Incremental Encoder Interface Board

- DS3002 RTLib Reference 📖
  for the DS3002 Incremental Encoder Interface Board
- DS4201 RTLib Reference 📖
  for the DS4201 Prototyping Board
- DS4201-S RTLib Reference 📖
  for the DS4201-S Serial Interface Board
- DS4302 RTLib Reference 📖
  for the DS4302 CAN Interface Board
- DS4330 RTLib Reference 📖
  for the DS4330 LIN Interface Board

Special I/O

Board overview

The following dSPACE boards can be controlled by the DS1007 board to perform more specialized I/O. See:

- DS2301 RTLib Reference 📖
  for the DS2301 Direct Digital Synthesis Board
- DS2302 RTLib Reference 📖
  for the DS2302 Direct Digital Synthesis Board
- DS2401 RTLib Reference 📖
  for the DS2401 Resistive Sensor Simulation Board

Integration of FPGA Applications

Board overview

The following dSPACE board can be controlled by the DS1007 board to perform custom FPGA applications. See:

- DS5202 RTLib Reference 📖
  for the DS5202 FPGA Base Board
- DS5203 RTLib Reference 📖
  for the DS5203 FPGA Board

PHS-Bus Handling

Introduction

Use these functions to handle the PHS bus, which is used for communication between the DS1007 processor board and the I/O boards.

I/O board base address

When using I/O board functions you always need the board's base address as parameter. This address can simply be obtained by using the `DSXXXX_n_BASE` macros where `DSXXXX` is the board name (e.g., DS2001) and `n` is an index which counts boards of the same type. The board with the lowest base address gets the index 1. The other boards of the same type get the consecutive numbers in order of their base addresses.

The macros refer to an internal data structure, which holds the addresses of all I/O boards in the system. This data structure is created during the initialization phase. Hence, when changing an I/O board base address, it is not necessary to recompile the code of your application.

> **Note**
>
> The `DSXXXX_n_BASE` macros can only be used after the initialization function was called.

Example

This example demonstrates the using of the `DSXXXX_n_BASE` macros. There are two DS2001 boards, two DS2101 boards and one DS2002 board connected to a PHS bus. Their base addresses have been set to distinct addresses. The following table shows the I/O boards, their base addresses and the macros which can be used as base address.

Board	Base address (Hex)	Macro
DS2001	00	`DS2001_1_BASE`
DS2002	20	`DS2002_1_BASE`
DS2101	80	`DS2101_1_BASE`
DS2001	90	`DS2001_2_BASE`
DS2101	A0	`DS2101_2_BASE`

Programmable signals

Three of PHS-bus signals are programmable by the user:

I/O error line The DS1007 processor board and the I/O boards can activate the I/O error line if an error occurred. Thus, the other devices are able to react individually.

SYNCIN line A pulse in the SYNCIN line triggers the input channels of all I/O boards to sample their input values.

SYNCOUT line A pulse in the SYNCOUT line triggers the output channels of all I/O boards to update their input values.

Where to go from here

Information in this section

To handle the SYNCIN and SYNCOUT line

get_peripheral_addr

Syntax

```
phs_addr_t get_peripheral_addr(
      UInt32 board_id,
      int board_no)
```

Include file

`dsphs.h`

Purpose

To get the base address of an I/O board specified by board ID and board number.

Parameters

board_id Specifies the ID of the I/O board. For each I/O board, there is a symbol predefined as `DSxxxx_BOARD_ID`, where DSxxxx stands for the board name. For example, to get the base address of a DS2210 I/O board, you must specify `DS2210_BOARD_ID` for this parameter.

board_no Specifies the board number that distinguishes boards with the same `board_id`.

Return value

This function returns the I/O board base address or 0xFFFFFFFF if the I/O board could not be found in the PHS bus.

PHS_BOARD_BASE_GET

Syntax

```
phs_addr_t PHS_BOARD_BASE_GET(
      int board_type,
      int board_no)
```

Include file

`dsphs.h`

Purpose

To get the base address of an I/O board specified by board type and board number.

Parameters

board_type Specifies the type number of the I/O board. The following symbols are predefined:

Predefined Symbol	Meaning
PHS_BT_NO_BOARD	No I/O board at the specified PHS-bus address
PHS_BT_UNKNOWN	I/O board type is not known
PHS_BT_INVALID_BASE	Specified PHS-bus address is not valid
PHS_BT_DSxxxx	Specifies the board type, where DSxxxx stands for the board name. For example, you must use `PHS_BT_DS2210` to specify the board type of a DS2210 I/O board.

board_no Specifies the board number that distinguishes boards with the same `board_id`.

Return value

This function returns the I/O board base address or `PHS_INVALID_BASE` if the I/O board could not be found in the PHS bus.

Related topics

References

phs_board_type_get 225

phs_board_type_get

Syntax

```
int phs_board_type_get(phs_addr_t base)
```

Include file

`dsphs.h`

Purpose

To identify the type of an I/O board at the specified PHS-bus address.

Parameters

base Specifies the I/O board base address, refer to PHS-Bus Handling on page 222.

Return value

This function returns the I/O board type or error code. The following symbols are predefined:

Predefined Symbol	Meaning
PHS_BT_NO_BOARD	No I/O board at the specified PHS-bus address
PHS_BT_UNKNOWN	I/O board type is not known
PHS_BT_INVALID_BASE	Specified PHS-bus address is not valid
PHS_BT_DSxxxx	Returned board type, where DSxxxx stands for the board name. For example, the function returns `PHS_BT_DS2210` for a DS2210 I/O board.

Related topics

References

phs_board_type_from_slot_get

Syntax

```
int phs_board_type_from_slot_get(int slot_number)
```

Include file

`dsphs.h`

Purpose

To identify the type of an I/O board at the specified PHS-bus slot number.

Parameters

slot_number Specifies the PHS-bus slot number in the range of 0 … 15.

Return value

This function returns the I/O board type or error code. The following symbols are predefined:

Predefined Symbol	Meaning
PHS_BT_NO_BOARD	No I/O board at the specified PHS-bus address
PHS_BT_UNKNOWN	I/O board type is not known
PHS_BT_INVALID_BASE	Specified PHS-bus address is not valid
PHS_BT_DSxxxx	Returned board type, where DSxxxx stands for the board name. For example, the function returns `PHS_BT_DS2210` for a DS2210 I/O board.

Related topics

References

phs_board_type_get........225

PHS_REGISTER_READ

Syntax

```
phs_data_u_t PHS_REGISTER_READ(
      phs_data_u_t base,
      phs_data_u_t offset)
```

Include file

`dsphs.h`

Purpose

To read a value from the register at the specified offset from the specified PHS-bus board base address.

Parameters

base Specifies the I/O board base address, refer to PHS-Bus Handling on page 222.

offset Specifies the offset for the register address within the range 0x00 … 0x0F.

Return value

This function returns the contents of the specified PHS-bus register.

Related topics

References

PHS_REGISTER_WRITE

Syntax

```
void PHS_REGISTER_WRITE(
      phs_data_u_t base,
      phs_data_u_t offset,
      phs_data_u_t value)
```

Include file

`dsphs.h`

Purpose

To write a value to the register at the specified offset from the specified PHS-bus board base address.

Parameters

base Specifies the I/O board base address, refer to PHS-Bus Handling on page 222.

offset Specifies the offset for the register address within the range 0x00 … 0x0F.

value Specifies the value to be written to the specified PHS-bus register.

Return value

None

Related topics

References

PHS_REGISTER_PTR

Syntax

```
phs_data_u_t* PHS_REGISTER_PTR(
      phs_data_u_t base,
      phs_data_u_t offset)
```

Include file

`dsphs.h`

Purpose

To get the register address at the specified offset from the specified PHS-bus board base address.

Parameters

base Specifies the I/O board base address, refer to PHS-Bus Handling on page 222.

offset Specifies the offset for the register address within the range 0x00 … 0x0F.

Return value

This function returns the address of the register.

Related topics

References

PHS_IO_ERROR_STATE

Syntax

```
PHS_IO_ERROR_STATE()
```

Include file

`dsphs.h`

Purpose

To get the state of the I/O error line.

Return value

This macro returns the state of the I/O error line:

Value	Meaning
TRUE	I/O error line is active
FALSE	I/O error line is not active

Related topics

References

PHS_IO_ERROR_SET....230

PHS_IO_ERROR_SET

Syntax

```
PHS_IO_ERROR_SET(state)
```

Include file

`dsphs.h`

Purpose

To activate or deactivate the I/O error line.

Parameters

state Specifies the state of the I/O error line:

Value	Meaning
TRUE	Activate I/O error line
FALSE	Deactivate I/O error line

Return value

None

Related topics

References

PHS_IO_ERROR_STATE....229

PHS_SYNCIN_TRIGGER

Syntax

```
PHS_SYNCIN_TRIGGER()
```

Include file

`dsphs.h`

Purpose

To generate a pulse on the PHS-bus SYNCIN line.

Return value

None

Related topics

References

PHS_SYNC_TRIGGER 232
PHS_SYNCOUT_TRIGGER 231

PHS_SYNCOUT_TRIGGER

Syntax

```
PHS_SYNCOUT_TRIGGER()
```

Include file

`dsphs.h`

Purpose

To generate a pulse on the PHS-bus SYNCOUT line.

Return value

None

Related topics

References

PHS_SYNC_TRIGGER 232
PHS_SYNCIN_TRIGGER 231

PHS_SYNC_TRIGGER

Syntax

```
PHS_SYNC_TRIGGER()
```

Include file

`dsphs.h`

Purpose

To generate a pulse on the PHS-bus SYNCIN and the SYNCOUT line simultaneously.

Return value

None

Related topics

References

PHS_SYNC_TIMER_SET

Syntax

```
PHS_SYNC_TIMER_SET(mode)
```

Include file

`dsphs.h`

Purpose

To select Timer A or Timer B as the trigger source for the SYNCIN or SYNCOUT lines.

Parameters

mode Specifies the mode to be set. Combine the following predefined symbols with the logical operator OR:

Predefined Symbol	Meaning
PHS_SYNCIN_DISABLE	SYNCIN line disabled
PHS_SYNCIN_TIMERA	SYNCIN line triggered by Timer A
PHS_SYNCIN_TIMERB	SYNCIN line triggered by Timer B
PHS_SYNCOUT_DISABLE	SYNCOUT line disabled

Predefined Symbol	Meaning
PHS_SYNCOUT_TIMERA	SYNCOUT line triggered by Timer A
PHS_SYNCOUT_TIMERB	SYNCOUT line triggered by Timer B

Return value

None

Related topics

References

PHS-Bus Interrupt Handling

PHS bus

The PHS bus (Peripheral High Speed Bus) is used in dSPACE systems to connect I/O and processor boards. The PHS bus supports an extended interrupt system, which adds 64 external interrupts to the standard interrupts recognized by the Real-Time Processor (RTP). The following topics describe the PHS-bus interrupt functions, which provide an easy-to-use, high-level programmer's interface to PHS-bus interrupts. With this software the operation of the extended interrupt system becomes completely transparent to the user.

> **Note**
>
> The PHS-bus interrupt functions may be used only in handcoded applications. Using them in Simulink applications (User-Code or S-functions) conflicts with the internal interrupt handling.

Where to go from here

Information in this section

General Information on PHS-Bus Interrupts

Where to go from here

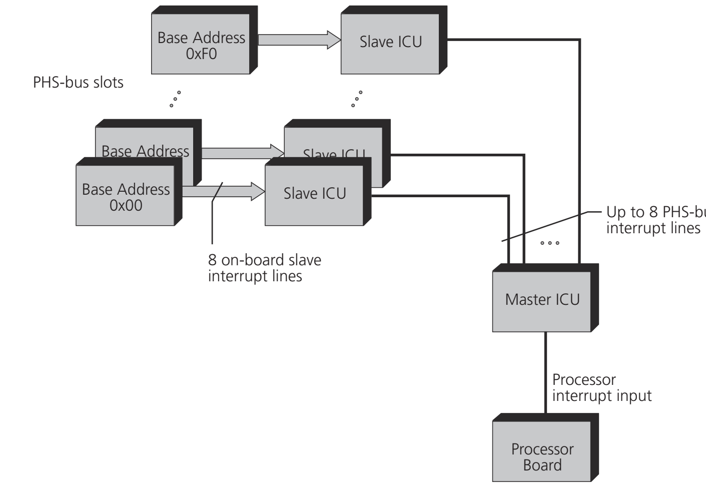
Information in this section

Basics of PHS-Bus Interrupts

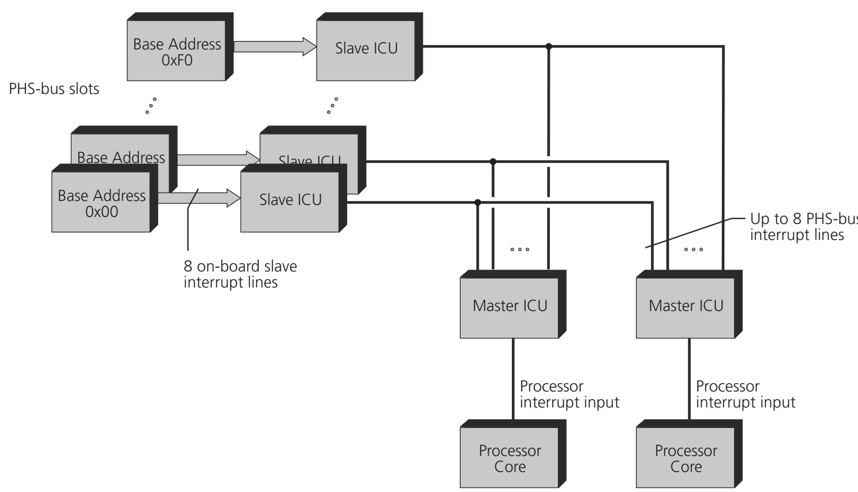
PHS-bus interrupt system

The following illustrations show the extended PHS-bus interrupt system.

Using a single-core processor board.



Using a multicore processor board.



Interrupt control unit

The processor board is equipped with one master interrupt control unit (ICU) per core. The master ICUs are separately configured and can issue interrupts only to the related CPU core. Each master ICU is connected to the eight interrupt lines of the PHS bus (see illustration above). An application always runs on one CPU core only and therefore it uses only the ICU that corresponds to the CPU core. All interrupt requests from these interrupt lines are mapped to an interrupt pin of the processor, one pin per CPU core. I/O boards with interrupt generating devices are provided with on-board slave interrupt controller units. The eight slave interrupt lines supported by each of the interrupt controllers are wired to the various board-specific interrupt sources, such as A/D converters, timers, etc. Each interrupt controller can be connected to one of the eight PHS-bus interrupt lines by programming three bits in the setup register of the respective I/O board. Thus, a maximum of 64 prioritized PHS-bus interrupts is added to the processor's interrupt system. For more information on the extended PHS-bus interrupt system, refer to the hardware reference manual of your processor board.

The standard initialization procedure provided by the Real-Time Library (RTLib) initializes the slave interrupt controllers for polling mode with all I/O interrupts disabled. The master interrupt controller on the processor board is not initialized. PHS-bus interrupts are disabled in the interrupt enable register of the processor.

Note

Initially, all slave interrupt controllers are connected to interrupt line 0 on the PHS bus. This is the default interrupt line number, which is reserved for I/O boards operated in polling mode.

In order to handle PHS-bus interrupts, the PHS-bus interrupt functions will both initialize the master ICU and reinitialize the slave ICUs that are selected for interrupt mode.

Please note that it is possible to use mixed-mode operation of several peripheral boards. Some boards may be operated in interrupt mode while others may use the standard polling mode.

Management of the Extended Interrupt System

Managing the interrupt system

The slave interrupt controller on a dSPACE I/O board can be connected to one of eight PHS-bus interrupt lines by programming its setup register. For more information, refer to Board Identification and PHS-Bus Interrupt Line Programming on page 236.

All PHS-bus interrupts are mapped to an interrupt input of the processor. How the PHS-bus interrupts are processed is explained in PHS-Bus Interrupt Processing on page 238.

The processor provides the means for hardware prioritization of the standard interrupts. For more information, refer to Interrupt Priorities on page 239.

Board Identification and PHS-Bus Interrupt Line Programming

Introduction

The slave interrupt controller on a dSPACE I/O board can be connected to one of eight PHS-bus interrupt lines by programming three bits in the setup register of the board. The PHS-bus interrupt functions use the dSPACE board identification scheme to find out which I/O boards are connected to the PHS bus and how to program the interrupt line numbers in the corresponding setup registers. The identification number consists of two 4-bit fields in the board's identification register (ID and SUB-ID field). The following table shows the relationship between board identification numbers and interrupt line programming.

Board ID	Board SUB-ID	Board Type	Interrupt Line Programming
0 … 13	don't care	Standard dSPACE I/O board	Provided by PHS-bus interrupt functions
14	0 … 15	Standard dSPACE I/O board	Provided by PHS-bus interrupt functions
15	0 … 14	Customer I/O board with ▪ Static interrupt line ▪ Programmable interrupt line	None (customer-provided)
15	15	No board	None

The ID and SUB-ID values of the standard I/O boards are defined by dSPACE. For standard I/O boards, the PHS-bus interrupt functions employ the built-in code for assigning and programming the interrupt line numbers. To install an interrupt handler, you only need to call `install_phs_int_vector`. To ensure proper operation, any previous setting of the interrupt lines is overridden. No part of the User-Code should directly modify the interrupt line numbers. Line number 0 is the default interrupt line and reserved for I/O boards that operate in polling mode. Thus, a maximum of seven PHS-bus boards can be used simultaneously in interrupt mode.

Customer-specific I/O boards

In addition to the standard I/O boards recognized by the PHS-bus interrupt functions, a dSPACE system may contain customer-specific I/O boards for which the automatic programming of the PHS-bus interrupt lines does not work. This holds in particular for the DS4201 prototyping board and for boards that are derived from it. (Nevertheless it is assumed that these nonstandard boards are provided with a standard PHS-bus interface including the slave interrupt controller.)

Some boards may use static PHS-bus interrupt lines that are defined by hardware settings. In this case you must call `declare_phs_int_line` for each of them. Other boards may be equipped with nonstandard setup registers, for which the customer has to provide the appropriate initialization code. `alloc_phs_int_line` must be called for these.

Note

Customer I/O boards must use ID 15 in order to be distinguishable from the standard dSPACE I/O boards. The SUB-ID must be in the range 0 … 14. A customer I/O board with a SUB-ID of 15 cannot be detected.

PHS-Bus Interrupt Processing

PHS-bus interrupt vector table

All PHS-bus interrupts are mapped to an interrupt input of the processor core and thus serviced by a common master interrupt service routine. The address of this master interrupt handler is installed in the processor's interrupt vector table. The vector table is created and maintained by the standard processor run-time environment to handle the standard processor interrupts. We will refer to this vector table as the system interrupt vector table.

To handle peripheral board interrupts, an interrupt dispatcher is needed for switching to the appropriate slave interrupt service routine. As a maximum of 16 PHS-bus boards can be connected to a processor board, each providing a maximum of eight slave interrupt sources, a second interrupt vector table with 128 entries must be maintained. This vector table will be referred to as the PHS-bus interrupt vector table.

Whenever an I/O interrupt is requested, the corresponding slave interrupt controller generates an interrupt vector number by adding the number of the interrupt service to a predefined board interrupt vector offset. The PHS-bus interrupt functions automatically assign each I/O board its own vector offset dependent on the PHS-bus base address of the board. As shown in the table below, a maximum of eight interrupt sources is associated with each PHS-bus base address. Therefore, the board interrupt vector offset is set to 0 for the board at PHS-bus base address 0x00, 8 for the board at base address 0x10, etc. The master interrupt handler reads the resulting vector number as generated by the slave interrupt controller and calls the service routine, whose address is found in the corresponding entry of the PHS-bus interrupt vector table.

In order to install or uninstall interrupt handlers, the PHS-bus library functions need to perform complete initialization sequences of the corresponding interrupt controller chips. As a consequence, each call to `install_phs_int_vector` or `deinstall_phs_int_vector` will clear all pending PHS-bus interrupt requests.

The following table shows the PHS-bus interrupt vector table.

PHS-bus Base Address	Board Interrupt Vector Offset	Slave Interrupt Number	Resulting Interrupt Vector Number
0x00	0	0 … 7	0 … 7
0x10	8	0 … 7	8 … 15
0x20	16	0 … 7	16 … 23
…	…	…	…
0xF0	120	0 … 7	120 … 127

Interrupt Priorities

Introduction

The processor provides the means for hardware prioritization of the standard interrupts according to their positions in the system interrupt vector table. The closer an interrupt's vector is to the base address of the vector table, the higher its hardware priority. This prioritization, however, applies only when more than one interrupt request is received in the same clock cycle. In that case, the interrupt with the highest priority is serviced first. On acceptance of an interrupt, the corresponding interrupt service routine is called with all interrupts globally disabled in the processor status register. For more information on the interrupt system, refer to the related documentation of the processor hardware.

Prioritization

The PHS-bus interrupt controllers use a similar hardware prioritization scheme according to interrupt line numbers. After initialization, interrupt requests at interrupt line 0 are of highest priority, those at interrupt line 7 are of lowest priority. The initial prioritization of the interrupt lines can be changed dynamically by sending priority rotation commands to the chip. This feature is currently not used by dSPACE software, leaving interrupt line 0 at the highest priority level all the time. This holds for the master interrupt controller on the processor board as well as for the slave interrupt controllers on the I/O boards. Consequently, the slave interrupt source at interrupt line 0 of the I/O board connected to PHS-bus interrupt line 1 represents the external interrupt with the highest priority, while slave interrupt 7 of the board connected to PHS-bus interrupt line 7 is of lowest priority. Remember that PHS-bus interrupt line 0 is reserved for boards operated in polling mode.

This prioritization again applies when more than one interrupt request is received in the same clock cycle. In that case, the interrupt with the highest priority is serviced first. On acceptance of an interrupt, the corresponding interrupt service routine is called with all interrupts globally disabled in the processor status register. `alloc_phs_int_line` and `install_phs_int_vector` assign interrupt lines to I/O boards in ascending order. If hardware prioritization of the PHS-bus interrupts is of importance, you should carefully adjust the sequence of function calls for allocating PHS-bus interrupt lines and installing interrupt handlers.

Finished-Interrupt command

The PHS-bus interrupt functions operate the interrupt controllers in FI Command mode. A Finished-Interrupt (FI) command is issued by software to acknowledge an interrupt. You can flexibly realize a software prioritization scheme according to the specific needs of your application.

> **Note**
>
> In order to implement preemptable interrupt handlers, you will have to enable interrupts at the beginning of the interrupt service routine and disable them at the end of it. The macros `RTLIB_INT_ENABLE` and `RTLIB_INT_DISABLE` provided by the Real-Time Library can be used for this purpose.
>
> It is necessary to disable interrupts globally before leaving an interrupt handler, because the context switch at the end of the interrupt service is a critical section that must not be interrupted.
>
> Consider reentrancy and overrun problems, which may be encountered if interrupts are enabled within an interrupt handler.

How to Program PHS-Bus Interrupts

Instructions

> **Note**
>
> The PHS-bus interrupt functions may be used only in handcoded applications. Using them in Simulink applications (User-Code or S-functions) conflicts with the internal interrupt handling.

To use the PHS-bus interrupt functions, the header file `brtenv.h` must be included in your source files. `brtenv.h` includes the header file for the PHS-bus interrupts functions `phsint.h`.

The following table summarizes the required sequences of function calls for different types of I/O boards.

Standard dSPACE I/O Board	Customer I/O Board with	
	Static Interrupt Line	**Programmable Interrupt Line**
	`declare_phs_int_line`	`alloc_phs_int_line`
		↓
	↓	board-specific interrupt line programming
		↓
`install_phs_int_vector`	`install_phs_int_vector`	`install_phs_int_vector`
↓	↓	↓
disable / enable macros (if required)	disable / enable macros (if required)	disable / enable macros (if required)
↓	↓	↓

Standard dSPACE I/O Board	Customer I/O Board with	
	Static Interrupt Line	Programmable Interrupt Line

`deinstall_phs_int_vector`	`deinstall_phs_int_vector`	`deinstall_phs_int_vector`
		↓
		`free_phs_int_line`
		↓
		board-specific interrupt line programming (set to default)

Unless otherwise specified, the functions return 0 on successful completion. As the functions are used to initialize the interrupt system, most of the errors must be considered severe programming faults, preventing the application from executing. An error message is generated and displayed within the dSPACE experiment software by means of the Real-Time Message Module. The error message contains the error code as well as a string describing the error and the name of the function where it occurred (see Error Codes of PHS-Bus Interrupt Functions on page 254). Program execution is automatically terminated by calling the `exit()` function.

Related topics

Basics

References

PHS-Bus Interrupt Functions

Where to go from here

Information in this section

install_phs_int_vector

Syntax

```
int install_phs_int_vector (
   phs_addr_t base,
   int n,
   void (*isr)())
```

Include file

`phsint.h`

Purpose

To install a PHS-bus interrupt handler.

Description

The corresponding I/O interrupt source as well as PHS-bus interrupts in general are enabled. Interrupts are not enabled globally. If the interrupt is the first one to be installed for the indicated I/O board, the board's interrupt controller is connected to a free PHS-bus interrupt line. This does not hold for customer I/O boards (such as the DS4201 Prototyping Board and its derivatives) which use static interrupt lines or customer-specific setup registers. For these boards, `declare_phs_int_line` or `alloc_phs_int_line` must be called before calling `install_phs_int_vector`.

> **Note**
>
> The vector can be uninstalled with `deinstall_phs_int_vector`. `deinit_phs_int` must not be applied to PHS-bus interrupts a vector is installed for.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

n Specifies the slave interrupt number. Valid numbers are defined by the mapping of interrupt sources to inputs 0 … 7 of the corresponding slave interrupt controller. Further information can be found in the *PHS Bus System Hardware Reference* and in the *RTLib Reference* of your I/O board.

isr Specifies the entry point address of the interrupt handler to be installed.

Related topics

Basics

HowTos

References

deinstall_phs_int_vector

Syntax

```
int deinstall_phs_int_vector(
   phs_addr_t base,
   int n)
```

Include file

`phsint.h`

Purpose

To uninstall a registered interrupt service routine.

Description

This is the counterpart of `install_phs_int_vector`. The corresponding I/O interrupt source is disabled. If the interrupt is the last one to be uninstalled for the specified I/O board, the interrupt controller of the board is reconnected to the default PHS-bus interrupt line (0), which is reserved for boards operated in polling mode. If the interrupt was the last enabled PHS-bus interrupt of all, PHS-bus interrupts (not interrupts in general) are disabled.

Note

This function must not be used to deinitialize a PHS-bus interrupt initialized with `init_phs_int`. Use `deinit_phs_int` for this purpose.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

n Specifies the slave interrupt number. Valid numbers are defined by the mapping of interrupt sources to inputs 0 … 7 of the corresponding slave interrupt controller. Further information can be found in the *PHS Bus System Hardware Reference* and in the *RTLib Reference* of your I/O board.

Related topics

Basics

HowTos

References

declare_phs_int_line

Syntax

```
int declare_phs_int_line(
   phs_addr_t base,
   int line)
```

Include file

`phsint.h`

Purpose

To declare a PHS-bus interrupt line.

Description

This function provides the means for declaring PHS-bus interrupt lines that are in use by nonstandard I/O boards. Normally, it is called once for each of the boards before any other function of the PHS-bus interrupt functions is called in order to prevent the system from allocating hard-wired interrupt lines to other PHS-bus boards. It is assumed that the interrupt line numbers of these boards have been statically defined by hardware and therefore cannot be modified by software.

> **Note**
>
> This function must be called before installing interrupt handlers for the specified board. Otherwise the function will issue an error message (error code `PHSINT_FUNC_NOT_ALLOWED`) and terminate the application.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

line Specifies the statically defined PHS-bus interrupt line number of the board. The value of `line` must match the hardware setting of the PHS-bus interrupt line number of the board.

Related topics

Basics

Management of the Extended Interrupt System........236

HowTos

How to Program PHS-Bus Interrupts........240

References

Error Codes of PHS-Bus Interrupt Functions........254

alloc_phs_int_line

Syntax

```
int alloc_phs_int_line(phs_addr_t base)
```

Include file

`phsint.h`

Purpose

To allocate PHS-bus interrupt lines for nonstandard I/O boards.

Description

In particular, the function must be called for customer I/O boards that need specialized code for programming the PHS-bus interrupt line number. It is assumed that the user provides and executes the initialization code after calling `alloc_phs_int_line`, but before installing interrupt handlers for the board. The initialization will normally include the setting of the interrupt line number in a board-specific setup register. The allocated interrupt line number is therefore passed to the caller as the return value of `alloc_phs_int_line`.

Note

This function must be called before installing interrupt handlers for the specified board. Otherwise the function will issue an error message (error code `PHSINT_FUNC_NOT_ALLOWED`) and terminate the application.

Parameters	**base** Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.
Return value	Valid PHS-bus interrupt line number if successful.

Related topics

Basics

HowTos

References

free_phs_int_line

Syntax

```
int free_phs_int_line(phs_addr_t base)
```

Include file

`phsint.h`

Purpose

To free PHS-bus interrupt lines for nonstandard I/O boards.

Description

This is the counterpart of **`alloc_phs_int_line`**. It releases a PHS-bus interrupt line that was previously allocated for an I/O board. It is assumed that the user provides and executes the code for resetting the interrupt line number of the board to the default value after calling **`free_phs_int_line`**. The default interrupt line number is therefore passed to the caller as the return value of **`free_phs_int_line`**. Remember that the default interrupt line is reserved for boards operated in polling mode.

> **Note**
>
> This function must not be called until all interrupt handlers for the specified board have been uninstalled. Otherwise it will issue an error message (error code `PHSINT_FUNC_NOT_ALLOWED`) and terminate the application.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

Return value

Default interrupt line number (0).

Related topics

Basics

HowTos

References

enable_phs_int

Syntax

```
enable_phs_int(
    phs_addr_t base,
    int n)
```

Include file

`phsint.h`

Purpose

To separately enable a single I/O interrupt.

Description

The corresponding interrupt handler must have been installed beforehand.

> **Note**
>
> As this macro is assumed to be executed under real-time conditions, the interrupt mask register of the interrupt controller on the specified I/O board is modified without error checking.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

n Specifies the slave interrupt number. Valid numbers are defined by the mapping of interrupt sources to inputs 0 … 7 of the corresponding slave interrupt controller. Further information can be found in the *PHS Bus System Hardware Reference* and in the *RTLib Reference* of your I/O board.

Related topics

Basics

HowTos

References

disable_phs_int

Syntax

```
disable_phs_int(
   phs_addr_t base,
   int n)
```

Include file

`phsint.h`

Purpose

To separately disable a single I/O interrupt.

Description

The corresponding interrupt handler must have been installed before.

> **Note**
>
> As this macro is assumed to be executed under real-time conditions, the interrupt mask register of the interrupt controller on the specified I/O board is modified without error checking.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

n Specifies the slave interrupt number. Valid numbers are defined by the mapping of interrupt sources to inputs 0 … 7 of the corresponding slave interrupt controller. Further information can be found in the *PHS Bus System Hardware Reference* and in the *RTLib Reference* of your I/O board.

Related topics

Basics

HowTos

References

init_phs_int

Syntax

```
init_phs_int(
   phs_addr_t base,
   int n)
```

Include file

`phsint.h`

Purpose

To initialize a slave interrupt controller without installing an interrupt handler separately.

Description

The specified I/O interrupt source is enabled at the slave interrupt controller. However, PHS-bus interrupts are not enabled generally. If the interrupt is the first one that is enabled for the indicated I/O board, the board's interrupt controller is connected to a free PHS-bus interrupt line. This does not hold for customer I/O boards (such as the DS4201 Prototyping Board and its derivatives) which use static interrupt lines or customer-specific setup registers. For these boards, `declare_phs_int_line` or `alloc_phs_int_line` must be called before calling `init_phs_int`.

> **Note**
>
> The PHS-bus interrupt can be deinitialized with `deinit_phs_int`. `deinstall_phs_int_vector` must not be used for this purpose.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

n Specifies the slave interrupt number. Valid numbers are defined by the mapping of interrupt sources to inputs 0 … 7 of the corresponding slave interrupt controller. Further information can be found in the *PHS Bus System Hardware Reference* 📖 and in the *RTLib Reference* of your I/O board.

Related topics

Basics

HowTos

References

deinit_phs_int

Syntax

```
deinit_phs_int(
   phs_addr_t base,
   int n)
```

Include file

`phsint.h`

Purpose

To deinitialize a slave interrupt controller without uninstalling an interrupt handler separately.

Description

It is the counterpart of `init_phs_int`. The specified I/O interrupt source is disabled at the slave interrupt controller. If the interrupt is the last one that is disabled for the specified I/O board, the interrupt controller of the board is reconnected to the default PHS-bus interrupt line (0), which is reserved for boards operated in polling mode. However, under no circumstances are PHS-bus interrupts disabled in general.

> **Note**
>
> This function must not be used to deinitialize a PHS-bus interrupt initialized with `install_phs_int_vector`. Use `deinstall_phs_int_vector` for this purpose.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

n Specifies the slave interrupt number. Valid numbers are defined by the mapping of interrupt sources to inputs 0 … 7 of the corresponding slave interrupt controller. Further information can be found in the *PHS Bus System Hardware Reference* and in the *RTLib Reference* of your I/O board.

Related topics

Basics

HowTos

References

set_phs_int_mask

Syntax

```
set_phs_int_mask(
   phs_addr_t base,
   int mask)
```

Include file `phsint.h`

Purpose To set the interrupt mask of a slave interrupt controller.

Description The interrupt controller must have been initialized beforehand. The constants listed in the table below have been defined to simplify the usage of this macro.

> **Note**
>
> As this macro is assumed to be executed under real-time conditions, the mask register of the interrupt controller on the specified I/O board is modified without error checking.

Parameters

base Specifies the PHS-bus base address of the I/O board. Use the macro as described in I/O board base address on page 222.

mask Specifies the bit pattern to be programmed into the interrupt mask register. A `"1"` bit disables the corresponding slave interrupt, while a `"0"` bit enables it. The following table shows the slave interrupt controller mask bits (defined in `phsint.h`).

Error Code	Value	Meaning
`PHS_MASK_SLAVE_INT0`	0x01	Bit position of slave interrupt 0
`PHS_MASK_SLAVE_INT1`	0x02	Bit position of slave interrupt 1
`PHS_MASK_SLAVE_INT2`	0x04	Bit position of slave interrupt 2
`PHS_MASK_SLAVE_INT3`	0x08	Bit position of slave interrupt 3
`PHS_MASK_SLAVE_INT4`	0x10	Bit position of slave interrupt 4
`PHS_MASK_SLAVE_INT5`	0x20	Bit position of slave interrupt 5

Error Code	Value	Meaning
PHS_MASK_SLAVE_INT6	0x40	Bit position of slave interrupt 6
PHS_MASK_SLAVE_INT7	0x80	Bit position of slave interrupt 7

Related topics

Basics

Management of the Extended Interrupt System........236

HowTos

How to Program PHS-Bus Interrupts........240

References

Error Codes of PHS-Bus Interrupt Functions........254

Troubleshooting for PHS-Bus Interrupt Handling

Error Codes of PHS-Bus Interrupt Functions

Error codes

There are a few errors where program execution may be continued. In these cases, a warning message is generated and the function returns the appropriate error code. The following table shows the error codes of the PHS-bus interrupt functions (defined in `phsint.h`).

Error Code	Meaning	Message	Action
PHSINT_NO_ERROR	No error	None	Return
PHSINT_NO_BOARD	No board detected at the specified PHS-bus base address	Error	Exit
PHSINT_UNKNOWN_BOARD	Unknown board identification number detected	Error	Exit
PHSINT_BOARD_UNINITIALIZED	The board to be accessed has not been initialized	Error	Exit
PHSINT_NO_INT_LINE	No free PHS-bus interrupt line available	Error	Exit
PHSINT_INVALID_BASE	Invalid board base address specified	Error	Exit
PHSINT_INVALID_SLAVE_NUM	Invalid slave interrupt number specified	Error	Exit
PHSINT_ICU_NOT_ENABLED	Slave interrupt controller has no interrupts enabled	Warning	Return
PHSINT_SLAVE_NOT_INST	Specified slave interrupt is not installed	Warning	Return
PHSINT_VECTOR_IN_USE	Slave interrupt vector is already in use	Error	Exit

Error Code	Meaning	Message	Action
PHSINT_INVALID_LINE_NUM	Invalid PHS-bus interrupt line number specified	Error	Exit
PHSINT_LINE_IN_USE	Specified PHS-bus interrupt line is already in use	Error	Exit
PHSINT_NO_STATIC_INT_LINE	Board at specified base address does not use a static PHS-bus interrupt line	Warning	Return
PHSINT_FUNC_NOT_ALLOWED	Function call is not allowed because slave interrupts are enabled	Error	Exit
PHSINT_NO_PIC	Board at specified base address does not contain a slave interrupt controller	Error	Exit
PHSINT_INCONSISTENT_LINES	Inconsistent line numbers declared for the board at specified base address	Error	Exit
PHSINT_NO_ISR_INSTALLED	No handler is installed for the triggered PHS-bus interrupt.	Warning	Return (from interrupt)

Multiprocessing Modules

Introduction

A DS1007 modular system provides the multicore (MC) and multiprocessor (MP) feature. You use the same functions for MC and MP systems.

Where to go from here

Information in this section

Initialization

Introduction

This chapter contains the functions to initialize a multiprocessor system (MP system).

Note

The multiprocessor features are supported by the DS1007 PPC Processor Board as multicore system (dual-core) or as multiprocessor system (two or more hardware platforms connected via Gigalink modules).

Where to go from here

Information in this section

Data Types for MP System Initialization

Data types

The following data types are defined:

mp_target_type

```
typedef struct {
   Int32 cpu_no;  /* CPU number */
   Int32 gl_no;   /* Gigalink number */
}mp_target_type;
```

mp_topology_type

```
typedef struct {
   mp_target_type target[4];
}mp_topology_type;
```

mp_cpu_available_type

```
UInt32 mp_cpu_available_type;
```

Every MP application has a global or local variable of type `mp_topology_type`, which is passed to **`dsgl_mp_init`**. This variable contains a `4` by `n` matrix, where `n` is the number of processors in the system. Hence, each element of the matrix describes one Gigalink in the MP system.

The elements (source Gigalink) are of type `mp_target_type`, which consists of the CPU and Gigalink number of the target Gigalink, to which the source Gigalink is connected. The matrix is redundant, as two connected Gigalinks form a matching pair in the matrix. The function **`dsgl_mp_init`** checks the consistency of the matrix. The #defines CPU_x (x = NONE, 0 … 15) help to specify the CPU number, the #defines GL_x (x = NONE, 0 … 3) help to specify the Gigalink number.

The elements of the array that specifies the available CPUs are of type `mp_cpu_available_type`. This array is passed to **`dsgl_mp_optional_cpu_reduce`** as parameter.

Global variables

The following global variables are defined:

rtlib_cpu_id ID of the local CPU. On default (single processor system) the local CPU is assigned ID 0. The CPU with ID 0 automatically is the master CPU in a MP system (e.g., for the DSTS module).

rtlib_num_cpus Number of CPUs in the system, specified by **`dsgl_mp_init`**.

dsgl_mp_init

Syntax

```
void dsgl_mp_init(
      int num_cpus,
      int cpu_id,
      mp_topology_typemp_topology,
      double mat_period,
      double timeout,
      int num_retries)
```

Include file

`dsgl_mp.h`

Purpose

To initialize all MP system relevant software modules.

Description

This function has to be carried out after the board initialization function `RTLIB_INIT()`. It performs the following setups:

- It assigns the local CPU (`rtlib_cpu_id`) the ID `cpu_id` and sets the number of CPUs (`rtlib_num_cpus`) to `num_cpus`.
- It checks all Gigalink connections that are specified for the local CPU in the topology matrix `mp_topology` and establishes them.
- It sets up the STBU (Synchronous Time Base Unit) by calling the time stamping initialization routine `ts_init`. The STBU is stopped. The processor with ID 0 is defined as time base master, which generates the system macrotick with a period of `mat_period`. The macrotick event is routed through the system. The Gigalinks of the slave processors (IDs from 1 to `num_cpus-1`) are configured for properly receiving the macrotick event.
- It initializes the MP trigger dispatching mechanism, which routes trigger events through the system. These triggers are needed for time stamping and distributed tracing.

The function tries to initialize the MP system until the `timeout` elapses. In this case an info message is issued and the initialization is retried. The maximum number of retries is specified by `num_retries`. When the maximum number of retries is reached and the initialization is still unsuccessful, an error message is issued. The further behavior depends on the sign of `num_retries`. If it is positive, the application is terminated. If it is negative, the application continues.

Note

The multiprocessor features are supported by the DS1007 PPC Processor Board as multicore system (dual-core) or as multiprocessor system (two or more hardware platforms connected via Gigalink modules).

Parameters

num_cpus Specifies the number of CPUs in the multiprocessor system.

cpu_id Specifies the ID of the local CPU within the range 0 … (`num_cpus`-1).

mp_topology Specifies the topology matrix that stores all Gigalink connections of the multiprocessor system. Each CPU has one row (index: CPU ID) and each Gigalink has one column (index: 0 … 3). An element at position (m, n) specifies the target of Gigalink n at CPU m. The target itself is a pair of (k, l), where k is the target CPU and l the target Gigalink.

mat_period Specifies the period of the multiprocessor system macrotick counter in seconds. Due to the dependency between `mat_period` and the time range and accuracy, we recommend a value of 1 … 10 ms (see Time-Stamping on page 44).

timeout Specifies the timeout in seconds.

num_retries Specifies the maximum number of retries. This value can be positive or negative. When a positive number of retries is used the application is terminated after all retries were unsuccessful. A negative number of retries lets the processor continue after the retries. In both cases an information message is issued after each retry. When the number of retries is set to `DS1007_INIT_INF_RETRIES` the function retries forever.

Return value

None

Related topics

References

dsgl_mp_synchronize

Syntax

```
void dsgl_mp_synchronize(void)
```

Include file

`dsgl_mp.h`

Purpose

To perform a synchronized start of an MP system.

> **Note**
>
> The multiprocessor features are supported by the DS1007 PPC Processor Board as multicore system (dual-core) or as multiprocessor system (two or more hardware platforms connected via Gigalink modules).

> **Note**
>
> - Use this function only during the initialization of your model or in the model background because the function also resets the Time Stamping module and stops the STBU. Use `ts_reset` to start the STBU again.
> - You have to initialize your MP system by calling **`dsgl_mp_init`** before using this function.

Description

The synchronization is performed in three steps:

- In the first step the status word dispatching mechanism is used to send a synchronization request bit from the master CPU (ID = 0) to all slave CPUs.
- The slave receive the request and acknowledge it by setting the appropriate bit in their status words, which are dispatched to the master CPU.

- When the master finds all acknowledge bits in the slave status words set, it signals the system start by sending a MAT interrupt. The slave processors poll the MAT interrupt line and exit this function when they find the interrupt set. So, there is only a small time jitter within all processors exit the synchronization function.

The status word dispatching mechanism and the MAT interrupt is initialized by **`dsgl_mp_init`**. When this function is not called all processors consider themselves as single processor system and omit synchronization.

Return value

None

Related topics

References

dsgl_mp_init 259
Time-Stamping 44

dsgl_mp_route_mat

Syntax

```
void dsgl_mp_route_mat(void)
```

Include file

`dsgl_mp.h`

Purpose

To route the macrotick interrupt through the system.

Description

This function is called by **`dsgl_mp_init`**. For more information on the macrotick interrupt, refer to Time-Stamping on page 44.

Return value

None

Related topics

References

dsgl_mp_init 259

dsgl_mp_optional_cpu_reduce

Syntax

```
void dsgl_mp_optional_cpu_reduce(
   int num_cpus,
   mp_topology_type *mp_topology,
   mp_cpu_available_type *mp_cpu_available)
```

Include file

`dsgl_mp.h`

Purpose

To clear the `mp_topology` matrix from not present members in a MP system.

Description

This function removes entries of not present CPUs from the `mp_topology` matrix. If a required CPU is not present, the program exits with an error message, if an optional CPU is not present, the entry in the `mp_cpu_available` array is changed to CPU_ABSENT.

Note

The multiprocessor features are supported by the DS1007 PPC Processor Board as multicore system (dual-core) or as multiprocessor system (two or more hardware platforms connected via Gigalink modules).

Parameters

num_cpus Specifies the number of CPUs in the multiprocessor system.

mp_topology Specifies the pointer to the topology matrix that stores all Gigalink connections of the multiprocessor system. Each CPU has one row (index: CPU ID) and each Gigalink has one column (index: 0 to 3). An element at position (m, n) specifies the target of Gigalink n at CPU m. The target itself is a pair of (k, l), where k is the target CPU and l the target Gigalink.

mp_cpu_available Specifies the pointer to the array of required and optional CPUs. The following symbols are predefined:

Predefined Symbol	Meaning
CPU_ABSENT	Optional member is not present
CPU_OPTIONAL	Optional member
CPU_REQUIRED	Required member

Return value

None

Example

```
#define NUM_CPUS 3
mp_topology_type mp_topology[NUM_CPUS] =
{ /* CPU_0 */ {{ {CPU_1, GL_0}, {CPU_2, GL_0},
                 {CPU_NONE, GL_NONE}, {CPU_NONE, GL_NONE} }},
  /* CPU_1 */ {{ {CPU_0, GL_0}, {CPU_2, GL_1},
                 {CPU_NONE, GL_NONE}, {CPU_NONE, GL_NONE} }},
};
mp_cpu_available_type mp_cpus[NUM_CPUS] =
   {CPU_REQUIRED, CPU_OPTIONAL};
…
RTLIB_INIT();
dsgl_mp_optional_cpu_reduce(NUM_CPUS, mp_topology, mp_cpus);
dsgl_mp_init(NUM_CPUS, CPU_0, mp_topology, 0.001, 5.0, 3);
…
```

Global Sample Rate Timer in an MP System

Introduction

This chapter contains the functions to initialize a global sampling rate timer in a multiprocessor system (MP system) or a multicore system (MC system).

Where to go from here

Information in this section

Example of Initializing a Global Sample Rate Timer

Introduction

This example shows an initialization of a DS1007 processor board in a three multiprocessor system.

Example

```
/* MP initialization of cpu no. 1 of a 3 processor system */
/* Connections: CPU0/GL0 <-> CPU1/GL0, CPU1/GL1 <-> CPU2/GL0 */
#include <brtenv.h>
#define NUM_CPUS 3
#define CPU_ID 0
volatile mp_topology_type mp_topology[NUM_CPUS]=
{
 {{{CPU_1, GL_0}, {CPU_NONE, GL_NONE}, {CPU_NONE, GL_NONE},
   {CPU_NONE, GL_NONE}}}
 {{{CPU_0, GL_0}, {CPU_2, GL_0}, {CPU_NONE, GL_NONE},
   {CPU_NONE, GL_NONE}}}
 {{{CPU_1, GL_1}, {CPU_NONE, GL_NONE}, {CPU_NONE, GL_NONE},
   {CPU_NONE, GL_NONE}}}
};
```

```
void global_isr()   /* interrupt service routine */
{
   dsgl_mp_begin_isr_global_srt();
   /* do something */
   dsgl_mp_end_isr_global_srt();
}
void main(void)
{
   /* initialize single-processor modules */
   init on page 19();
   /* initialize the multiprocessor system */
   dsgl_mp_init(NUM_CPUS, CPU_ID, mp_topology,
   0.001, 5.0, 5);
   /* synchronize cpus */
   dsgl_mp_synchronize();
   /* define start time (t=0) for the simulation */
   ts_init();
   /* start global sampling rate timer */
   dsgl_mp_start_isr_global_srt(0.001, global_isr);
   /* enter background loop */
   while(1)
   {
      RTLIB_BACKGROUND_SERVICE();
   }
}
```

dsgl_mp_global_srt_init

Syntax

```
void dsgl_mp_global_srt_init(
      double period,
      DSGL_MP_INT_HANDLER_TYPE timer_isr)
```

Include file

`dsgl_mp.h`

Purpose

To initialize a global sampling rate timer by routing the timer A interrupt through the system and to install an interrupt service routine (ISR).

Description

This function initializes a global sampling rate timer by routing the timer A interrupt from the master processor (cpu id = 0) through the system and installs the ISR. The function has to be called on every processor core in an MP system to route the interrupt properly.

The function performs the following actions on the master and the slave processors:

When the function is called at the master processor, all the Gigalinks which have slave processors connected to them are configured to send the timer A interrupt (Gigalink interrupt 0). Then the interrupt service routine `timer_isr` is installed at the timer A interrupt vector. The timer is programmed with period `period` and then started. Finally, the interrupt is enabled.

When the function is called at a slave processor, the Gigalink which is directed to the master processor is configured to receive Gigalink interrupt 0. The interrupt service routine `timer_isr` is installed at Gigalink interrupt 0. Gigalinks directed to further slave processors are configured to send interrupt 0. Interrupt 0 is enabled at the master Gigalink.

The function unmasks the timer/Gigalink interrupt, but does not enable interrupts globally. To do both, use **`dsgl_mp_start_isr_global_srt`** on page 267.

Note

- Initialize the multiprocessor system by calling **`dsgl_mp_init`** before calling this function. Otherwise the function behaves like **`srtk_start_isr_timerA`** and all Gigalink actions are omitted.

Parameters

period Specifies the sampling rate timer period in seconds.

timer_isr Specifies the name of the interrupt service routine.

Return value

None

Related topics

References

dsgl_mp_start_isr_global_srt

Syntax

```
void dsgl_mp_start_isr_global_srt(
      Float64 period,
      DSGL_MP_Int_Handler_Type isr_function_name)
```

Include file

`dsgl_mp.h`

Purpose

To initialize a global sampling rate timer and install an interrupt service routine (ISR) for the corresponding interrupt.

Description

A global sampling rate timer is used for synchronously triggering ISRs at every processor in an MP system or every core in an MC system. The interrupt is generated by the master processor or master core and dispatched by virtual Gigalinks to all other processors or cores.

When the function is called at the master processor or master core, Timer A is initialized with the period `period` and the ISR `isr_function_name` is installed at its interrupt vector. Subsequently the Timer A interrupt is routed to the Gigalink module. The interrupt is dispatched as Gigalink hardware subinterrupt 0.

When the function is called at a slave processor or slave core, the ISR `isr_function_name` is installed at the Gigalink subinterrupt vector 0. The interrupt is then routed further to other Gigalinks.

Note

- When a global sampling rate timer is desired, this function must be called on every processor or core in the multiprocessor system for proper initialization.
- Initialize the multiprocessor system by calling **`dsgl_mp_init`** before calling this function. Otherwise the function behaves like **`srtk_start_isr_timerA`**.

Parameters

period Specifies the sampling rate timer period in seconds.

isr_function_name Specifies the name of the interrupt service routine. This function must not have an input parameter or a return value, meaning, `void isr_function_name(void)`. You can implement an overrun check within your interrupt service routine with the functions **`dsgl_mp_begin_isr_global_srt`** and **`dsgl_mp_end_isr_global_srt`**.

Return value

None

Related topics

References

dsgl_mp_begin_isr_global_srt 268
dsgl_mp_end_isr_global_srt 269

dsgl_mp_begin_isr_global_srt

Syntax

```
void dsgl_mp_begin_isr_global_srt(void)
```

Include file	`dsgl_mp.h`
Purpose	To implement an overrun check mechanism for the global sampling rate timer together with **`dsgl_mp_end_isr_global_srt`**.
Description	Use both functions in the interrupt service routine of the global sampling rate timer. The code enclosed by them is executed with enabled interrupts. When the interrupt reoccurs, an overrun error message is issued and the interrupt is disabled.
Return value	None
Related topics	References

dsgl_mp_end_isr_global_srt........269
dsgl_mp_start_isr_global_srt........267

dsgl_mp_end_isr_global_srt

Syntax	`void dsgl_mp_end_isr_global_srt(void)`
Include file	`dsgl_mp.h`
Purpose	To implement an overrun check mechanism for the global sampling rate timer together with **`dsgl_mp_begin_isr_global_srt`**.
Description	Use both functions in your interrupt service routine of the global sampling rate timer. The code enclosed by them is executed with enabled interrupts. When the interrupt reoccurs an overrun error message is issued and the interrupt is disabled.
Return value	None

Related topics

References

dsgl_mp_begin_isr_global_srt 268
dsgl_mp_start_isr_global_srt 267

Interprocessor Interrupts

Introduction

This chapter contains the functions that can be used to configure interrupt transmission between multiple processors by using the Gigalink modules and/or multiple cores by using the virtual internal Gigalinks.

> **Note**
>
> The multiprocessor features are supported by the DS1007 PPC Processor Board as multicore system (dual-core) or as multiprocessor system (two or more hardware platforms connected via Gigalink modules).

Where to go from here

Information in this section

dsgl_ipi_init

Syntax

```
int dsgl_ipi_init(void)
```

Include file

`dsgl_ipi.h`

Purpose

To register the interprocessor interrupt module at the VCM (version and config section management) module and initialize the interprocessor subinterrupt handling.

Description

This function is called by **`dsgl_mp_init`**.

Return value

This function returns one of the following values:

Value	Meaning
0	Gigalink module not present
1	Gigalink module present

Related topics

References

dsgl_ipi_configure

Syntax

```
int dsgl_ipi_configure(
      int gl_no,
      int line,
      int config)
```

Include file

`dsgl_ipi.h`

Purpose

To set the interrupt source for an outgoing interrupt line at the specified Gigalink.

Description

Use the return value to determine whether this outgoing interrupt line is already being used by a different source than the software interrupt.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

line Specifies the outgoing interrupt line number within the range 0 … 12.

config Specifies the interrupt line configuration. The following symbols are predefined:

Predefined Symbol	Meaning
IPI_CFG_SOFTWARE	Software interrupt only
IPI_CFG_HARDWARE	Hardware and software interrupts
IPI_CFG_GIGALINK	Gigalink and software interrupts

Return value

This function returns the old interrupt configuration. The predefined symbols are shown in the table above.

Example

The following example configures Gigalink 0 Line 1 to be driven by Timer B (hardware source). If the line was already being used an error is issued.

```
if (dsgl_ipi_configure(0, 1, IPI_CFG_HARDWARE) !=
      IPI_CFG_SOFTWARE)
{
  msg_error_set(MSG_SM_USER, 0,
      "Gigalink 0 Line 1 was already used.");
}
```

Related topics

References

dsgl_ipi_sint_max_snd_set...280

dsgl_ipi_interrupt

Syntax

```
void dsgl_ipi_interrupt(
      int gl_no,
      int int_no)
```

Include file

`dsgl_ipi.h`

Purpose

To trigger an interrupt for an outgoing interrupt line or a software dispatched subinterrupt.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

int_no Specifies the outgoing interrupt line number within the range 0 … 12 or software dispatched subinterrupt within the range 16 … (15+`max_ints`). The variable `max_ints` is specified by the function **`dsgl_ipi_sint_max_rcv_set`**.

Return value

None

Related topics

References

dsgl_ipi_acknowledge

Syntax

```
void dsgl_ipi_acknowledge(
      int gl_no,
      int line)
```

Include file

`dsgl_ipi.h`

Purpose

To acknowledge a single incoming Gigalink interrupt.

Description

The real-time software acknowledges an incoming Gigalink interrupt. Call this function only when you want to delete a disabled interrupt before it is enabled.

Note

This function must not be called for software dispatched subinterrupts.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

line Specifies the input interrupt line number within the range 0 … 12.

Return value

None

dsgl_ipi_enable

Syntax

```
void dsgl_ipi_enable(
     int gl_no,
     int line)
```

Include file

`dsgl_ipi.h`

Purpose

To enable a single incoming interrupt line of the interrupt control unit.

> **Note**
>
> This function must not be called for software dispatched subinterrupts.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

line Specifies the input interrupt line number within the range 0 … 12.

Related topics

References

dsgl_ipi_disable...276
dsgl_ipi_enable_bm...277

dsgl_ipi_disable

Syntax

```
void dsgl_ipi_disable(
     int gl_no,
     int line)
```

Include file

`dsgl_ipi.h`

Purpose

To disable a single incoming interrupt line in the interrupt control unit.

> **Note**
>
> This function must not be called for software dispatched subinterrupts.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

line Specifies the input interrupt line number within the range 0 … 12.

Return value

None

Related topics

References

dsgl_ipi_enable_bm

Syntax

```
void dsgl_ipi_enable_bm(
      int gl_no,
      UInt32 bitmask)
```

Include file

`dsgl_ipi.h`

Purpose

To enable several incoming interrupt lines of the interrupt control unit.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

bitmask Specifies the interrupt lines to be enabled. Each incoming interrupt line corresponds with one bit in the bitmask (Bit 0 sets interrupt line 0, Bit 1 sets interrupt line 1, etc.). Set for each interrupt line that is to be enabled the corresponding bit to 1.

Return value

None

Related topics

References

dsgl_ipi_disable_bm

Syntax

```
void dsgl_ipi_disable_bm(
      int gl_no,
      UInt32 bitmask)
```

Include file

`dsgl_ipi.h`

Purpose

To disable several incoming interrupt lines in the interrupt control unit.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

bitmask Specifies the interrupt lines to be disabled. Each incoming interrupt line corresponds with one bit in the bitmask (Bit 0 sets interrupt line 0, Bit 1 sets interrupt line 1, etc.). Set for each interrupt line that is to be disabled the corresponding bit to 1.

Return value

None

Related topics

References

dsgl_ipi_disable 276
dsgl_ipi_enable_bm 277

dsgl_ipi_mask_set

Syntax

```
void dsgl_ipi_mask_set(
      int gl_no,
      UInt32 mask)
```

Include file

`dsgl_ipi.h`

Purpose

To set the interrupt mask of the interrupt control unit.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

mask Specifies the input interrupt mask to mask the incoming interrupts. A set bit disables an incoming interrupt, a clear bit enables it. Bit 0 affects interrupt 0, bit 1 interrupt 1, etc.

Return value

None

Related topics

References

dsgl_ipi_mask_get........279

dsgl_ipi_mask_get

Syntax

```
UInt32 dsgl_ipi_mask_get(int gl_no)
```

Include file

`dsgl_ipi.h`

Purpose

To get the interrupt mask of the interrupt control unit.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

Return value

The current input interrupt mask. The interrupt mask masks the incoming interrupts. If a bit is set, the corresponding interrupt is disabled. If a bit is cleared, the corresponding interrupt is enabled. Bit 0 corresponds to interrupt 0, bit 1 to interrupt 1, etc.

Related topics

References

dsgl_ipi_mask_set........278

dsgl_ipi_sint_max_snd_set

Syntax

```
void dsgl_ipi_sint_max_snd_set(
      int gl_no,
      int max_ints)
```

Include file

`dsgl_ipi.h`

Purpose

To set the maximum number of software dispatched subinterrupts that can be sent from the given Gigalink.

Description

Choose a value as low as possible to decrease the time needed to transfer the subinterrupt information.

> **Note**
>
> On the receiving Gigalink the number of software dispatched subinterrupts has to be set to the same value with the function **`dsgl_ipi_sint_max_rcv_set`**.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

max_ints Specifies the maximal number of software dispatched subinterrupts that can be sent.

Return value

None

Related topics

References

dsgl_ipi_sint_max_rcv_set.. 280

dsgl_ipi_sint_max_rcv_set

Syntax

```
void dsgl_ipi_sint_max_rcv_set(
      int gl_no,
      int max_ints)
```

Include file

`dsgl_ipi.h`

Purpose

To set the maximal number of software dispatched subinterrupts that can be received by the given Gigalink.

Description

Choose a value as low as possible to decrease the time needed to transfer the subinterrupt information.

> **Note**
>
> On the sending Gigalink the number of software dispatched subinterrupts has to be set to the same value with the function **`dsgl_ipi_sint_max_snd_set`**.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

max_ints Specifies the maximal number of software dispatched subinterrupts that can be sent.

Return value

None

Related topics

References

dsgl_ipi_sint_max_snd_set 280

dsgl_ipi_install_handler

Syntax

```
ipi_handler_type dsgl_ipi_install_handler(
      int gl_no,
      int int_no,
      ipi_handler_type handler)
```

Include file

`dsgl_ipi.h`

Purpose

To install an interrupt service routine for an outgoing interrupt line or a software dispatched subinterrupt.

> **Note**
>
> This function cannot be used together with the RT Kernel or RTI.

Parameters

gl_no Specifies the Gigalink number within the range 0 … 3.

int_no Specifies the outgoing interrupt line number within the range 0 … 12 or software dispatched subinterrupt within the range 16 … (15+`max_ints`). The variable `max_ints` is specified by the function **`dsgl_ipi_sint_max_rcv_set`**.

handler Specifies the address of the interrupt service routine. This function must not have an input parameter or a return value, meaning, `void ipi_handler_type(void)`.

Return value

This function returns the address of the previous interrupt service routine at the given interrupt number.

Gigalink Communication

Introduction

This chapter contains the functions for transmitting data between multiple processor boards when you use DS1006 or DS1007 boards or between multiple processor cores when you use a DS1006 or DS1007 board.

For the Gigalink communication the following data type is used.

gl_scantbl_entry_t

Gigalink scantable entry type

```
typedef struct
{
   UInt32  target_board_snr;   /* serial number of the board */
   UInt32  target_gl_no;       /* Gigalink number*/
   Int32   target_cpu_id;      /* CPU ID */
   UInt32  reserved;           /* reserved */
} gl_scantbl_entry_t;
```

Where to go from here

Information in this section

dsgl_init

Syntax

```
int dsgl_init()
```

Include file

`dsgl.h`

Purpose

To initialize all four Gigalink channels and clear the receiver buffers.

Description

This function is called by **`RTLIB_INIT`**.

Return value

This function returns one of the following values:

Value	Meaning
0	An error occurred during initialization
1	Initialization done

dsgl_initialized

Syntax

```
int dsgl_initialized(UInt32 gl_no)
```

Include file

`dsgl.h`

Purpose

To check whether a Gigalink connection has been initialized by the **dsgl_mp_init** function.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

Return value

This function returns one of the following values:

Value	Meaning
0	Gigalink connection is not initialized
1	Gigalink connection is initialized

dsgl_synchronized

Syntax

```
int dsgl_synchronized(
      UInt32 gl_no,
      gl_scantbl_entry_t *gl_st_ptr)
```

Include file

`dsgl.h`

Purpose

To check whether a Gigalink connection is synchronized with another board.

Description

This function has to be called in a loop for synchronizing the required Gigalink connections. If `gl_st_ptr` is specified, the serial number and the Gigalink number of the target board are stored.

The function is called by the **RTLIB_INIT** function.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...

Predefined Symbol	Meaning
GL_3	Gigalink number 3

gl_st_ptr Pointer to a structure `gl_scantbl_entry_t`, if the serial number and Gigalink number of target board are to be stored, or 0.

Return value

One of the following values:

Value	Meaning
1	Gigalink module is synchronized
0	Gigalink module is not synchronized

Example

```
gl_scantbl_entry_t gl_0_scan, gl_1_scan;
Int32 gl_0_synchronized = 0, gl_1_synchronized = 0;
RTLIB_INIT();
...
do
{
...
if (!gl_0_synchronized)
  gl_0_synchronized = dsgl_synchronized(GL_0,&gl_0_scan);
if (!gl_1_synchronized)
  gl_1_synchronized = dsgl_synchronized(GL_1,&gl_1_scan);
...
}
while(!(all_synchronized || timeout))
```

dsgl_scanner_init

Syntax

```
vcm_module_descriptor_type * dsgl_scanner_init(
      UInt32 parent_module)
```

Include file

`dsgl.h`

Purpose

To initialize a Gigalink scanner.

Description

This function initializes the Gigalink scanner and registers it at the VCM (version and config section management) module.

Parameters

parent_module Module ID of the parent module.

Return value

The function returns the pointer to the VCM entry of the Gigalink, or NULL pointer, if the Gigalink module is not present.

dsgl_scan

Syntax

```
int dsgl_scan(
      UInt32 gl_no,
      gl_scantbl_entry_t *gl_st_ptr)
```

Include file

`dsgl.h`

Purpose

To scan a Gigalink connection of a board.

Description

This function is used to scan the Gigalink connections of a board.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

gl_st_ptr Pointer to a structure `gl_scantbl_entry_t`, if the serial number and Gigalink number of target board are to be stored, or 0.

Return value

One of the following values:

Value	Meaning
0	No connection detected
1	Connection detected

dsgl_background_scan

Syntax

```
void dsgl_background_scan(void)
```

Include file	`dsgl.h`
Purpose	To scan unused Gigalinks for connections with other boards in the background of an application.
Description	The connections are stored in the additional config memory block of the Gigalink scanner module. This function is called by **`RTLIB_BACKGROUND_SERVICE`**.
Return value	None
Related topics	References

RTLIB_BACKGROUND_SERVICE........ 20

dsgl_write32

Syntax

```
void dsgl_write32(
      int gl_no,
      int channel_no,
      int offset,
      Int32 data)
```

Include file `dsgl.h`

Purpose To write a 32-bit data word to the write buffer of a receiver at a specified buffer offset.

Note

Do not exceed the maximum buffer size of 2KW. For performance reasons it is not checked by this function.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

data Specifies the 32-bit data word.

Return value

None

Related topics

References

dsgl_block_write.......... 293
dsgl_write32_and_switch.......... 290
dsgl_write64.......... 292

dsgl_write32_and_switch

Syntax

```
void dsgl_write32_and_switch(
      int gl_no,
      int channel_no,
      int offset,
      Int32 data)
```

Include file

`dsgl.h`

Purpose

To write a 32-bit data word to the write buffer of a receiver at a specified buffer offset and to switch the buffer.

> **Note**
>
> Do not exceed the maximum buffer size of 2KW. For performance reasons it is not checked by this function.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

data Specifies the 32-bit data word.

Return value

None

Related topics

References

dsgl_write32 289

dsgl_write64

Syntax

```
void dsgl_write64(
      int gl_no,
      int channel_no,
      int offset,
      Float64 data)
```

Include file

`dsgl.h`

Purpose

To write a 64-bit floating-point value to the write buffer of a receiver at a specified buffer offset.

> **Note**
>
> Do not exceed the maximum buffer size of 2KW. For performance reasons it is not checked by this function.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

data Specifies a 64-bit floating-point value.

Return value

None

Related topics

References

dsgl_block_write

Syntax

```
void dsgl_block_write(
      int gl_no,
      int channel_no,
      int offset,
      int count,
      const void *data,
      int buf_switch)
```

Include file

`dsgl.h`

Purpose

To write a data block from a source buffer to the write buffer of a receiver.

> **Note**
>
> Do not exceed the maximum buffer size of 2KW. For performance reasons it is not checked by this function.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode

Predefined Symbol	Meaning
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the sender buffer.

count Specifies the number of 32-bit words to be transmitted.

data Specifies the pointer to the data source buffer.

buf_switch Specifies the buffer switch by using one of the following constants:

Constant	Meaning
GL_BUF_SWITCH_ON	Switch buffer after write
GL_BUF_SWITCH_OFF	Do not switch buffer

This parameter is ignored if a channel in the virtual shared memory mode is used.

Return value

None

Related topics

References

dsgl_block_write64 294
dsgl_write32 289
dsgl_write32_and_switch 290

dsgl_block_write64

Syntax

```
void dsgl_write64(
      int gl_no,
      int channel_no,
      int offset,
      int count,
      const Float64 *data,
      int buf_switch)
```

Include file

`dsgl.h`

Purpose

To send a block of 64-bit floating-point values to a specific Gigalink channel.

Description

This function is a variant of the `dsgl_block_write` function, but performs byte order conversion for 64-bit floating-point data types. You have to convert the values if you transfer data between systems with a different byte order, for example, between DS1007 and DS1006 or DS1007 and SCALEXIO. When the internal Gigalink connections are used, the block-wise transfer achieves a higher data rate than multiple calls of `dsgl_write64`.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the sender buffer. It must be a multiple of 2.

count Specifies the number of 32-bit words to be transmitted. It must be a multiple of 2.

data Specifies the pointer to the data source buffer that must be 8 byte aligned.

buf_switch Specifies the buffer switch by using one of the following constants:

Predefined Symbol	Meaning
GL_BUF_SWITCH_ON	Switch buffer after write
GL_BUF_SWITCH_OFF	Do not switch buffer

This parameter is ignored if a channel in the virtual shared memory mode is used.

Return value

None

Related topics

References

dsgl_write_buffer_switch

Syntax

```
void dsgl_write_buffer_switch(
      int gl_no,
      int channel_no)
```

Include file

`dsgl.h`

Purpose

To send a write buffer switch command to switch the receiver write buffer.

Remarks

The buffer is not switched if a channel in the virtual shared memory mode is used.

> **Note**
>
> Since the last memory location in the receiver buffer is cleared, this location may not be used for any data.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode

Predefined Symbol	Meaning
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

Return value

None

dsgl_read32

Syntax

```
Int32 dsgl_read32(
      int gl_no,
      int channel_no,
      int offset)
```

Include file

`dsgl.h`

Purpose

To read a 32-bit word from a receiver read buffer.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode

Predefined Symbol	Meaning
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

Return value

This function returns the 32-bit value to be read.

Related topics

References

dsgl_read64

Syntax

```
Float64 dsgl_read64(
      int gl_no,
      int channel_no,
      int offset)
```

Include file

`dsgl.h`

Purpose

To read a 64-bit floating-point value from a receiver read buffer.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode

Predefined Symbol	Meaning
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

Return value

This function returns the 64-bit value to be read.

Related topics

References

dsgl_block_read 299
dsgl_read32 297

dsgl_block_read

Syntax

```
void dsgl_block_read(
      int gl_no,
      int channel_no,
      int offset,
      int count,
      void *data,
      int buf_switch)
```

Include file

`dsgl.h`

Purpose

To copy a data block from a receiver read buffer to a destination buffer.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

count Specifies the number of 32-bit words to be copied from the receiver buffer.

data Specifies the pointer to the destination buffer.

buf_switch Specifies the buffer switch by using one of the following constants:

Constant	Meaning
GL_BUF_SWITCH_ON	Switch buffer before read
GL_BUF_SWITCH_OFF	Do not switch buffer

This parameter is ignored if a channel in the virtual shared memory mode is used.

Return value

None

Related topics

References

dsgl_block_read64 ... 301
dsgl_read32 ... 297
dsgl_read64 ... 298

dsgl_block_read64

Syntax

```
void dsgl_block_read64(
      int gl_no,
      int channel_no,
      int offset,
      int count,
      Float64 *data,
      int buf_switch)
```

Include file

`dsgl.h`

Purpose

To receive a block of 64-bit floating-point values from a specific Gigalink channel.

Description

This function is a variant of the **`dsgl_block_read`** function, but performs byte order conversion for 64-bit floating-point data types. You have to convert the values if you transfer data between systems with a different byte order, for example, between DS1007 and DS1006 or DS1007 and SCALEXIO. When the internal Gigalink connections are used, the block-wise transfer achieves a higher data rate than multiple calls of **`dsgl_read64`**.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode

Predefined Symbol	Meaning
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

offset Specifies the word offset (32-bit) within the buffer.

count Specifies the number of 32-bit words to be copied from the receiver buffer.

data Specifies the pointer to the destination buffer.

buf_switch Specifies the buffer switch by using one of the following constants:

Predefined Symbol	Meaning
GL_BUF_SWITCH_ON	Switch buffer before read
GL_BUF_SWITCH_OFF	Do not switch buffer

This parameter is ignored if a channel in the virtual shared memory mode is used.

Return value

None

Related topics

References

dsgl_block_read 299
dsgl_block_write64 294
dsgl_read64 298

dsgl_read_buffer_switch

Syntax

```
void dsgl_read_buffer_switch(
      int gl_no,
      int channel_no)
```

Include file

`dsgl.h`

Purpose

To switch a receiver read buffer.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode

Return value

None

Related topics

References

dsgl_write_buffer_switch 296

dsgl_read_buffer_is_updated

Syntax

```
int dsgl_read_buffer_is_updated(
      int gl_no,
      int channel_no)
```

Include file

`dsgl.h`

Purpose

To check the status of the receiver read buffer.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

channel_no Specifies the channel number. Following symbols are predefined:

Predefined Symbol	Meaning
SBUF_CH_0	Channel 0 with swinging buffer mode
...	...
SBUF_CH_7	Channel 7 with swinging buffer mode
SMEM_CH_0	Channel 0 with virtual shared memory mode. This channel is already used as a service channel by RTLib and RTI.
SMEM_CH_1	Channel 1 with virtual shared memory mode
...	...
SMEM_CH_7	Channel 7 with virtual shared memory mode

Return value

This function returns the status of the receiver read buffer:

Value	Meaning
True	Receiver read buffer has been switched
False	Receiver read buffer has not been switched

In case of a shared memory channel (channel number 9 to 15) the return value is always true.

dsgl_module_present

Syntax

```
int dsgl_module_present(void)
```

Include file

`dsgl.h`

Purpose

To check whether an internal or external Gigalink module is present.

Return value

This function returns one of the following values:

Value	Meaning
0	Gigalink module is not present
1	Gigalink module is present

Note

If you are using a DS1007 PPC Processor Board, this function always returns 1, because of its internal virtual Gigalinks. Use **`dsgl_phys_module_present`** to check for an external Gigalink module. For further information, refer to DS1007 Multiprocessor Systems (DS1007 Features 📖).

dsgl_phys_module_present

Syntax

```
int dsgl_phys_module_present(void)
```

Include file

`dsgl.h`

Purpose

To check whether an external Gigalink module is present.

Return value

This function returns one of the following values:

Value	Meaning
0	Gigalink module is not present
1	Gigalink module is present

dsgl_opto_signal_detect

Syntax

```
int dsgl_opto_signal_detect(int gl_no)
```

Include file

`dsgl.h`

Purpose

To check if an optical signal at a Gigalink is detected.

Parameters

gl_no Specifies the Gigalink number. Following symbols are predefined:

Predefined Symbol	Meaning
GL_0	Gigalink number 0
...	...
GL_3	Gigalink number 3

Return value

This function returns one of the following values:

Value	Meaning
0	No optical signal detected
1	Optical signal detected

Host Programs

Introduction

There are some utilities installed on the host PC for building custom applications.

Where to go from here

Information in this section

Information in other sections

Firmware Manager Manual
Introduces you to the features provided by the Firmware Manager. It provides detailed information on the user interface, its command line options and instructions using the firmware management.

Host Settings

Introduction

This chapter describes the definitions, settings, files and libraries that are necessary to write your own C-coded programs for the PowerPC processor of DS1007 PPC Processor Board.

Where to go from here

Information in this section

Compiler and C Run-Time Libraries

Compiler and C run-time libraries

The compiler for building DS1007 applications is automatically installed when you install dSPACE software. The associated C run-time libraries are also used. The GNU compiler for QNX is installed in `<RCP_HIL_InstallationPath>\Compiler`. Further GNU tools, with the prefix `ntoppc` are installed for DS1007.

For information on the C++ support, refer to Integrating C++ Code on page 317.

Environment Variables and Paths

dSPACE command prompt

The dSPACE software installation does not set environment variables and other settings such as enhancements to the search path.

Use the Command Prompt for dSPACE RCP and HIL for the host tools. You find the command prompt as a shortcut in the Windows Start menu. The required paths and environment settings are then automatically set.

Folder Structure

Folder structure

The folder structure of the DS1007 software is as follows:

Folder	Contents
<RCP_HIL_InstallationPath>\DS1007\Lib	Library files of the DS1007
<RCP_HIL_InstallationPath>\DS1007\Include	Header files of the DS1007
<RCP_HIL_InstallationPath>\DS1007	Make files of the DS1007
<RCP_HIL_InstallationPath>\Win32	DS1007 related host tools and DLLs.
<RCP_HIL_InstallationPath>\Demos\DS1007	Demo examples
<RCP_HIL_InstallationPath>\Demos\DS1007MP	Demo examples for multiprocessing

DS1007 Real-Time Library

librt1007.so

All functions of the DS1007 Real-Time Library were compiled with the highest optimization level and collected in the library **librt1007.so**. Required objects from this library are dynamically linked to an application. This lets you update the RTLib1007 without having to rebuild the application. The header files are located in <RCP_HIL_InstallationPath>DS1007\Include.

Note

All necessary modules and header files are included by **Brtenv.h**.

In addition, the library **librt1007.so** contains the modules related to the supported I/O boards.

File Extensions

File extensions

The following file extensions are used:

File Extension	Meaning
.c[1)]	C source files
.cpp	C++ source files
.so	Dynamically linked libraries (shared objects)
.a	Statically linked libraries (archives)
.mk	Makefiles
.ppc	Executable programs for the PowerPC

File Extension	Meaning
`.rta`	Real-time application file, contains the PPC file and further required files, if available.

[1] For C++ support, refer to Integrating C++ Code on page 317.

Compiling, Linking and Downloading an Application

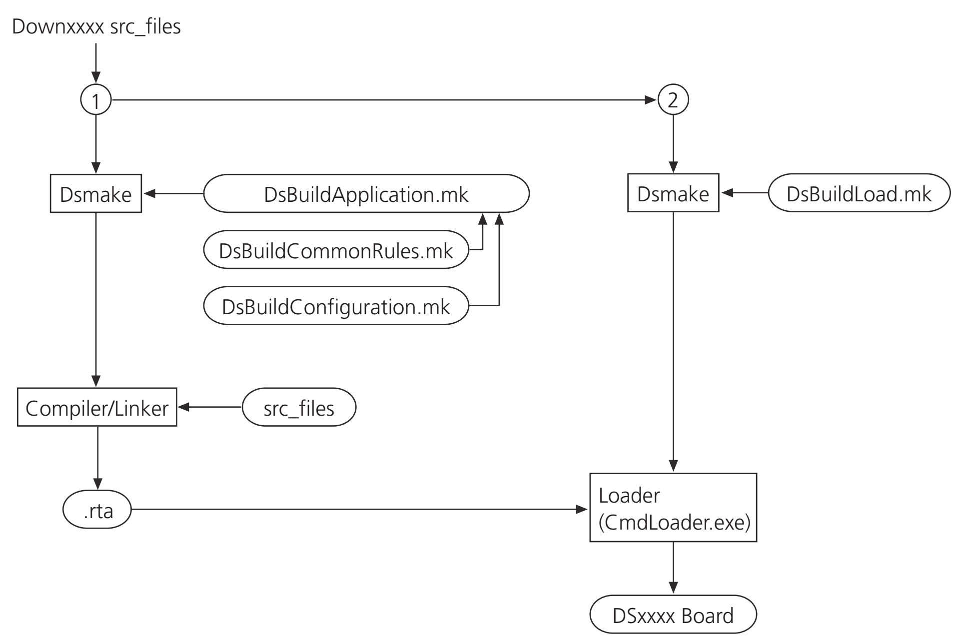
Introduction

If you want to build a user application and download it to the target hardware, you can use the `Down` tool for your board.

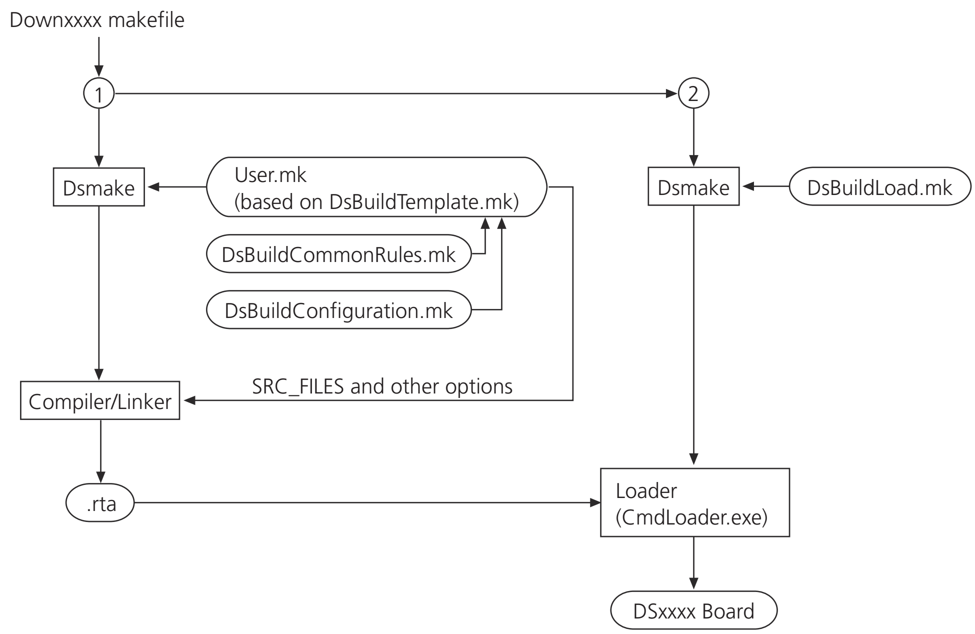
> **Tip**
>
> The executable file Down1007 can be called in a Command Prompt window (DOS window) of your host PC.
> If you use the Command Prompt for dSPACE RCP and HIL shortcut in the Windows Start menu, the required paths and environment settings are automatically set.

Process overview

The following schematic shows you the process overview for using Down with the source files as arguments. `DsBuildApplication.mk` is then used for the make process.



The following schematic shows you the process overview for using Down with the custom makefile as an argument. It is recommended to base the makefile on `DsBuildTemplate.mk`.



Where to go from here

Information in this section

Down1007.exe

Syntax

```
down1007 file.mk [options] [/?]
```

or

```
down1007 src_file(s) [options] [/?]
```

Purpose

To compile or assemble, link, and download handcoded applications.

Description

The following file types can be handled:

Local makefile (.mk) To compile, link, and download the application using the specified local makefile. Use the makefile DsBuildTemplate.mk as a template to write your own makefile. The resulting program file is named according to the name of the specified makefile.

C-coded source file (.c) To specify the file(s) to be compiled and linked using DsBuildApplication.mk. The resulting program file is named according to the name of the first specified source file.

C++-coded source file (.cpp) To specify the file(s) to be compiled and linked using DsBuildApplication.mk. The resulting program file is named according to the name of the first specified source file.

Note

If you use the Down tool with source files, the relocatable object files are deleted and the object file of the application is overwritten. If you call the Down tool with a user makefile as the argument, the object files remain unchanged until a modified source file requires recompilation.

If the file name extension is omitted, `Down1007` searches for existing files in the above order. If more than one source file is specified at the command line, the first file is treated as the main source file that names the complete application. The remaining source files are compiled or assembled, and linked to the application.

The built application is loaded by default to the DS1007 platform named 'ds1007'. The platform name is set by the dSPACE software, e.g., ControlDesk during platform registration. If you want to access another platform instead, you can specify the platform name using the /p option.

For a graphical process overview, refer to Compiling, Linking and Downloading an Application on page 311.

For further information on the C++ support, refer to Integrating C++ Code on page 317.

Options

The following command line options are available:

Option	Meaning
/c <NetworkName>	To specify the platform to be registered and used.
/co <option>	To specify additional compiler options; refer to the GNU Compiler documentation.
/d	To disable downloading the application; only compiling and linking.
/g	To compile for source level debugging;
/l	To write all output to `down1007.log`.

Option	Meaning
/lib <lib_file>	To specify an additional library to be linked.
/lko <option>	To specify a single additional linker option.
/lo <option>	To specify a single additional loader option.
/mo <option>	To specify a single additional DSMAKE option; call `dsmake -h` to get more information.
/n	To disable beep on error.
/p <PlatformName>	To specify a platform name that differs from the default. The default platform name is ds1007 if you call `Down1007.exe`.
/pause	To pause execution of `Down1007` before exit.
/x	To switch off code optimization.
/z	To download an existing object file without building.
/?	To display information.

Messages

The following messages are defined:

Message	Description
ERROR: not enough memory!	The attempt to allocate dynamic memory failed.
ERROR: environment variable PPC_ROOT not found! ERROR: environment variable X86_ROOT not found! ERROR: environment variable DSPACE_ROOT not found!	The respective environment variable is not defined in the DOS environment. The environment variables are set during the dSPACE software installation.
ERROR: can't load DLL '%DSPACE_ROOT %/exe/wbinfo.dll'! [number]	Loading the dynamic link library WBINFO.DLL failed. The number in brackets specifies the internal Windows error.
ERROR: can't read address of function 'GetWorkingBoardName()'! ERROR: can't read address of function 'GetWorkingBoardClient()'! ERROR: can't read address of function 'GetWorkingBoardConnection()'! ERROR: can't read address of function 'GetWorkingBoardType()'!	The address of the respective function could not be found in the dynamic link library WBINFO.DLL.
ERROR: can't read working board name! ERROR: can't read working board client! ERROR: can't read working board connection! ERROR: can't read working board type!	The respective working board information could not be read from the dspace.ini file. Register your hardware system using ControlDesk's Platform Manager.
WARNING: The working board type is DS???? instead of DS????! Accessing default board ds????.	The detected working board type is not responding to the DOWN1007 version. For example, if you are using DOWN1007 and the working board is of type DS1104, the board name ds1007 is used.
ERROR: unable to obtain full path of <file name>!	This error occurs if the full path name of the source file contains more than 260 characters, or if an invalid drive letter has been specified, for example, 1:\test.

Message	Description
ERROR: unable to access file <file name>!	The specified file cannot be accessed by Down1007. The file does not exist or another application is accessing it.
ERROR: source files must be available in the same directory!	All source files to be compiled must be available in the same application folder.
ERROR: make file <name> not allowed as additional source file!	Only assembly and C source files are allowed as additional source files.
ERROR: can't redirect stdout to file! ERROR: can't redirect stdout to screen!	The redirection of stdout to a file or to the screen has failed.
ERROR: can't invoke %DSPACE_ROOT %\exe\dsmake.exe: ...	Down1007 was not able to invoke DSMAKE.EXE successfully.
ERROR: making of <file name> failed! ERROR: building of <file name> failed!	An error occurred while executing a makefile, compiling or assembling a source file. See the screen output to get information about the reasons, for example, there can be programming errors in the source file.
ERROR: downloading of <file name> failed!	DOWN1007 was not able to download the application successfully. See dSPACE.log for more information.
ERROR: can't install exit handler!	The available memory space is too small for registering the exit handler.

Related topics

References

DsBuildApplication.mk

Description

This makefile is used to compile or assemble the application source files. It is called by Down1007.exe if no other makefile is specified. It uses the highest optimization level of the C compiler.

It includes:

- DsBuildCommonRules.mk
- DsBuildConfiguration.mk

Note

Do not edit.
Use the required option with Down1007 or a custom makefile based on DsBuildTemplate.mk instead.

Related topics

References

Down1007.exe 312
DsBuildTemplate.mk 316

DsBuildLoad.mk

Description

This file is automatically invoked by Down1007.exe to load the application to the target hardware after building, unless you use the /d option. It is also called if you use the /z option for download only.

Note

Do not edit or change this file.

Related topics

References

Down1007.exe 312

DsBuildTemplate.mk

Description

You can customize this makefile to match your individual requirements:

- CUSTOM_SRC_FILES

  You can add additional source files to be compiled by adding the names of the source files.
- CUSTOM_OBJ_FILES

  You can add additional object files to be linked to the application by adding the names of the object files.

- CUSTOM_LIB_FILES
  You can add additional libraries to be linked to the application by adding the names of the libraries.
- CUSTOM_C_OPTS
  You can add additional options for the C compiler.
- CUSTOM_ASM_OPTS
  You can add additional options for the assembler.
- CUSTOM_LK_OPTS
  You can add additional options for the linker.
- USER_BUILD_CPP_APPL
  You can enable the C++ support by setting this make macro to `ON`. For further information, refer to Integrating C++ Code on page 317.

Related topics

References

Down1007.exe..................312

Integrating C++ Code

Introduction

To integrate C++ code to your handcoded RTLib application, you have to enable the C++ support.

Adapting the user makefile

For adding C++ code to your application you have to adapt the `DsBuildTemplate.mk` file.

- Enable the C++ support
- Add C++ source files, C++ object files and C++ libraries

Example:

```
# Enable C++ support
USER_BUILD_CPP_APPL = ON
...
# Additional C/C++ source files to be compiled
CUSTOM_SRC_FILES = main.c example.cpp
...
# Additional user object files to be linked
USER_OBJS = MyModule3.o03 MyModule4.cppo03
...
# Additional user libraries to be linked
USER_LIBS = MyCLib.lib MyCppLib.lib
```

The number at the object files reflect the different processor platforms. In the example, `.o03` and `.cppo03` shows that the files has been built for a DS1401 (MicroAutoBox II).

For further information on the user makefile, refer to DsBuildTemplate.mk on page 316.

Debugging an Application

Introduction

Simple application errors can be found by implementing messages in your source code to log measured or calculated values of variables (refer to Message Handling on page 126).

ntoppc-objdump

Syntax

```
ntoppc-objdump [options] objfile
```

Purpose

To display information about one or more object files.

Description

This utility is mainly used for debugging purposes. For example, it can disassemble an object file and show the machine instructions with their memory locations. The display of particular information is controlled by command line options. At least one option besides `-l` (`--line-numbers`) must be specified.

> **Note**
>
> To make it possible for ntoppc-objdump to display correlating source code information, you must build your application with the debug option `-g`.

You can find this utility in `<RCP_HIL_InstallationPath>\Compiler\QNX650\host\win32\x86\usr\bin`.

Options

The following command line options are available:

Option		Meaning
-a	--archive-headers	Shows object file format and header information from an archive object file.
	--adjust-vma=<offset>	Adds `offset` to all the section addresses. This is useful for dumping information, if the section addresses do not correspond to the symbol table.
-b <bfdname>	--target=<bfdname>	Specifies the object code format as `bfdname`. This might not be necessary, because many formats can be recognized automatically. You can list the available formats with `-i`.
-g	--debugging	Displays debugging information using a C-like syntax.
-e	--debugging-tags	Displays debugging information using ctags style.

Option		Meaning
-W	--dwarf	Displays DWARF information in the file.
-C	--demangle	Decodes low-level symbol names into user-level names. As style, you can specify: ▪ auto ▪ gnu ▪ lucid ▪ arm ▪ hp ▪ edg ▪ gnu-v3 ▪ java ▪ gnat
-d	--disassemble	Displays the assembler mnemonics for the machine instructions from sections which are expected to contain instructions.
-D	--disassemble-all	Displays the assembler mnemonics for the machine instructions from all sections.
-M	--disassembler-options=<options>	Specifies options to be used by the disassembler. Use `-H` to get a list of the available options.
-z	--disassemble-zeroes	Also disassembles blocks of zeros.
	--prefix-addresses	Prints the complete address of the disassembled code on each line. This is the older disassembly format.
-EB	--endian=big	Specifies the object file as big endian.
-EL	--endian=little	Specifies the object file as little endian.
-f	--file-headers	Displays summary information from the overall header of each file on **`objfile`**.
-h	--section-headers --headers	Displays summary information from the section headers of the object file.
-H	--help	Displays the **`objdump`** usage.
@<file>		Reads the options from the specified file.
-i	--info	Displays a list showing all architectures and object formats available for specification with `-b` or `-m`.
-I	--include=<folder>	Adds the specified folder to the search list for source files.
-j <section>	--section=<section>	Displays information only for the specified section.
-l	--line-numbers	Labels the display with the file name and the source line numbers corresponding to the object code shown. This option is useful only with `-d` or `-D`.
-m <machine>	--architecture=<machine>	Specifies the architecture the object file is for. You can list the supported architectures by using `-i`.
-p	--private-headers	Displays information that is specific to the object file format.
-r	--reloc	Displays the relocation entries of the object file. If used with `-d` or `-D`, the relocations are printed interspersed with the disassembly.

Option		Meaning
-R	--dynamic-reloc	Displays the dynamic relocation entries of the object file. This is only useful for dynamic objects, such as certain types of shared libraries.
-s	--full-contents	Displays the full contents of any sections requested.
-S	--source	Displays source code intermixed with disassembly, if possible. This option implies `-d`.
	--show-raw-insn	Displays disassembled instructions in HEX as well as in symbolic form.
	--no-show-raw-insn	Does not display the instruction bytes of disassembled instructions.
-G	--stabs	Displays the contents of .stab, .stab.index and .stab.excl sections from an ELF file.
	--start-address=<address>	Starts displaying at the specified address. This affects the output of the `-d`, `-r` and `-s` options.
	--stop-address=<address>	Stops displaying at the specified address. This affects the output of the `-d`, `-r` and `-s` options.
-t	--syms	Displays the symbol table entries of the object file.
-T	--dynamic-syms	Displays the dynamic symbol table entries of the object file. This is only useful for dynamic objects, such as certain types of shared libraries.
-w	--wide	Formats some lines for output devices that have more than 80 columns.
-v	--version	Displays the version number.
-x	--all-headers	Displays all available header information, including the symbol table and relocation entries. This option implies `-a`, `-f`, `-h`, `-r` and `-t`.

Example

For debugging an application, it is useful to disassemble all sections together with information on the line numbers and corresponding source code of the displayed assembler instructions. ntoppc-objdump prints a great amount of data, so it is recommended to redirect the output to a dump file, which you can open with a text editor. The command looks like this:

```
ntoppc-objdump -S -l -D appl.rta > result.dmp
```

W