DS2302 Direct Digital Synthesis Board

DS2302 DSP Programming

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

Software Reference 51

Advanced Programming Techniques 111

About This Document

Contents

This document provides detailed instructions on programming slave applications running on the floating-point DSPs.

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.

Introduction

Introduction

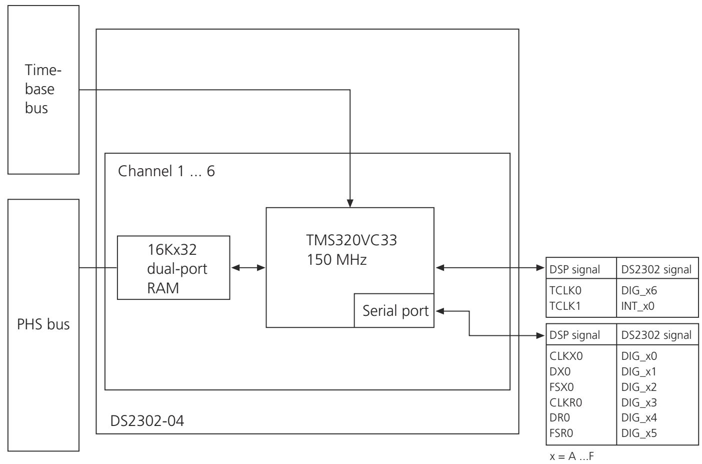
The DS2302 Direct Signal Synthesis (DDS) board is specifically designed for signal generation. It can be used to generate complex waveforms, for example, for hardware-in-the-loop simulation applications.

Architectural Overview

Introduction

The DS2302 Direct Digital Synthesis (DDS) board (starting from board revision DS2302-04) is a member of the PHS-bus-based system specifically designed for signal generation and analysis. The DS2302 board can cover one I/O-module for each channel. Its key features are:

- 6 independent channels featuring
  - TMS320VC33 floating-point DSP with 13 ns cycle time
  - 34K × 32-bit on-chip RAM
  - 16K × 32-bit dual-port memory for program downloading and host communication.
  - Digital interrupt input.
  - 6 digital I/O lines, each programmable as input or output.
  - One I/O module slot for analog I/O modules:
    - DS2302M DAC module
- All DSPs can communicate with each other via dual-port memory.
- All DSPs can be interrupted by each other.
- PHS bus interface.
- PHS bus interrupt controller.
- PC host interface for application loading purposes.



The DS2302 consists of 6 independent channels, each based on the Texas Instruments TMS320VC33 floating-point Digital Signal Processor (DSP), which is the main processing unit providing fast instruction cycle time for numerically intensive algorithms. Each DSP has been supplemented by a 16 KWords dual-port memory for program downloading and communication by a host, a master processor board or any of the on-board DSPs. Each DSP is connected to an I/O slot, covering one I/O module.

The DS2302 can act as an intelligent input or output unit which generates complex waveforms or performs signal preprocessing in a PHS-bus-based system.

Getting Started

Introduction

This section introduces you to the basic features of the DS2302 board. It demonstrates the implementation of a simple slave applications.

Where to go from here

Information in this section

Implementing a Simple Signal Generator

Where to go from here

Information in this section

First Steps in Programming the Slave DSP

Introduction

Some preparatory step are required for programming the slave DSP.

First steps

The DS2302 software needs the `timer30.h` and `serprt30.h` header files that come with the Texas Instruments C compiler. If not already done, these files must be extracted from the source library `prts30.src`. To extract all files from the library, type the following at the DOS prompt:

```
bldtirts.bat
```

As you will see in the following sections, implementing simple signal generator applications on the DS2302 DDS board is easy. You normally do not need to know any of the hardware and software details of the DS2302 at this point.

All you need to run the example applications listed in this section is an oscilloscope connected to output channel 1 of your DS2302 board to make sure that the slave applications are operating properly.

Note that the sampling rates in the example programs have not been optimized and do not make full use of the maximum processor performance.

To load slave applications to the DS2302, it must be connected to a processor board via the PHS bus.

When you have successfully worked through the Getting Started section and you are familiar with the DS2302's basic features, you can refer to Advanced Programming Techniques on page 111 for further details. This contains suggestions and examples for programming of high-performance signal generators and synchronizing several DSPs, and other useful information.

Related topics

Basics

Application Program Structure

Introduction

Most slave applications consist of a **main()** routine performing initialization and background operations and a timer interrupt service routine executing the signal generation algorithm at a specified sampling rate. The name of the timer interrupt service routine is **c_int09()** or **c_int10()**, depending on which of the TMS320VC33's two on-chip timers is used.

A typical application program frame reads as follows:

```
#include <ds2302.h>     /* hardware definitions and macros */

#define TS 10.0e-6                      /* sampling period */
...

void c_int09()         /* timer0 interrupt service routine */
{
  ...
  *dac = (long) ( ... );      /* write output value to DAC */
}

main()                                    /* main routine */
{
  init();                          /* initialize hardware */
  timer0(TS);                        /* initialize timer0 */
  for (;;);                    /* wait for timer interrupts */
}
```

The **init()** function performs hardware initialization and must be called in the initialization section of a slave application (i.e., normally before the sampling clock timer is started). The **timer0(time)** function initializes and starts the TMS320VC33's timer0 to generate timer interrupts at the sampling rate specified by the **time** parameter. If execution of the interrupt service routine cannot keep up with the rate at which timer interrupts are received (i.e., if the processor is overloaded), interrupts are lost. Both functions are contained in the object library **ds230x.lib**, which is automatically linked if the standard compile/assemble tool **cl230x.exe** is used (refer to CL230x on page 44).

The **ds2302.h** header file contains frequently used definitions, declarations, macros and function prototypes for the **ds230x.lib** object library. This header file must be included in every slave application.

If you do not need backward compatibility to the older versions of the DS2302 board, you can additionally include `ds2302_advanced.h`, to get access to the new features of the DS2302-04 board. Using this header file will also cause the build process to use the additional 32KW RAM of the VC33 DSP for your slave application.

When initialization has finished, the main routine enters a forever-loop. The timer interrupt service routine (i.e. `c_int09()` in the example) is executed each time a timer interrupt is received. The forever-loop can execute a background process such as parameter calculation or host communication. The background process is interrupted by timer interrupts.

The timer interrupt routine performs the actual signal generation and writes the generated output signal to the D/A converter (DAC). Access to the DAC is performed by writing to a memory-mapped 16-bit register. A pointer to this register named `*dac` is already declared in the `ds2302.h` header file. Since this register is declared to be of long type, a float output value must be scaled to a 16-bit integer value and cast to a variable of long type before it is written to the DAC output register.

If a floating-point value in the range (-1.0 ≤ `value` < +1.0) has to be scaled to the full DAC output voltage range (±10 V) and written to the DAC output register, you can also use the `dac_out(value)` macro from the `ds2302.h` header file.

Related topics

References

CL230x.. 44

How to Execute A Simple Slave Application

Objective

A sawtooth generator is a simple signal generator. It is copied to the hard disk during software installation and can be found in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\tutorial`.

Description

The following listing shows the `sawtooth.c` file.

```
/* SAWTOOTH.C ****************************************************************

  Simple Sawtooth Generator

  application demo program for the DS2302 DDS board

  (C) 1997 dSPACE GmbH

  $RCSfile:$ $Revision:$ $Date:$
*****************************************************************************/

#include <ds2302.h>                         /* hardware definitions and macros */

#define TS 10.0e-6                                         /* sampling period */

#define AMPL (0.5 * SCAL)                       /* amplitude of output signal */
#define FREQ 333.3                                  /* signal frequency in Hz */

float sawtooth = 0.0;                                       /* sawtooth value */
float delta_s;                          /* sawtooth increment per sampling step */


void c_int09()                              /* timer0 interrupt service routine */
{
  sawtooth += delta_s;                                     /* integrate delta_s */

  if (sawtooth > AMPL)                      /* if signal has reached amplitude */
    sawtooth -= AMPL;                                    /* subtract amplitude */

  *dac = (long) sawtooth;                            /* write output value to DAC */
}

main()
{
  delta_s = AMPL * FREQ * TS;                   /* initialize sawtooth increment */

  init();                                      /* initialize hardware system */
  timer0(TS);                                          /* initialize timer0 */
  for(;;);                                        /* wait for timer interrupts */
}
```

To keep the slave application very simple, the `AMPL` and `FREQ` parameters are defined as symbolic constants. They can be altered only by modifying the source code and recompiling the slave application.

Precondition

The DS2302 must be connected to a processor board via PHS bus.

Method

To execute a simple slave application

1 Unzip `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\tutorial\tutorial.zip` to your working folder.

2 On the Windows Start menu, select dSPACE RCP and HIL 20xx-x – Command Prompt for dSPACE RCP and HIL 20xx-x to open a Command Prompt window in which the required paths and environment settings are preset.

3 Change to your working folder.

4 To compile and download the main application, enter the following command:

   `down<xxxx> loadDemo`

   `down<xxxx>` must correspond to the processor board type, for example, `down1006` for a DS1006.

   The slave application is loaded to the slave DSP via the `loadDemo` processor board application.

5 Use an oscilloscope to observe the sawtooth signal at the DAC output of channel 1.

How to Implement Your Own Signal Generator

Objective

You can use the sawtooth generator to implement a pulse-width-modulation (PWM) generator.

Description

The PWM generator is not installed with the demo software so that you can create it yourself from the sawtooth example.

In the example, the amplitude of the sawtooth signal is set to 1.0 to allow direct comparison with the duty cycle. The output signal must be set to its high output level at the beginning of each sawtooth period and must change from high to low every time the duty cycle expires.

Method

To implement your own signal generator

1 Unzip `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\tutorial.zip` to your working folder.

2 On the Windows Start menu, select dSPACE RCP and HIL 20xx-x – Command Prompt for dSPACE RCP and HIL 20xx-x to open a Command Prompt window in which the required paths and environment settings are preset.

3 Change to your working folder.

4 Copy the file `sawtooth.c` to a file named `pwm.c`.

5 Open an editor and modify the `pwm.c` file as follows (changes are printed bold).

```
#include <ds2302.h>     /* hardware definitions and macros */
#define TS 10.0e-6                      /* sampling period */
#define FREQ       333.3         /* signal frequency in Hz */
#define DUTY_CYCLE 0.25              /* pulse/period ratio */
#define LOW        (0.0 * SCAL)           /* PWM low value */
#define HIGH       (0.5 * SCAL)          /* PWM high value */

float sawtooth = 0.0;                    /* sawtooth value */
float delta_s;     /* sawtooth increment per sampling step */

void c_int09()         /* timer0 interrupt service routine */
{
   sawtooth += delta_s;              /* integrate delta_s */

   if (sawtooth > 1.0)              /* subtract amplitude */
     sawtooth -= 1.0; /* if sawtooth has reached amplitude */

   if (sawtooth < DUTY_CYCLE)  /* set output to high value */
     *dac = (long) HIGH;     /* during duty cycle duration */
   else                      /* and to low value otherwise */
     *dac = (long) LOW;
}

main()
{
  delta_s = FREQ * TS;     /* initialize sawtooth increment */
  init();                     /* initialize hardware system */
  timer0(TS);                          /* initialize timer0 */
  for(;;);                     /* wait for timer interrupts */
}
```

6 Compile and convert the slave application by entering the following command:

```
CL230x pwm.c
```

This generates a C file called `Slv2302_pwm.slc` containing the slave application data.

7 Now edit the `loadDemo.c` processor board application to include the new slave application. Open an editor and modify the load function in `loadDemo.c` as follows:

```
...
#include Slv2302_pwm.slc

...

   ds2302_load_board(DS2302_1_BASE, 0,
      DS2302_START_CHANNELS_SYNC | DS2302_RESET_ALL_CHANNELS,
      pwm, NULL, NULL, NULL, NULL, NULL)
```

8 To compile and download the main application, enter the following command:

`down<xxxx> loadDemo`

`down<xxxx>` must correspond to the processor board type, for example, `down1006` for a DS1006.

The slave application is loaded to the slave DSP via the `loadDemo` processor board application.

Result

The PWM signal should appear on the oscilloscope.

Keep the source file `pwm.c` because it is needed to create further examples.

Related topics

HowTos

How to Execute A Simple Slave Application........ 16

Accessing the DS2302 Board

Introduction

The DS2302 DDS board can be accessed by a processor board via PHS bus. You require access, for example, if signal generator parameters need to be changed on-the-fly, without interrupting the signal generation.

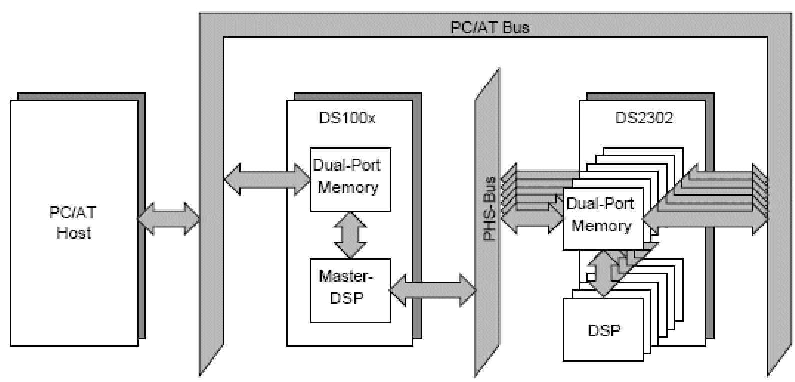
Where to go from here

Information in this section

DS2302 Access via PHS Bus

Accessing a DS2302

The illustration below shows how the DS2302 board is accessed by a processor board via PHS bus.



To control DS2302 functions and access DS2302 memory bus by a master processor program via PHS you need the master processor interface functions and the related header file `ds2302.h`. This file contains definitions and function prototypes and must be included in master processor programs accessing the DS2302.

Note

The DS1006 processor board uses the IEEE floating-point format, which is different from the DS2302's VC33 floating-point format. To write or read floating-point values from a DS1006 board to the DS2302 board, you must use the `ds2302_read_float()`, `ds2302_write_float()`, `ds2302_read_block_float()` and `ds2302_write_block_float()` functions. Alternatively, the IEEE conversion macros `RTLIB_CONV_FLOAT32_TO_TI32()` or `RTLIB_CONV_FLOAT32_FROM_TI32()` from the DSSTD module can be used in a DS1006 application.

How to Access a DS2302 Board

Objective

The instructions show the implementation of a pulse-width-modulation (PWM) generator whose duty cycle can be modified by a real-time application running on a processor board.

Accessing a DS2302

The PWM generator is modified to allow modification of the duty cycle by a processor board. To be accessed by the processor board, the duty cycle can no longer be defined as a symbolic constant. It must be stored in a variable allocated at a defined address in the DSP's dual-port memory.

In the listing, the `duty_cycle` variable is declared as a pointer. The pointer is initialized by the `DP_MEM_BASE` constant to point to the DS2302's first dual-port memory location. `DP_MEM_BASE` is a symbolic constant defined in the `ds2302.h` header file. In the main routine, the contents of this memory location are initialized by the `DUTY_CYCLE` constant.

All pointer variables accessing the dual-port memory must be declared as volatile to avoid problems with the optimizer of the compiler. Without using volatile the dual-port memory variable may be optimized to an internal register variable, because the compiler does not know that the dual-port memory is accessed and changed from outside the application.

When the duty cycle is a variable in the dual-port memory, the processor board can modify the duty cycle online, i.e., while the DSP is executing the slave application. The following main application named `pwmctrl<xxxx>.c` continually increases the duty cycle in 0.01 steps in the range 0.0 … 1.0. The program is included in `tutorial.zip` installed in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\tutorial`.

Precondition

You must have implemented the PWM generator, see How to Implement Your Own Signal Generator on page 18.

Method

To access a DS2302 board

1 Unzip `tutorial.zip` to your working folder.

2 Open an editor and modify the `pwm.c` file as follows (changes are printed bold).

```
#include <ds2302.h>              /* hardware definitions and macros */
#include <util2302.h>          /* additional definitions and macros */

#define TS 10.0e-6                                /* sampling period */

#define FREQ       333.3                   /* signal frequency in Hz */
#define DUTY_CYCLE 0.25                /* initial pulse/period ratio */
#define LOW        (0.0 * SCAL)               /* PWM low level value */
#define HIGH       (0.5 * SCAL)              /* PWM high level value */

float sawtooth = 0.0;                              /* sawtooth value */
float delta_s;               /* sawtooth increment per sampling step */

/* pulse/period ratio */
volatile float *duty_cycle = (float *) DP_MEM_BASE;

void c_int09()                     /* timer0 interrupt service routine */
{
  sawtooth += delta_s;                           /* integrate delta_s */

  if (sawtooth > 1.0)                          /* subtract amplitude */
    sawtooth -= 1.0;            /* if sawtooth has reached amplitude */

  if (sawtooth < *duty_cycle)              /* set output to high value */
    *dac = (long) HIGH;                      /* during pulse duration */
  else                                   /* and to low value otherwise */
    *dac = (long) LOW;
}

main()
{
  delta_s = FREQ * TS;      /* initialize sawtooth increment */
  *duty_cycle = DUTY_CYCLE;                  /* initialize duty cycle */

  init();                                /* initialize hardware system */
  timer0(TS);                                    /* initialize timer0 */
for(;;);                                  /* wait for timer interrupts */
}
```

The processor board can now access the dual-port memory and modify the duty cycle online, i.e., while the DSP is executing the slave application.

3 Compile and convert the slave application by entering the following command:

```
CL230x pwm.c
```

This generates a C file called `Slv2302pwm.slc` containing the slave application data.

4 The following application named `pwmctrl<xxxx>.c` continually increases the duty cycle in 0.01 steps in the range 0.0 … 1.0.

```
#include <brtenv.h>                        /* basic real-time environment */
#include <ds2302.h>
#include "Slv2302_pwm.slc"

#define DUTY_CYCLE_ADDR DS2302_DP_MEM_BASE          /* duty cycle address */
#define TS              10.0e-3               /* duty cycle update period */
#define D_DUTY_CYCLE     1.0e-2           /* duty cycle update steps size */
#define DUTY_CYCLE_MAX  1.0                /* duty cycle range 0.0 .. 1.0 */

dsfloat duty_cycle;                                 /* current duty cycle */


/*----------------------------------------------------------------------*/
void isr_t1()                          /* timer 1 interrupt service routine */
{
  ds2302_write_float
    (DS2302_1_BASE, DS2302_CH1, DUTY_CYCLE_ADDR, &duty_cycle);

  duty_cycle += D_DUTY_CYCLE;
  if (duty_cycle > DUTY_CYCLE_MAX)
    duty_cycle -= DUTY_CYCLE_MAX;
}


/*----------------------------------------------------------------------*/
main()
{
  init();                                  /* initialize hardware system */
  ds2302_init(DS2302_1_BASE);                 /* initialize DS2302 board */
  ds2302_load_board(DS2302_1_BASE, 0,
        DS2302_START_CHANNELS_SYNC | DS2302_RESET_ALL_CHANNELS,
        pwm, NULL, NULL, NULL, NULL, NULL)
  msg_info_set(MSG_SM_RTLIB, 0, "System started.");

  /* read initial duty cycle from the DS2302's dual-port memory  */
  ds2302_read_float(DS2302_1_BASE, DS2302_CH1, DUTY_CYCLE_ADDR, &duty_cycle);

  RTLIB_SRT_START(TS, isr_t1);          /* initialize sampling clock timer */

  for (;;)                                  /* wait for timer interrupts */
  {
    RTLIB_BACKGROUND_SERVICE();
  }
}
```

5 To compile and download the main application, enter the following command:

`down<xxxx> pwmctl<xxxx>`

`down<xxxx>` must correspond to the processor board type, for example, `down1006` for a DS1006.

The slave application is loaded to the slave DSP via the main application.

Result

The PWM signal appears on the oscilloscope with the duty cycle changing continuously between 0.0 and 1.0.

Related topics

HowTos

How to Implement Your Own Signal Generator.. 18

Standard Slave Applications

Introduction

Some slave applications for demonstrating the DS2302 features are installed with the DS2302 software.

Where to go from here

Information in this section

Overview of the Standard Slave Applications

Standard slave applications

The following slave applications come with the DS2302 software. They are installed in subfolders under `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\`. The following table lists the slave applications and the subfolders.

Demo Application	Subfolder	Refer to...
Sine wave generator	`sin`	Sine Wave Generator (sin) on page 28
PRBS and Gaussian noise generators	`noise`	PRBS and Gaussian Noise Generators (noise) on page 28
D/F conversion	`d2f`	D/F Conversion (d2f) on page 32
PWM generation	`pwm`	PWM Generation (pwm) on page 33
Table look-up	`tlu`	Table Look-up (tlu) on page 34
Crankshaft sensor signal generator	`crank`	Crankshaft Sensor Signal Generator (crank) on page 35
Crankshaft/camshaft sensor signal generator	`crcam`	Polling the INT1 Interrupt Flag (crcam) on page 124
Knock sensor signal generator	`knock`	Using an INT1 Interrupt Service Routine (knock) on page 126
Function generator	`fgen`	ControlDesk Controls DS2302 via Master Processor (fgen) on page 130
Fourier synthesis	`fourier`	Superposition of Signals (Fourier) on page 139
Incremental sensor simulator	`incr`	Using the Digital I/O Lines (incr) on page 135
dSPACE logo xy-image generator	`draw`	X/Y Image Generator (draw) on page 145

The following topics contain excerpts of the source codes. For the complete listings, refer to the respective source files.

All the source files are zipped. Before you can use them, you must unzip them to your working folder.

Measuring execution times

Execution time requirements can be measured by using the SPEEDCHK utility and with automatic assembly code optimization performed by SPEEDUP. For further details of SPEEDCHK and SPEEDUP, refer to Utilities on page 148. The SPEEDUP section also contains a complete table of execution time requirements for the demo applications with and without automatic assembly code optimization.

Loading slave applications

The applications for the slave DSP are converted into SLC files and must be loaded to the DS2302 board via the processor board application (see Loading Slave Applications on page 38). If the demo does not contain a specific processor board application, the `loadDemo` application is available that loads the DSP application(s) via the processor board.

Sine Wave Generator (sin)

Introduction

The sine wave generator slave application `sin5.c` is based on a 5th-order polynomial sine approximation, which yields a maximum error that is below the resolution of the 16-bit DAC. All the necessary files are in <RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\sin. This folder also contains the `sweep100x.c` main applications, which you can use to continually alter the sine wave frequency like a sweep function generator does.

The timer interrupt service routine of the sine wave generator is listed below.

```
void c_int09()          /* timer0 interrupt service routine */
{
  angle += d_angle;                  /* add angle increment */
  if (angle > PIH)
  {
    angle -= PI;             /* subtract PI and toggle sign */
    scale = -scale;
  }
  angle_sq = angle * angle;                /* square angle */

  /* write output value to DAC */
  *dac = (long) (scale * *ampl *
    ((C5 * angle_sq + C3) * angle_sq + C1) * angle);
}
```

The polynomial approximation of the sine function is performed for the argument range ±π/2. This range is repeatedly passed and the angle increment depends on the required frequency and the current sampling rate. The remaining intervals of the sine function (-π..-π/2, π/2..π) are constructed by simply inverting its sign. Here the output scaling factor, needed to scale the floating-point output value to a 16-bit integer value, is used to carry the sign information.

The `ampl` parameter is located in the DS2302's dual-port memory for access by the host PC or by a master processor board.

PRBS and Gaussian Noise Generators (noise)

Introduction

PRBS signals and noise signals with a Gaussian probability density, are often used for system identification purposes. They can also be used for reproducing real signals, such as knock-sensor signals. A knock-sensor signal generator is described in Advanced Programming Techniques on page 111.

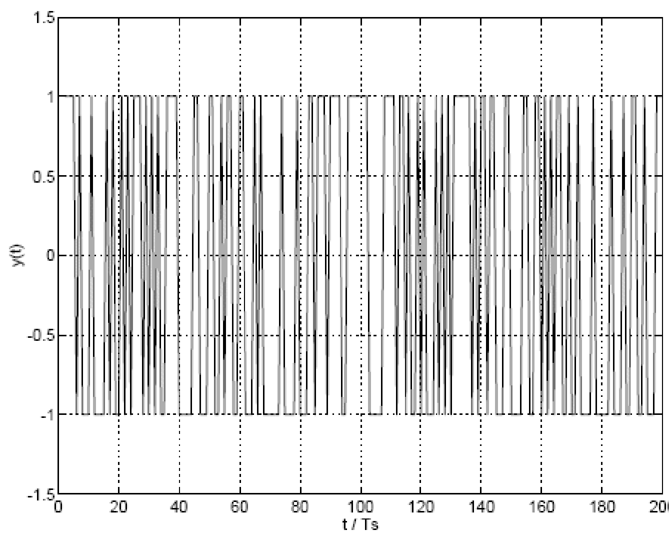
The PRBS application generates a pseudo-random binary sequence (PRBS) by using a 31-bit shift register with appropriate modulo 2 feedback.

```
void c_int09()                              /* timer0 interrupt service routine */
{
  prbs_reg =
    (prbs_reg >> 1) | ((prbs_reg ^ (prbs_reg << 28)) & 0x40000000);

  /* write output signal to DAC */
  *dac = (long) (SCAL * *ampl *
    ((float) (prbs_reg & 0x00000001) * 2.0 - 1.0));
}
```

The amplitude depends on the value of `*ampl` and is initially set to 0.5 in the application demo program. The following illustration assumes an amplitude of ±1.0. The signal has only two discrete values, i.e., the maximum positive and negative amplitudes.

The following illustration shows the PRBS time history.



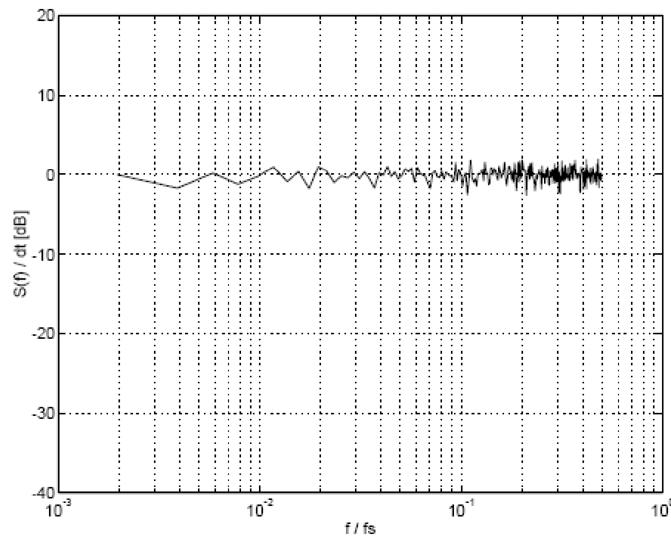
The sequence period *T* (i.e., the duration until the sequence is repeated) is given by the equation

$T = (2^n - 1)\,\Delta t$

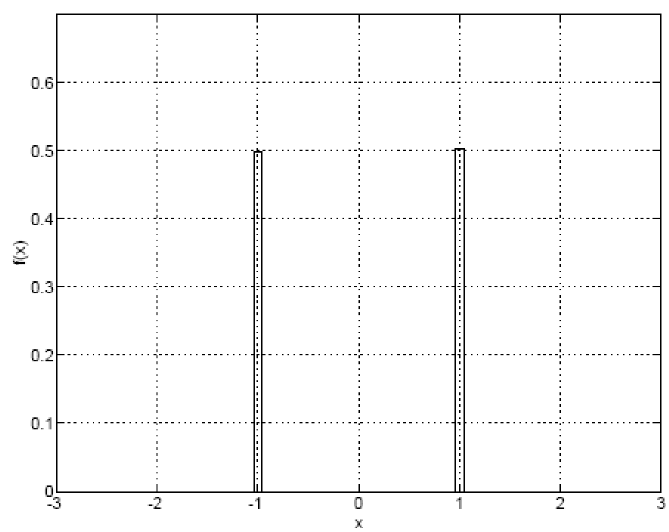
where *n* is the PRBS shift register word length and Δt is the sampling period. Thus a sampling period of Δt = 0.05 ms (20 kHz sampling rate) yields a PRBS sequence which repeats after *T* = 107374 s (approximately 30 hours).

The following two illustrations show the power spectral density and the probability density of the PRBS signal. The average of 100 FFT results has been calculated, to obtain the power density spectrum. A Hanning window has been applied to the sampled data, to reduce spectral leakage. Because the PRBS signal has only two discrete values, the probability density has peaks at -1.0 and +1.0.

The following illustration shows the power spectral density of PRBS signal.



The following illustration shows the probability density of the PRBS signal.



The Gaussian noise generator is based on the PRBS signal described above, which is fed through a 3rd-order allpass filter.

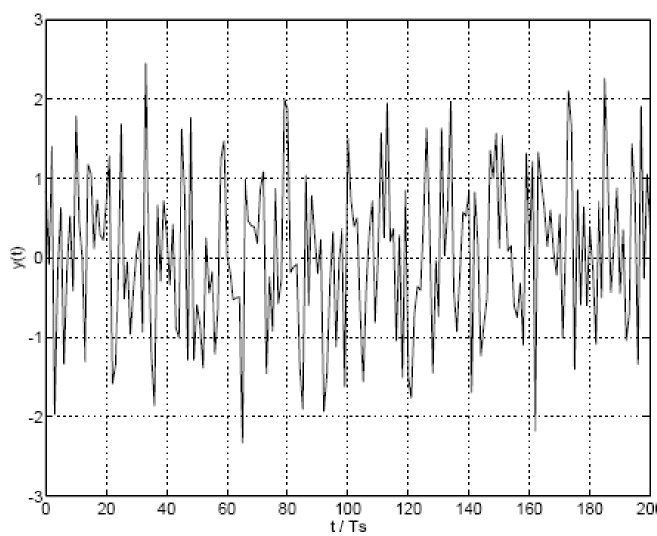
```
void c_int09()                                    /* timer0 interrupt service routine */
{
  prbs_reg =
    (prbs_reg >> 1) | ((prbs_reg ^ (prbs_reg << 28)) & 0x40000000);

  rk = (float) (prbs_reg & 0x00000001) * 2.0 - 1.0 -
    A2 * rk1 - A1 * rk2 - A0 * rk3;
  yk = GAIN * (rk + B2 * rk1 + B1 * rk2 + B0 * rk3);
  rk3 = rk2;
  rk2 = rk1;
  rk1 = rk;

  *dac = (long) (SCAL * *ampl * yk);             /* write output signal to DAC */
}
```

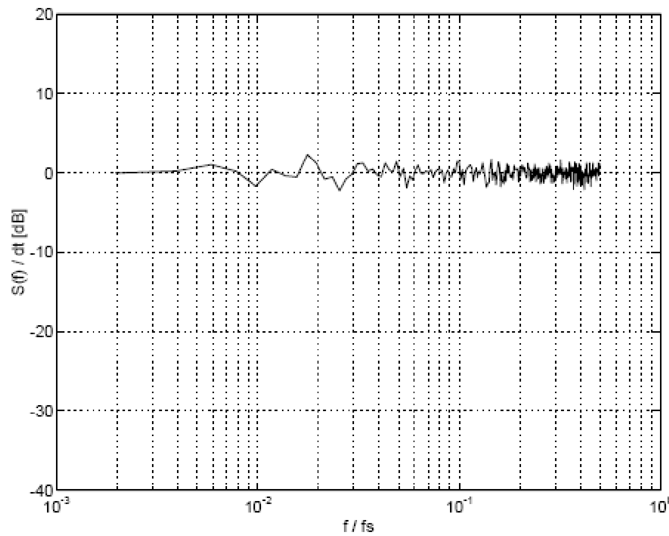
The maximum amplitude increases to approximately ±3.2 · `*ampl`, and `*ampl` is initially set to 0.2 in the demo application program. The examples below use an unscaled amplitude at the allpass filter output of approximately ±3.2, which corresponds to a variance of $\sigma = 1.0$.

The following illustration shows the Gaussian noise time history.

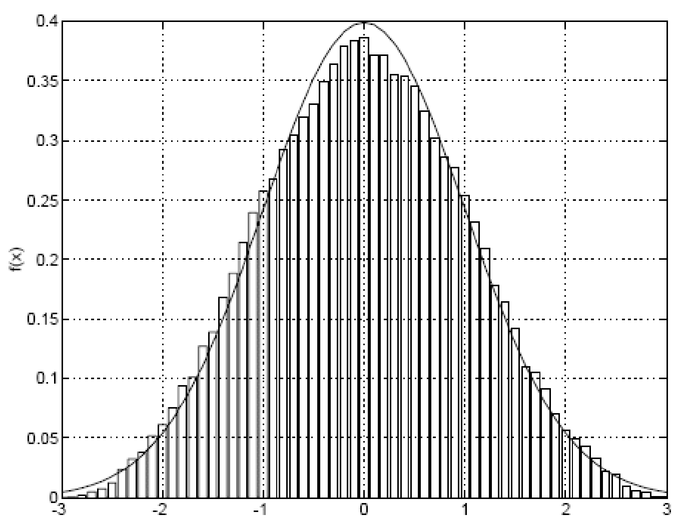


The illustrations below show that the probability density is completely different from the raw PRBS signal, while the power spectral density is almost unchanged.

The following illustration shows the power spectral density of the Gaussian noise signal.



The following illustration shows the probability density of the Gaussian noise signal.



D/F Conversion (d2f)

Introduction

The slave application listed below shows a very simple D/F conversion application. It generates a square-wave signal of variable frequency. The reference frequency is supplied by the host PC or a master processor board via

the dual-port memory. The relevant files are in
<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\d2f.

```
void c_int09()            /* timer0 interrupt service routine */
{
  time += TS;                                  /* update time */
  if (time >= thresh)                /* test if period expired */
  {
    time -= thresh;                      /* subtract threshold */
    y *= -1;                             /* toggle output sign */
  }
  *dac = y;                        /* write output value to DAC */
}
```

The actual threshold depends on the current reference frequency. Since computation of the threshold requires a division and does not need updating in every sampling step, it is computed in the background loop to keep the execution time of the timer interrupt service routine small.

```
main()
{
  ...

  for (;;)
  {
    thresh = 0.5 / *freq;                  /* update threshold */
  }
}
```

PWM Generation (pwm)

Introduction

The PWM generator is an extended version of the D/F conversion application described in the previous section. The duty cycle and the period of the generated square-wave signal are variable and can be altered by a master processor board via the dual-port memory. Thus the PWM generation program can also be used to implement D/F conversion. The relevant files are in
<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\pwm.

```
void c_int09()            /* timer0 interrupt service routine */
{
  time += TS;                                  /* update time */
  if (time > thresh2)                /* test if period expired */
  {
    time -= thresh2;
    *dac = HIGH;
  }
  else if (time > thresh1)     /* test if duty cycle expired */
  {
    *dac = LOW;
  }
}
```

The signal period must be supplied here instead of the frequency. A second threshold is used to evaluate when the duty cycle is expired and the DAC output level must change from high to low.

```
main()
{
  ...

  for (;;)
  {
    thresh1 = *duty_cycle * *period;                /* update duty cycle threshold
*/
    thresh2 = *period;                                  /* update period threshold
*/
  }
}
```

Table Look-up (tlu)

Introduction

The
`<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\tlu`
folder contains two table look-up example programs. The `tlu.c` slave application uses an array for table data storage and `ptlu.c` uses a chained list, which can be read out much faster. However, the latter version requires more memory for data storage. The data type for a chained list can be defined as shown below.

```
typedef struct tbl_struct tbl_t;

struct tbl_struct
{
  long value;
  tbl_t *next;
};
```

Each list element contains a data value, i.e., a sample of the signal to be generated, and a pointer to another list element. The data values must be initialized in the `main()` routine to form the signal waveform and each element's `next` pointer must point to the subsequent list element. The `next` pointer of the last element contained in the list is set to point to the first list element to obtain a closed loop.

The following code to read out the current data value and to proceed to the next list element is very short, and thus very fast.

```
void c_int09()           /* timer0 interrupt service routine */
{
  *dac = cur->value;           /* write current value to DAC */
  cur = cur->next;                /* proceed to next element */
}
```

The following times are only valid for the 60 MHz DS2302-01 board: This timer interrupt service routine requires a maximum of 11 timer ticks (0.73 µs) execution time. The array-based table look-up application `tlu.c` needs 16 timer ticks (1.06 µs).

If the data list becomes too large to fit into on-chip memory, it must be allocated in the dual-port memory, though this increases execution time requirements a little bit. In this case the list can also be initialized by a master processor board.

Crankshaft Sensor Signal Generator (crank)

Introduction

The generation of crankshaft sensor signals is useful application example for the DS2302 board. Such signals are needed in the automotive development field, for example, in conjunction with hardware-in-the-loop experiments.

The slave application is based on a table look-up technique where table look-up is angle-controlled. It generates a pulse train signal with 7 pulses per revolution distributed uniformly over the crankshaft circle. The last pulse is missing, so 6 pulses per revolution are actually generated. The demo files are in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\crank`.

Although the table look-up is angle-controlled, it is implemented in a timer interrupt service routine. During each sampling step, the angle is incremented by an individual angle increment which depends on the current engine speed and on the sampling rate. The current angle increment per sampling step is continuously computed in the background to follow the engine speed supplied by a master processor board via the DS2302's dual-port memory.

The data types of the table elements are shown below.

```
typedef struct edge_struct edge_t;

struct edge_struct
{
  float pos;
  long value;
  float angle_reset;
  edge_t *next;
};
```

Each table element contains an output value and the corresponding angle. When this angle is reached, the output value becomes the current signal value and the `cur` table pointer is switched to the next table element. This allows the output samples to be distributed arbitrarily over the entire crankshaft circle.

The `next` pointer in each table element is initialized to point to the succeeding table element and the last table element is connected to the first, closing the loop. The `cur` table pointer is initially set to point to the first element corresponding to the 0 degree position.

The `angle_reset` value in each table element is used to reset the angle at the end of each revolution without needing another if-then construction. The `angle_reset` value is initialized to zero for all table elements except for the last one, which contains the value 360.

The listing below shows the timer interrupt service routine from the slave application.

```
void c_int09()          /* timer0 interrupt service routine */
{
  if (angle >= cur->pos)
  {
    *dac = cur->value;
    angle -= cur->angle_reset;
    cur = cur->next;
  }
  angle += delta_deg;
}
```

For achieve maximum precision, the maximum possible sampling rate has been chosen. The precision, i.e., the angle increment at the maximum engine speed $\Delta\Phi_{max}$ is given by the equation

$$\Delta\Phi_{max} = 360 \cdot N_{max} / 60 \cdot T_s$$

where N_{max} is the maximum speed in *rpm* and T_s is the sampling period in seconds. The example yields a resolution of 0.09 degrees at 6000 rpm engine speed.

Since this method of signal generation is based on a table look-up technique, it is not limited to pulse train signals. You can initialize the table to generate arbitrary signal waveforms. The complexity of the generated signal is limited only by memory constraints.

Note that two extended versions of this demo application are described in DSP Synchronization by Interrupts on page 124. These versions additionally generate camshaft sensor signals and knock sensor signals.

Transferring APU Values to a DS2302 Demo

Introduction

This application demonstrates the APU value transfer from the DS2211 board to a DS2302-04 board via the time-base connector.

> **Note**
>
> This application can only be used with a DS2302 starting with board revision DS2302-04. Earlier board revisions do not provide a time-base bus connector.

This demo application also demonstrates the use of messages for slave DSP debugging purposes. You can see the messages in the **Message Viewer** of the experiment software.

Preparations

To run the demo, connect the time-base connector of the DS2211 with the DS2302-04 connector using a standard 26-pin ribbon cable.

Basics

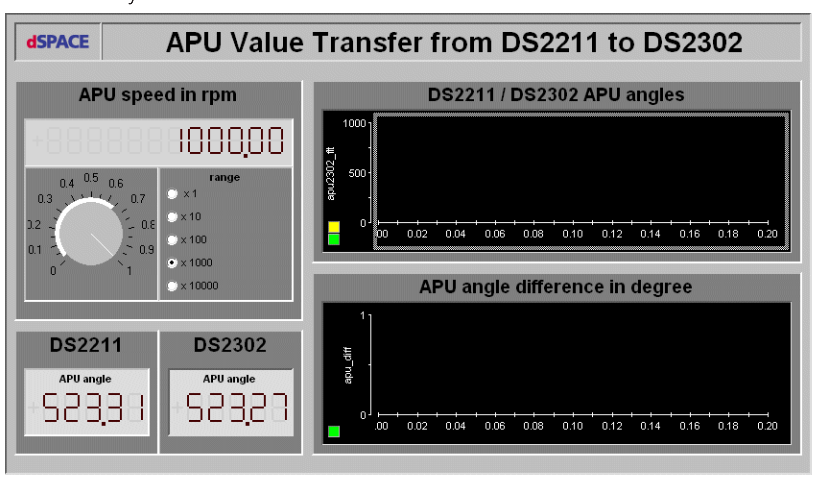
The APU export demo application consists of the `master100x.c` main application (running on processor board) and the `Rx2302.c` slave application (running on a slave DSP of the DS2302). The demo application is in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\Apu`.

ControlDesk project

To work with this demo, a backup file (`master100<x>.ZIP`) for ControlDesk is installed in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\Apu`. It is not necessary to unzip the backup file, you can open it directly in ControlDesk. Refer to Open Project + Experiment from Backup (ControlDesk Project and Experiment Management 📖).

Main application

The `master100x.c` main application loads the slave application to the DS2302 board and reads the APU values from both boards. They are displayed in the layout shown in the illustration below. The values are not exactly synchronous because they are not read at the same time.



Slave application

The `rx2302` slave application reads the APU value using a pointer to the APU register of the DSP channel. The value is received from the DS2211 board via the time-base connector of the DS2302 board.

Compiling and Loading Slave Applications

Introduction

Slave applications are loaded to the DS2302 Direct Digital Synthesis board via PHS bus by a processor board.

You can load individual slave applications to each of the 6 channels of the DS2302 board and start all the slave applications simultaneously.

Where to go from here

Information in this section

Loading Slave Applications

Where to go from here

Information in this section

Basics of Loading Slave Applications

Introduction

The host PC cannot access a slave DSP directly to load slave applications. To load slave applications to the slave DSP, you must first include them in the processor board application in an intermediate format (SLC format). When the processor board application is executed on the processor board, the slave applications are loaded to the slave DSP via the PHS bus.

The DS2302 board provides 6 independent direct digital synthesis (DDS) channels, each with a separate floating-point DSP. You can therefore use up to 6 independent slave applications on a DS2302.

How to Load a Slave Application via an RTLib Function

Basics

This loader concept allows the slave applications to be loaded by the master processor. The slave application data is stored as a C array in the .bss section of the master processor memory. The .bbs section is used for global variables and the slave application data is therefore always available.

The compile and link utility `CL230x.exe` compiles the slave application data and converts it to a C array by using the `coffconv` utility. The slave DSP application data of this C array is loaded to the slave DSP using the `ds2302_load_board` function.

Method

To load a slave application via an RTLib function

1. On the Windows Start menu, select dSPACE RCP and HIL 20xx-x – Command Prompt for dSPACE RCP and HIL 20xx-x to open a Command Prompt window in which the required paths and environment settings are preset.
2. Change to the folder of the slave application.
3. To compile and convert the slave application, enter the following command:

   ```
   CL230x test_prg.c
   ```

   This generates a C file called `Slv2302_test_prg.slc` containing the slave application data. Copy this file to the folder of your master processor application.
4. Add the following lines to your master processor application to load the slave application via the master processor board:

   ```
   /* DS2302 slave application data */

   #include "Slv2302_test_prg.slc"
   extern unsigned long test_prg[];
   void main(void)
   {
      init();        /* initialize master processor system */
      ds2302_init(DS2302_1_BASE);     /* initialize DS2302 */
      ds2302_load_board(DS2302_1_BASE, 0,
         DS2302_START_CHANNELS_SYNC | DS2302_RESET_ALL_CHANNELS,
         test_prg, NULL, NULL, NULL, NULL, NULL);

      ...

   }
   ```

In this example, the `test_prg` slave application is loaded to the first slave DSP and then started.

5 To compile and load your processor board application, enter the following command:

```
down<xxxx> master.c
```

`down<xxxx>` must correspond to the processor board type, for example, `down1006` for a DS1006.

Related topics

HowTos

References

How to Load Slave Applications via RTI

Basics

This loader concept allows the slave applications to be loaded by the master processor using constantly available slave application data. The slave application data is stored as a C array in the .bss section of the master processor memory. The .bbs section is used for global variables and the slave application data is therefore always available.

The compile and link utility `CL230x` compiles the slave application data and converts it to a C array by using the `coffconv` utility (refer to coffconv on page 43). The slave DSP application data of this C array is loaded to the slave DSP using the DS2302_DSP_SETUP_Bx block.

Method

To load a slave application via RTI

1 On the Windows Start menu, select dSPACE RCP and HIL 20xx-x – Command Prompt for dSPACE RCP and HIL 20xx-x to open a Command Prompt window in which the required paths and environment settings are preset.

2 Change to the folder of the slave application.

3 To compile and convert the slave application, enter the following command:

```
CL230x test_prg.c
```

This generates a C file called `Slv2302_test_prg.slc` containing the slave application data.

4 Start MATLAB and open the Simulink model.

5 Open the DS2302 blockset.

6 Drag the **DS2302_DSP_SETUP_Bx** block to the Simulink model.

7 Open the block dialog and specify the parameters:
   - On the **Unit** page, select the board number.
   - On the **Channel Configuration** page, select **Enable DSP application** for the channels that you want to use to load the slave applications.
   - Click **Browse** and select the slave applications.
   - Select interrupt sources, if required.

8 Build and download the real-time application.

Result

When the real-time application is started, the selected slave applications are loaded to the DSPs of the DS2302.

Related topics

HowTos

How to Set the Compiler Path.. 42

References

CL230x.. 44
coffconv... 43
DS2302_DSP_SETUP_Bx (DS2302 RTI Reference 📖)

Tools for Loading Slave Applications

Introduction

The following topics provide information on the utilities you can use to customize the software environment and to implement slave DSP applications.

Where to go from here

Information in this section

How to Set the Compiler Path

Objective

Before you can use `CL230x` to compile and link source code for the slave DSP of your DS2302 board, you have to specify the installation path of your Texas Instruments Compiler (TI Compiler) as an environment variable.

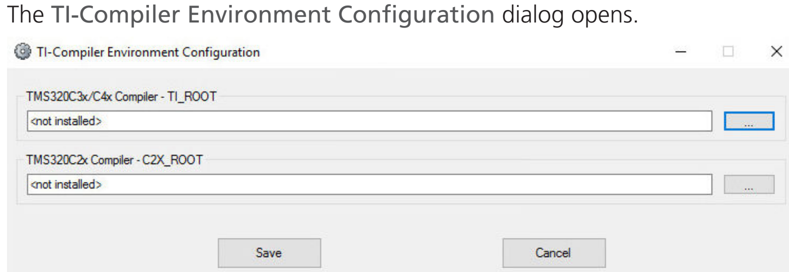
Method

To set a compiler path

1. From the Windows Start menu, select dSPACE RCP and HIL <ReleaseVersion> - Command Prompt for dSPACE RCP and HIL <ReleaseVersion>.

   A command prompt with required default settings is started.
2. Type `DsConfigTiEnv` and click **Enter**.

   The TI-Compiler Environment Configuration dialog opens.



3 Click the Browse button in the TMS320C3x/C4x Compiler - TI_ROOT setting to open a file explorer.

4 Navigate to the *main path* of the installed TI Compiler and click OK.

The main path to be specified depends on the installed compiler.

Compiler	Main Path
C3x/C4x TI Compiler Version 4.70 C3x/C4x TI Compiler Version 5.11	<InstallationPath>\c3xtools
C3x/C4x Code Composer Tools	<InstallationPath>\tic3x4x

5 Close the dialog by clicking Save.

Result

The compiler path of your TI compiler is set. The required paths for compiling and linking the source code of the slave DSP are now available in the Command Prompt for dSPACE RCP and HIL.

Related topics

References

CL230x.......... 44

coffconv

Syntax

```
coffconv obj_file [options] [/?]
```

Purpose

To convert a COFF (common object file format) object file to an assembly file that can be included into a processor board application. `coffconv` adds the prefix *Slv2302_* to the name of the given object file.

This tool is automatically invoked by the `CL230x` tool. Use it manually only if you have old DS2302 DSP object files without sources.

Options

The following command line options are available:

Option	Meaning
/a	Generates an assembly file with the default extension `asm`.
/b	Generates a binary file with the default extension `bin`.

Option	Meaning
/slc	Generates a C-source file with the default extension `slc`.
/n	Disables beep on error.
/o <output_file>	Name of the file to be generated
/q	Quiet mode
/t <board_type>	Specifies the target board type for the object file to be converted. In this case: DS2302
/?	Displays a list of the options available.

Generated files and loading mechanism

C-source file The coffconv output file contains a data array named according to the converted object file. The data array is needed for the loader function of the processor board application (refer to **`ds2302_load_board`** (DS2302 RTLib Reference)).

Assembler file The coffconv output file contains the data section `SlvSect` with the application data. This section will be loaded by the host loader to the master's memory only temporarily. When the application has been loaded to the slave DSP, the data section will be cleared from the memory. For more information on the slave loading procedure, refer to Loading Slave Applications on page 38.

Binary file The coffconv output file contains the application data and can be used by other conversion tools.

Example

```
coffconv demo.obj -slc -t DS2302
```

The object file `Demo.obj` will be converted to the assembly file `Slv2302_demo.slc`.

CL230x

Introduction

The `CL230x` tool compiles and assembles a DS2302 slave application and links it with the object library `ds230x.lib` and the Texas Instruments run-time library `rts30.lib`.

Syntax

```
cl230x source_file [options] [/?]
```

source_file `CL230x.exe` can be invoked with the name of a makefile (name.mk), the name of a C file (name.c) or with the name of an assembly file (name.asm). If no extension is specified the utility searches for the source file in the following order:

1. Makefile
2. C-source file
3. Assembly source file

Options The following command line options are available:

Option	Meaning
/ao <option>	Additional assembler options; refer to the Texas Instruments Assembler documentation.
/co <option>	Additional compiler options; refer to the C compiler documentation.
/g	Enables symbolic debugging (using the CL30 options –g –as).
/l	Writes all outputs to the `Cl230x.log` file.
/n	Disables beep on error.
/p	Pauses execution of `Cl230x.exe` after errors. The Command Prompt window is not closed automatically. This allows you to read error messages.
/s	Optimizes assembly code (SPEEDUP).
/so <option>	Additional speedup option (can be used several times).
/x	Switches code optimizing off.
/?	Displays a list of the options available.

Description

The `CL230x` tool uses the DSMAKE make utility. If `CL230x` is invoked with the name of a local makefile it will invoke DSMAKE with that makefile. After the make process is started, existing object files are not deleted by the makefile. Only source files which have been changed will be recompiled. Thus, if the application has to be translated with other compiler/assembler options, the object files must be deleted manually. The resulting application object file has the same name as the local makefile. The template makefile `tmpl230x.mk` can be used, to write a local makefile. It is in `<RCP_HIL_InstallationPath>\ds230x`.

If `CL230x.exe` is invoked with the name of an assembly/C source file, it uses the standard makefile `ds230x.mk` for the make process. An existing object file is deleted by the makefile so that it can use new assembler/compiler options. The resulting object file has the same name as the source file.

`CL230x.exe` calls `coffconv.exe`, which converts the object file to the SLC file format. The SLC file is used by the processor board loader function to load the slave application to the slave DSP.

Error messages

The following error messages are defined for the `cl230x` tool:

Message	Meaning
ERROR: not enough memory!	The attempt to allocate dynamic memory failed.
ERROR: environment variable TI_ROOT not found! Please open 'Command Prompt for RCP and HIL' and enter the following command to configure the compiler path: 'DsConfigTiEnv.exe'	The respective environment variable is not defined in the DOS environment. For more information, refer to How to Set the Compiler Path on page 42.
ERROR: environment variable DSPACE_ROOT not found!	The respective environment variable is not defined in the DOS environment.
ERROR: unable to access file <file_name>! ...	The specified file could not be accessed. Either another application has locked the file or the file does not exist.
ERROR: file <file_name> not found!	The specified file was not found.
ERROR: can't redirect stdout to file! ERROR: can't redirect stdout to screen!	Redirecting the standard output to a file or to the screen failed.
ERROR: can’t invoke ..\exe\dsmake!	Starting `Dsmake.exe` failed. Check whether `Dsmake.exe` is located in the given folder.
ERROR: making of <file_name> failed ERROR: assembling of <file_name> failed ERROR: compiling of <file_name> failed	An error occurred while executing a makefile, compiling, or assembling a source file. Refer to the standard output to get information on the error reason, for example, programming errors in the source file.

Related topics

HowTos

dummy.lk and ds230x.lk

Introduction

The linker command file defines where to place the STARTUP code and the different sections created by the C compiler in the VC33's memory and tells the linker which object modules and libraries to link.

Usually it is not necessary to change or customize the linker command files when building slave-DSP applications using the utility CL230x. Appropriate application specific linker command files are generated during the build process.

Two linker command files are available in the slave-DSP's software environment: `dummy.lk` and `ds230x.lk`

dummy.lk During the build process an appropriate linker command file is generated by the `GetSizeC3x.exe` utility. For further information on the `GetSizeC3x.exe` utility, refer to GetSizeC3x.exe on page 48.

To get information about the section sizes of the slave-DSP application, a dummy linking process is performed. The `dummy.lk` linker command file provides a virtual memory of 1 GB. The resulting map file of the dummy link is parsed by the `GetSiceC3x.exe` to get the required section sizes, before the application-specific linker command file is generated. The stack and heap size entries in the `dummy.lk` file are used as default values for the linker command file generation.

ds230x.lk If you need an individual memory layout for a particular application, you can use a local linker command file. The `ds230x.lk` file located in <RCP_HIL_InstallationPath>\DS230x can be used as a template to write a custom local linker command file. Local linker command files must be named by the file name prefix of the corresponding application C source file and the `.lk` suffix. CL230x finds a linker command file in the directory containing the application C source file, it is used for linking instead of a generated linker command file.

The linker command files contain several predefined memory configurations.

The following mentioned RAM blocks are internal memory of the TI DSP. The 2 KW small blocks RAM0 and RAM1 are available on both DSPs, the C31 and the VC33 DSP. If the size of the respective sections exceeds the size of RAM0 and RAM1, you can use RAMA instead. This assigns the sections to the internal memory RAM0 + RAM1 and ignores the internal memory block boundaries. This results in some lack of performance, since the OP code and operands can be accessed via a single data bus.

The DS2302-04 board's VC33 DSPs hold the additional memory blocks RAM2 and RAM3, which are provide 32-kWord additional RAM. As described above, memories RAM2 + RAM3 can be replaced by RAMB, which ignores the internal memory block boundaries.

You can assign the sections of your application to any RAM block except for the sections `.vectors` and `.trap`, which must always be assigned to the VECS and TRAP memory.

In generated linker command files appropriate sizes of stack and heap are chosen automatically, depending on the DSP version. If you want to use a custom, local linker command file, you can define additional options to increase the size of the heap and the stack in the linker command file, see example below. The stack is used for context saves, and local variables, and to pass parameters to functions. The heap is used for memory allocated with `malloc()`, for example, for dynamic data. The default sizes of the heap and the stack are 256 words for the DS2301-01. For the DS2302-04, you can completely assign the stack to RAM1 and the heap to RAM0.

Example: DS2302-01

```
-heap 0x0100           /* 256 byte heap size */
-stack 0x0100         /* 256 byte stack size */
```

Example: DS2302-04

```
-stack 0x03c1           /* 961 word stack */
-heap 0x03fe              /* 1022 word heap */
```

GetSizeC3x.exe

Introduction

The `GetSizeC3x` utility can evaluate a specified map file written by the Texas Instruments compiler during application build. The tool can be used for the DS2210, DS2211 and DS2302 slave-DSP software environment. The target board of the application is automatically detected by evaluating specific entries in the map file. The `GetSizeC3x` utility is used as a linker command file generator in the DS230x software environment, when the `CL230x` build utility is used.

`GetSizeC3x` reads the section size of the each section from the map file and checks whether all the sections can be assigned to the C31 / VC33 DSPs memory. Several variants of assigning the application to the memory are tested and displayed on the screen.

`GetSizeC3x` can also generate a linker command file that exactly matches the requirements of the application. It uses the first matching variant for the specified application. This utility automatically detects whether the application was built for a DS2302-04 board, which provides more memory than the DS2302-01 boards.

- DS2302-04:

  The sections of the application can be assigned to memory blocks RAM0, RAM1, RAM2, and RAM3.
- DS2302-01:

  Only memory blocks RAM0 and RAM1 are available for an application.

Command line options

The following command line options are available for `GetSizeC3x <map_file> [options] [/?]`:

Option	Meaning
/b <board_type>	Lets you select the board type (DS2302, DS2210, DS2211). If the option is omitted, the board type is automatically detected.
/g	Lets you generate a linker command file.
/h <size>	Custom .sysmem size in words. Enter size hexadecimal, for example, 0x200.
/n	No beep on error.
/o	Name of generated linker command file. The default is ds230x.lk.
/q	Quit mode

Option	Meaning
/s <size>	Custom .stack size in words. Enter the size in hexadecimal, for example, 0x200.
/?	Displays a list of the options available.

Example

When `GetSizeC3x` is invoked with the map file of the `fgen` application, it displays the size of each section and several variants of assigning them to the memory. The application of this example was compiled for DS2302-01 boards and only the RAM0 and RAM1 memory blocks are available. As you can see, the application fits into the memory, only with variant 3, where the memory boundaries of RAM0 and RAM1 are ignored. In variant 1 the RAM0 size was exceeded by 36 words and in variant 2 the size of RAM1 was exceeded by 97 words. `GetSizeC3x` has detected that the application does not need heap memory (no malloc used), so the .sysmem section is set to zero. After the application sections were assigned, all remaining memory was assigned to the .stack section.

```
getsizec3x fgen.map

GETSIZEC3x - C3x Application Memory Size Explorer, Vs 2.2 - 32, (C) 2008, dSPACE
GmbH

Memory assignment for C31 DSP of DS2302 board using fgen.map
--------------------------------------------------------------------------------

RAM A = RAM 0 + RAM 1

        ||   variant 1   ||   variant 2   ||variant 3
Section || RAM 0 | RAM 1 || RAM 0 | RAM 1 || RAM A
        ||0x003FE|0x003C1||0x003FE|0x003C1||0x007BF
=======================================================
.startup||0x00012|       ||       |0x00012||0x00012
.cinit  ||0x0007B|       ||       |0x0007B||0x0007B
.text   ||0x00395|       ||       |0x00395||0x00395
.const  ||       |0x0001C||0x0001C|       ||0x0001C
.data   ||       |0x00000||0x00000|       ||0x00000
.bss    ||       |0x0006A||0x0006A|       ||0x0006A
.stack  ||       |0x00317||0x00317|       ||0x00317
.sysmem ||       |0x00000||0x00000|       ||0x00000
=======================================================
free    ||    -36|     36||     97|    -97||      0
```

If you use the additional command line switch "-g", the `ds230x.lk` linker command file is generated according to variant 3, which can be used for application build as performed by the `CL230x` utility.

```
getsizec3x fgen.map -g
```

Software Reference

Where to go from here

Information in this section

Introduction to the Functions and Macros

Where to go from here

Information in this section

Overview of the DS2302 Programming Environment

Introduction

To make it easy to implement user application programs on the DS2302, operations that are frequently needed, such as system initialization or I/O access, have been put into macros or library functions. The resulting object library and header files form the basic software environment that comes with the board.

Library functions

Interface libraries allow access to the DS2302 via PHS bus by a processor board.

The basic software environment for the DS2302 DSPs contains macros and functions to perform the system initialization, to access the on-board peripherals and to control interrupt operations. It is copied to the directory <RCP_HIL_InstallationPath>\ds230x during software installation.

Operations that are not used in time-critical program parts are implemented as functions that have been put into the `ds230x.lib` object library. This library must be linked to every application program, which is automatically done if the standard compile/link tools are used.

Header files

Time-critical operations, such as I/O access, are implemented as macros that can be found in the `ds2302.h`, `ds2302_advanced.h`, and `util2302.h` header files. The `ds2302.h` header file contains basic board definitions, function prototypes for the object library `ds230x.lib`, and macro definitions. The `ds2302_advanced.h` header file contains macros and function prototypes only usable on the new DS2302-04 board. It also causes the build process to use the additional 32 KW memory of the new VC33 DSP. The `util2302.h` header file contains additional board definitions for dual-port memory access and macro definitions for execution time measurement (for example, by using SPEEDCHK).

Several other header files described below, provides access to the DMA controller, to the synchronous serial port, the message module or to the execution time measurement.

Function Overview

Introduction

This topic gives you an overview of all the functions and macros contained in the basic DS2302 software environment.

Initialization functions

Function	Refer to
init	Initialization Functions on page 59
timer0	
timer1	
timer0_sync	
ds2302_dsp_clock_init	

I/O functions

Function	Refer to
dac_out	Analog Output via D/A Converter Module on page 62
init_dig_out<n>[1)] dig_out<n>[1)] dig_in<n>[1)]	Digital I/O on page 63

[1)] <n> is 1, 2, …, 7

Interrupt control functions

Function	Refer to
disable_int<n>[1)] enable_int<n>[1)] int<n>_ack[1)] int<n>_init[1)] int<n>_pending[1)]	Interrupt Control on page 69

Function	Refer to
int<n>_status[1] int<n>_aux_status[1]	
disable_tint<m>[2] enable_tint<m>[2]	
global_disable global_enable	
phs_bus_interrupt_request	
int_xf<m>[2]	

[1] <n> is 0, 1, or 3

[2] <m> is 0 or 1

Accessing the DMA controller

Function	Refer to
dma_init	Accessing the DMA Controller on page 72
dma_stop	
dma_stop_when_finished	
dma_restart	
dma_reset	
dma_interrupt_enable	
dma_interrupt_disable	

Accessing the serial interface

Function	Refer to
serial_init_std_handshake	Accessing the Serial Interface on page 79
serial_init_ds2302	
serial_init	
serial_disable	
serial_rx_int_init	
serial_tx_int_init	
serial_tx_int_start	
disable_rx_int, disable_tx_int	
enable_rx_int, enable_tx_int	
serial_tx_word_poll	
serial_tx_word_int	
serial_rx_word_poll	
serial_rx_word_int	

Floating-Point format conversion

Function	Refer to
cvtie3	Converting the Floating-Point Format on page 96
cvtdsp	

Status LED access

Function	Refer to ...
led_state	Accessing the LED Function on page 99

Execution time profiling

Function	Refer to
timer_count	Measuring Execution Times with the util2302.h Module on page 102
time_elapsed	
void tic<m>_init(void)[1] void tic<m>_start(void)[1] void tic<m>_halt(void)[1] void tic<m>_continue(void)[1] float tic<m>_read(void)[1] float tic<m>_read_total(void)[1] void tic<m>_delay(float duration)[1]	Measuring Execution Times with the tic3x.h Module on page 103

[1] <m> is 0 or 1

Identifiers and Declarations

Where to go from here

Information in this section

Identifiers for Numerical Constants

Identifiers

The identifiers listed below are defined in the *ds2302.h* header file. These definitions are already used by the standard functions and macros described in the sections System Initialization on page 59 to Interrupts on page 66. They can also be used in user applications.

Identifier	Value	Meaning
SCAL	2147483648.0	Scaling factor for D/A output values.
TIMER_CLOCK	(clock_per_sec/2.0)	Timer clock rate of DS2302 board.
IOCTL_ADDR	0x700000	Address of the on-board I/O control register (IOCTL).
IOSTS_ADDR	0x600000	Address of the on-board I/O status register (IOSTS).
IOMOD_ADDR	0x500000	Base address of the module port.
SPEEDCHK_MAX	0x603FFD	Address of the maximum execution time for the speed_check macro.
SPEEDCHK_MIN	0x603FFC	Address of the minimum execution time for the speed_check macro.
INT_TBL_ADDR	0x809FC0	Start address of the interrupt vector table.
INT3_ADDR	0x403FFF	Reserved dual-port memory location for host interrupt INT3 (as seen by the DS2302 on-board DSP's).
DSPINT0	0x40000000	INT0 and INT1 interrupt status / end-of-interrupt bits in the IOCTL register.
DSPINT1	0x80000000	
INT3	0x00000008	INT3 interrupt status bit in the VC33's IF register.
INT0STS	0x40000000	INT0 interrupt status bit in the on-board IOSTS register.
INT0AUX	0x80000000	Auxiliary INT0 interrupt status bit in the on-board IOSTS register.
ioctl	((volatile long *) 0x00700000)	Pointer to the on-board IOCTL register.
iosts	((volatile long *) 0x00600000)	Pointer to the on-board IOSTS register.
int3	((volatile long *) 0x00403FFF)	Pointer to the dual-port memory location 0x00403FFF reserved for the host interrupt INT3.

Identifier	Value	Meaning
dp_mem	((volatile float_or_int *) 0x00400000)	Pointers to the local DPMEM of channels 1 … 6. The pointer is of union type to transfer floating-point values as well as integer values. For example, the memory location j is accessed by dp_mem[j].f for a value of type float or dp_mem[j].i for an integer value.
dp_mem_1	((volatile float_or_int *) 0x00040000)	
dp_mem_2	((volatile float_or_int *) 0x000C0000)	
dp_mem_3	((volatile float_or_int *) 0x00140000)	
dp_mem_4	((volatile float_or_int *) 0x001C0000)	
dp_mem_5	((volatile float_or_int *) 0x00240000)	
dp_mem_6	((volatile float_or_int *) 0x002C0000)	
DP_MEM_BASE	0x400000	Base addresses of the local dual-port memory and the dual-port memories of channels 1 … 6. At location DP_MEM_ALL_BASE, all 6 channels can be written simultaneously.
DP_MEM_1_BASE	0x040000	
DP_MEM_2_BASE	0x0C0000	
DP_MEM_3_BASE	0x140000	
DP_MEM_4_BASE	0x1C0000	
DP_MEM_5_BASE	0x240000	
DP_MEM_6_BASE	0x2C0000	
DP_MEM_ALL_BASE	0x3C0000	
DP_MEM_SIZE	0x4000	Size of the DPMEM.

Predefined Pointer Declarations and Global Variables

Introduction

Some pointer declarations and global variables are predefined.

Predefined pointer declarations and global variables

The following declarations of global pointers and variables are defined in the `init.c` file. These variables are already used by the standard functions and macros described in sections System Initialization on page 59 and Execution Time Profiling on page 102. They can also be used in user applications.

Note

Each initialized pointer requires 3 words in the .cinit section and an additional word in the .bss section.

Pointer	Meaning
`long *dac`	Pointer to the DAC output register (module port offset 0)
`long *int_tbl`	Pointer to the interrupt vector table
`float clock_per_sec`	Slave DSP clock frequency
`float time_per_tick`	Slave DSP timer period (2.0/clock_per_sec)

Pointer	Meaning
`long _board`	DDS board type ▪ 230201: for DS2302-01 boards with 60 MHz ▪ 230204: for DS2302-04 boards with 150 MHz

System Initialization

Where to go from here

Information in this section

Startup Code

Introduction

The first code executed after a program start is called the startup code.

Startup code

The startup code is contained in the `startup.asm` file. The corresponding object code is included in the `ds230x.lib` object library. The startup code initializes the interrupt and TRAP vectors and the DSP's primary bus control register. Program execution proceeds at the C run-time entry point `c_int00`. This code is automatically linked to every application program.

Initialization Functions

Introduction

Before you can use functions of a DS2302 board, it must be initialized.

Initialization functions

The table below lists the initialization functions:

Syntax	Description	Parameter
`void init (void)`	The `init()` function initializes the slave DSP as follows: ▪ Reset the hardware system ▪ Clear pending interrupts ▪ Initialize global pointers and variables ▪ Detects DS2302 board revision number	–

Syntax	Description	Parameter
	▪ Initializes DSP clock This function must be called at the beginning of the `main()` routine in every user application program.	
`void timer0 (float time)` `void timer1 (float time)`	The on-chip timer0 or timer1 of the DSP is initialized to generate timer interrupts at the sampling rate specified by the `time` parameter. The appropriate interrupt vector is set to point to the corresponding timer interrupt service routine `c_int09()` for timer0 and `c_int10()` for timer1. The corresponding timer interrupt TINT0 or TINT1 is enabled in the interrupt enable register (IE) of the DSP and interrupts are enabled globally in the VC33's status register (ST). After either `timer0()` or `timer1()` is invoked, it immediately starts to generate timer interrupts at the specified sampling rate.	`time` Required sampling period in seconds at which timer interrupts must be issued.
`void timer0_sync (float time)`	The on-chip timer0 of the DSP is initialized to generate timer interrupts at the sampling rate specified by the `time` parameter. The appropriate interrupt vector is set to point to the corresponding timer interrupt service routine `c_int09()`. Timer0 is started and the corresponding interrupt TINT0 is enabled by a global host interrupt INT2 to start multiple DS2302 channels simultaneously. After invocation of `timer0_sync()`, no timer0 interrupts are generated until an INT2 interrupt has been received. An INT2 interrupt is generated by the DS2302 program loader or the master processor board load facility after the download is complete. All DS2302 channels using this function for timer initialization will simultaneously start to generate timer interrupts.	

Switching DSP Clock

Introduction

The DS2302 (starting from board revision DS2302-04) allows the input clock of the DSP to be changed. Changing the DSP clock will change the cycle time and therefore the execution time.

Switching DSP clock

The fastest DSP clock (150 MHz) is initialized automatically in the `init()` function. Refer to Initialization Functions on page 59. Using the `ds2302_dsp_clock_init` function is only necessary if you want to use the DSP's serial port to match certain transmission frequencies.

Call the `ds2302_dsp_clock_init` function before you call timer initialization functions (`timer0()`, `timer1()`, or `timer0_sync()`). Otherwise you must call the timer initialization functions again to consider the new DSP-clock value.

The following table shows the function for switching the DSP clock.

Syntax	Description	Parameter
`void ds2302_dsp_clock_init (long mode)`	Lets you change the input clock of the DSP. The new clock becomes valid after a delay of 2.2 µs.	`mode` ▪ DS2302_PLL_60 = 60 MHz ▪ DS2302_PLL_75 = 75 MHz ▪ DS2302_PLL_100 = 100 MHz ▪ DS2302_PLL_150 = 150 MHz

Example

The example shows how to use the function.

```
init();
/* set DSP clock to 75 MHz */
ds2302_dsp_clock_init(DS2302_PLL_75);
/* initialize timer0 */
timer0(TS);
```

Accessing On-Board I/O

Where to go from here

Information in this section

Analog Output via D/A Converter Module

Basics

Analog output using the D/A converter (DAC) can be performed by simply writing to the predefined address `*dac`. The `*dac` pointer is defined in the `ds2302.h` header file. The analog output signal of each channel is available at the DS2302's I/O connector (P4) pins OUT A … OUT F.

The 16-bit DAC is connected to data bus bits 16 to 31. The value being written to the DAC must be in the 16-bit signed integer range -2.14748365E9 … +2.14748365E9 corresponding to the DAC's full analog output range of ±10 V. Thus floating-point values must be scaled to the proper range and cast to a long integer type before they can be written to the DAC output register. The scaling factor for the DAC value is defined as `SCAL` in the `ds2302.h` header file.

Example

```
float y;

y = ...
*dac = (long) (SCAL * y);
```

If a floating-point value in the range (-1.0 ≤ value < +1.0) has to be scaled to the full DAC range and written to the DAC, as in the example above, you can also use the predefined `dac_out()` macro.

Macro

The table below shows the `void dac_out(float value)` macro:

Syntax	Description	Parameter
`void dac_out(float value)`	The output `value` parameter is scaled by the factor 2.147483648E9 and cast to the `long integer` type before it is written to the DAC output register.	`value` The `value` parameter is output to the DAC. It must be in the range (-1.0 <= *value* < +1.0).

Further information

For more information on the DAC Module, refer to DAC Module Data Sheet (PHS Bus System Hardware Reference 🕮).

Digital I/O

Introduction

The DS2302 board supports up to 8 digital I/O lines per channel via the timer and serial port pins of the DSP. Each line can be configured as an input or output.

Initialization, input, and output

The `ds2302.h` header file contains macros to initialize, input and output data.

Use the `init_dig_out1()` ... `init_dig_out7()` macros to initialize the digital I/O lines for input or output.

Use the `dig_out1()` ... `dig_out7()` macros to output data to the respective I/O lines.

Use the `dig_in1()` ... `dig_in7()` macros to input data from the respective I/O lines.

For the pin layout of the I/O lines, refer to Signal Mapping to I/O Pins (PHS Bus System Hardware Reference 🕮).

Macros and DS2302 signals

The 6 digital I/P pins DIG_x1 .. DIG_x5 are shared with the serial port of the VC33 DSP. If using the serial port for transmission, the I/O functions of the mentioned pins must not be used. The macros relate to the digital output lines as listed in the table below:

Macro	DSP Signal	DS2302 Signal

`init_dig_out1()`
`dig_out1()`
`dig_in1()`
	DX0	DIG_x1 [1)]

Macro	DSP Signal	DS2302 Signal
init_dig_out2() dig_out2() dig_in2()	FSX0	DIG_x2 [1]
init_dig_out3() dig_out3() dig_in3()	CLKR0	DIG_x3 [1]
init_dig_out4() dig_out4() dig_in4()	DR0	DIG_x4 [1]
init_dig_out5() dig_out5() dig_in5()	FSR0	DIG_x5 [1]
init_dig_out6() dig_out6() dig_in6()	TCLK0	DIG_x6 [1]
init_dig_out7() dig_out7() dig_in7()	TCLK1/INT0	INT_x0 [1]

[1] x means: Channel indicator (A, B, C, D, E, F)

Note

dig_out7() uses the INT0 line. Changing levels on this line will set the corresponding INT0 bit in the IF and the IOSTS register even if this line is configured for digital I/O. Ensure that INT0 is disabled in this case.

Macro description

The macros are described in the table below:

Syntax	Description	Parameter
void init_dig_out1 (long value) ... void init_dig_out7 (long value)	The init_dig_out<n>() macros initialize the respective TMS320VC33 pin for input or output. If value = 0 the port is initialized as an input. If value = 1 the port is set to output mode.	value The value parameter initializes the I/O port for input or output. ▪ 0 = input ▪ 1 = output
void dig_out1(long value) ... void dig_out7(long value)	The respective digital I/O line is set to 0 or 1 depending on the value parameter. Note that dig_out7() uses the INT0 line. Changing levels on this line will set the corresponding INT0 bit in the IF and the IOSTS register, although this line is used for digital I/O. Ensure that INT0 is disabled in this case. The init_dig_out<n>(1) macros must be called to configure the I/O line for output, before this macro can be used.	value The value parameter specifies the state of the digital output line. It must be 0 or 1.

Syntax	Description	Parameter

`long dig_in1(void)`
…
`long dig_in7(void)`
	This macro returns the state of the respective digital I/O line.
	The `init_dig_out<n>(0)` macro must be called to configure the I/O line for input, before this macro can be used.
	The return values are 0 or 1.
		–

Interrupts

Introduction

Each of the DSP contained on the DS2302 board can be interrupted by an external interrupt or a master processor board, or by any other slave DSP.

Where to go from here

Information in this section

Basics of Interrupts

Introduction

Each of the DSP contained on the DS2302 board can be interrupted by an external interrupt or a master processor board, or by any other on-board DSP.

Basics

The VC33's interrupt INT0 is used for external interrupts. The corresponding interrupt line for each channel is available at the DS2302's I/O connector pins INT0 A .. INT0 F.

Interrupts from another on-board DSP are received by the INT1 interrupt. The INT1 interrupt source for each channel (the XF0 or XF1 flag of another on-board DSP) is selected by the interrupt select register.

The interrupt INT2 is used by the program loader and the master processor board load facility to synchronously start sampling clock interrupt generation on multiple DS2302 channels in conjunction with the timer initialization function `timer0_sync()`. This interrupt cannot be used for other purposes and you must not initialize it yourself. For further information on the `timer0_sync()` function, refer to Initialization Functions on page 59.

The master processor board can interrupt an individual DS2302 DSP by writing to the predefined last dual-port memory location at offset 3FFFH. This requests an INT3 interrupt on the respective DSP. The value written to the dual-port memory can be used for interrupt-driven data transfer, etc.

The table below gives an overview of all interrupts supported by the DS2302.

Interrupt	**Service Routine Name**		**Description**
	Internal Name	**Alias**	
INT0	`c_int01()`	`isr_int0()`	External interrupt triggered by the external interrupt line on the I/O connector. Initialization is performed by the `int0_init()` function.[1)]
INT1	`c_int02()`	`isr_int1()`	Interrupt from other on-board DSP (selected by interrupt setup, see Interrupt Selection on page 71). This interrupt is initialized by the `int1_init()` function.[1)]
INT2	`c_int03()`	`isr_int2()`	Reserved for synchronous timer start by the `timer0_sync()` function. You must not initialize this interrupt.[1)]
INT3	`c_int04()`	`isr_int3()`	Host interrupt triggered by writing to the dual-port memory location at offset 3FFFH. It is initialized by the `int3_init()` function.[1)]
XINT0	`c_int05()`	`isr_transmit()`	Transmit interrupt of DSP's serial interface. It is initialized by the `serial_tx_int_init()` function.[2)]
RINT0	`c_int06()`	`isr_receive()`	Receive interrupt of the DSP's serial interface. It is initialized by the `serial_rx_int_init()` function.[3)]
TINT0	`c_int09()`	`isr_t0()`	Interrupt for the built-in timer0. It can be used to generate sampling clock interrupts and is initialized by `timer0()` or `timer0_sync()` function.[1)]
TINT1	`c_int10()`	`isr_t1()`	Interrupt for the built-in timer1. It can be used to generate sampling clock interrupts and is initialized by the `timer1()` function.[1)]

[1)] For details, refer to Initialization Functions on page 59.
[2)] For details, refer to `serial_tx_int_init` on page 88.
[3)] For details, refer to `serial_rx_int_init` on page 87.

The interrupts INT0, INT1, and INT3 can be served in two different ways. The simplest method is to poll the corresponding interrupt flag in the DS2302's IOCTL register or in the VC33's interrupt flag register (IF). In this case no interrupt service routine must be implemented, so the interrupt does not have to be initialized. You can use the appropriate macro `int0_pending()`, `int1_pending()`, or `int3_pending()` to poll the interrupt flag. After an interrupt has been received, you must clear the interrupt flags in the DS2302's IOCTL register and in the VC33's IF register, e.g. by using the appropriate macro `int0_ack()`, `int1_ack()`, or `int3_ack()`.

Example:

```
if (int0_pending())        /* test for pending interrupt INT0 */
{
  ...                              /* perform interrupt service */
  int0_ack();                     /* acknowledge interrupt INT0 */
}
```

The second method uses an interrupt service routine to serve a requested interrupt. The appropriate initialization function `int0_init()`, `int1_init()`, or `int3_init()` must be called in the initialization part of the application program. You must supply an interrupt service routine for the interrupt. The interrupt service routine must clear the interrupt flags in the DS2302's IOCTL register and in the VC33's IF register, e.g. by using the appropriate macro `int0_ack()`, `int1_ack()`, or `int3_ack()`. In most cases this method is much better because no time is wasted in polling the interrupt flag.

Example:

```
void c_int01()             /* INT0 interrupt service routine */
{
  ...                           /* perform interrupt service */
  int0_ack();                  /* acknowledge interrupt INT0 */
}

main()
{
  ...
  int0_init();         /* initialize INT0 interrupt service */
  ...
}
```

Note

Interrupt service routines must use the naming conventions in the *TMS320 Floating-Point DSP Optimizing C Compiler User's Guide* from Texas Instruments. As an alternative you can use the alias names listed in the table above. They are defined in the `ds2302.h` header file.

Interrupt Initialization

Introduction

The functions described below perform initialization of the interrupt vector in the interrupt vector table and enable the interrupt.

Syntax	Description	Parameter

`void int0_init (void)`
`void int1_init (void)`
`void int3_init (void)`
	The interrupt vector for interrupt INT0, INT1 or INT3 is set to point to the corresponding interrupt service routine `c_int01()`, `c_int02()` or `c_int04()`. The interrupt is	–

Syntax	Description	Parameter
	enabled in the VC33's interrupt enable register (IE) and interrupts are enabled globally in the VC33's status register (ST). This function must be called before the interrupt can be served by an interrupt service routine. Note that you must provide the interrupt service routine. Otherwise the linker detects an unresolved external reference. You must also clear the interrupt flags in the DS2302's IOCTL register and in the VC33's IF register at the end of the interrupt service routine, for example, by using the `int0_ack()`, `int1_ack()` or `int3_ack()` macro.	

Interrupt Control

Introduction

The macros discussed in this section are needed to control interrupt operation, i.e. to request interrupts on other DS2302 channels, to acknowledge interrupts after interrupt service has finished, to test interrupt flags, and to enable/disable individual interrupts.

Syntax	Description	Parameter

`void disable_int0(void)`
`void disable_int1(void)`
`void disable_int3(void)`
`void disable_tint0(void)`
`void disable_tint1(void)`
	The interrupt enable bit for an individual interrupt INT0, INT1, INT3, TINT0, or TINT1 is cleared in the TMS320VC33's IE register to disable the corresponding interrupt.	–

`void enable_int0(void)`
`void enable_int1(void)`
`void enable_int3(void)`
`void enable_tint0(void)`
`void enable_tint1(void)`
	The interrupt enable bit for an individual interrupt INT0, INT1, INT3, TINT0, or TINT1 is set in the TMS320VC33's IE register to enable the corresponding interrupt.	–

`void global_disable(void)`	The global interrupt enable bit (GIE) in the TMS320VC33's status register (ST) is cleared to disable interrupts globally.	–
`void global_enable(void)`	The global interrupt enable bit (GIE) in the TMS320VC33's status register (ST) is set to enable interrupts globally.	–
`void int0_ack(void)` `void int1_ack(void)`	The interrupt bit DSPINT0 or DSPINT1 in the on-board IOCTL register and the corresponding interrupt flag in the TMS320VC33's IF register are cleared. These macros can be used to acknowledge an	–

Syntax	Description	Parameter
	interrupt request after an INT0 or INT1 interrupt service finishes.	
`void int3_ack(long value)`	The INT3 interrupt flag in the TMS320VC33's IF register is cleared. The parameter value returns the data value that has been written to the reserved dual-port memory location at offset 0x03FFF by a master processor board to request an INT3 interrupt. These macros can be used to acknowledge an interrupt request after an INT3 interrupt service finishes. Note that the local dual-port memory address 0x403FFF as seen by the DSP corresponds to the address offset 0x3FFF as seen by the master processor.	`value` The `value` parameter returns the contents of the dual-port memory location 0x403FFF (as seen by the DSP).
`long int0_pending(void)` `long int1_pending(void)`	The state of the DSPINT0 or DSPINT1 interrupt bit in the on-board IOCTL register is returned to indicate whether an interrupt is pending (DSPINTx = 1) or not (DSPINTx = 0). This macro can be used if an interrupt request has to be served without an installed interrupt service routine. Simply poll the interrupt flag in the IOCTL register.	Return parameter: ▪ 0: No interrupt is pending ▪ 1: Interrupt is pending
`void int3_pending(long state)`	The state of the INT3 interrupt flag in the VC33's IF register is returned through the parameter *state* to indicate whether an interrupt is pending or not. This macro can be used if an interrupt request has to be served without an installed interrupt service routine. Poll the interrupt flag in the VC33's IF register.	`state` ▪ 0 = interrupt is not pending ▪ 1 = interrupt is pending
`void int_xf0(void)` `void int_xf1(void)`	A low pulse is generated on the TMS320VC33's flags XF0 or XF1 to request an INT1 interrupt on other on-board DSPs. The relations between the XF0 and XF1 flags and the INT1 interrupt lines of each on-board DSP are specified in the interrupt select register by the master processor, or by host utilities such as ControlDesk.	–
`long int0_status(void)`	The state of the external input line INT0, monitored in the IOSTS register, is returned to indicate whether the input is high or low. The return values are 0 or 1.	–
`long int0_aux_status(void)`	The state of the external input INT0 from the adjacent channel, monitored in the IOSTS register, is returned to indicate whether the	–

Syntax	Description	Parameter
	input is high or low. Refer to section I/O Status Register (IOSTS) on page 162. The return values are 0 or 1.	
`void phs_bus_interrupt_request(void)`	A low pulse is generated on the TMS320VC33's CLKX0 pin to request an PHS bus interrupt. For proper operation the DS2302 PHS bus interrupt of the corresponding channel must be initialized on the master processor board.	–

Interrupt Selection

Introduction

A DSP can be interrupted by another slave DSP. The interrupt setup selects the interrupt source for a DSP. The interrupt setup can be specified with the `int_mask` parameter in the `ds2302_load_board` function in a handcoded application and the DS2302_DSP_SETUP_Bx block in a Simulink model.

Possible interrupt combination

Interrupts from another on-board DSP are received by the INT1 interrupt. The INT1 interrupt source for each channel (the XF0 or XF1 flag of another on-board DSP) is selected by the interrupt setup. The following table shows the valid combinations of interrupt sources for a DSP.

Interrupted DSP	INT1 Interrupt Source											
	XF0 Line of DSP x						XF1 Line of DSP x					
	1	2	3	4	5	6	1	2	3	4	5	6
1	n. a.	✓	✓	✓	✓	✓	n. a.	✓	✓	✓	–	–
2	✓	n. a.	✓	✓	✓	✓	–	n. a.	✓	✓	✓	–
3	✓	✓	n. a.	✓	✓	✓	–	–	n. a.	✓	✓	✓
4	✓	✓	✓	n. a.	✓	✓	✓	–	–	n. a.	✓	✓
5	✓	✓	✓	✓	n. a.	✓	✓	✓	–	–	n. a.	✓
6	✓	✓	✓	✓	✓	n. a.	✓	✓	✓	–	–	n. a.

Related topics

References

DS2302_DSP_SETUP_Bx (DS2302 RTI Reference 📖)
ds2302_load_board (DS2302 RTLib Reference 📖)

Accessing the DMA Controller

Where to go from here

Information in this section

dma_init

Syntax

```
void dma_init(
      unsigned long src_addr,
      unsigned long dst_addr,
      unsigned long count,
      unsigned int src_mode,
      unsigned int dst_mode,
      unsigned int int_sync,
      unsigned int tc,
      unsigned int tcint)
```

Include file

`Dma31.h`

Purpose

To initialize and start the DMA controller of the slave DSP.

Description

Use `dma_stop` or `dma_stop_when_finished` to stop the DMA controller.

Parameters

src_addr Address of the source data to be transferred by the DMA controller

dst_addr Destination address to which the data will be transferred

count Number of words to be transferred in the range 1 … 16,777,215

src_mode Mode of source address modification. The following symbols are predefined:

Predefined Symbol	Meaning
`DMA_NO_MODIFY`	The source address is not modified.
`DMA_INCREMENT`	The source address is incremented after each DMA read access.
`DMA_DECREMENT`	The source address is decremented after each DMA read access.

dst_mode Mode of destination address modification. The following symbols are predefined:

Predefined Symbol	Meaning
`DMA_NO_MODIFY`	The destination address is not modified.
`DMA_INCREMENT`	The destination address is incremented after each DMA write access.
`DMA_DECREMENT`	The destination address is decremented after each DMA write access.

int_sync DMA synchronization mode. The following symbols are predefined:

Predefined Symbol	Meaning
`DMA_NO_SYNC`	No synchronization
`DMA_SRC_SYNC`	Source synchronization. This means that a read access is performed when a DMA interrupt occurs.
`DMA_DST_SYNC`	Destination synchronization. This means that a write access is performed when a DMA interrupt occurs.

tc DMA transfer mode. The following symbols are predefined:

Predefined Symbol	Meaning
`DMA_CONTINUOUS`	Transfer restarts when the specified number of words has been transferred.
`DMA_TERMINATE`	Transfer is terminated when the specified number of words has been transferred.

tcint Sets the mode for the DMA to CPU interrupt. The following symbols are predefined:

Predefined Symbol	Meaning
`DMA_TCINT_DISABLE`	No DMA interrupt is generated when the transfer has finished.

Predefined Symbol	Meaning
DMA_TCINT_ENABLE	A DMA interrupt is generated when the transfer has finished.

Return value

None

Related topics

References

dma_stop

Macro

```
void dma_stop
```

Include file

`Dma31.h`

Purpose

To stop the DMA controller.

Description

The current word read or write operation is completed.

Return value

None

Related topics

References

dma_stop_when_finished

Macro	`void dma_stop_when_finished`
Include file	`Dma31.h`
Purpose	To stop the DMA controller when the entire transfer has been completed.
Return value	None
Related topics	References

dma_restart.. 75
dma_stop.. 74

dma_restart

Macro	`void dma_restart`
Include file	`Dma31.h`
Purpose	To restart the DMA controller from reset or a previous state.
Return value	None
Related topics	References

dma_stop.. 74
dma_stop_when_finished... 75

dma_reset

Macro	`void dma_reset`
Include file	`Dma31.h`
Purpose	To reset the DMA controller.
Return value	None
Related topics	References

dma_interrupt_enable

Syntax	`void dma_interrupt_enable(unsigned long mask)`
Include file	`Dma31.h`
Purpose	To enable the external DMA interrupts.
Description	The interrupt sources of the slave DSP are connected to the CPU and to the DMA controller. To enable a DMA interrupt, the respective interrupt enable flag is set in the IE register of the slave DSP.

Parameters

mask Interrupt(s) to be enabled. The following symbols are predefined:

Predefined Symbol	Meaning
`DMA_EINT0`	External interrupt INT0
`DMA_EINT1`	External interrupt INT1
`DMA_EINT2`	External interrupt INT2
`DMA_EINT3`	External interrupt INT3

Predefined Symbol	Meaning
DMA_EXINT0	Serial interface transmit interrupt
DMA_ERINT0	Serial interface receive interrupt
DMA_ETINT0	Timer0 interrupt
DMA_ETINT1	Timer1 interrupt
DMA_EDINT	DMA controller interrupt

Return value

None

Related topics

References

dma_interrupt_disable.. 77

dma_interrupt_disable

Syntax

```
void dma_interrupt_disable(unsigned long mask)
```

Include file

`Dma31.h`

Purpose

To disable the external DMA interrupts.

Description

The interrupt sources of the slave DSP are connected to the CPU and to the DMA controller. To disable a DMA interrupt, the respective interrupt enable flag is cleared in the IE register of the slave DSP.

Parameters

mask Interrupt(s) to be disabled. The following symbols are predefined:

Predefined Symbol	Meaning
DMA_EINT0	External interrupt INT0
DMA_EINT1	External interrupt INT1
DMA_EINT2	External interrupt INT2
DMA_EINT3	External interrupt INT3
DMA_EXINT0	Serial interface transmit interrupt
DMA_ERINT0	Serial interface receive interrupt

Predefined Symbol	Meaning
DMA_ETINT0	Timer0 interrupt
DMA_ETINT1	Timer1 interrupt
DMA_EDINT	DMA controller interrupt

Return value

None

Related topics

References

dma_interrupt_enable... 76

Accessing the Serial Interface

Where to go from here

Information in this section

Basics of the Serial Interface

Basics

The DS2302 board's VC33 DSP contains a bidirectional serial interface. This interface allows you to connect a DS2302 board, a DS2210 board or a DS2211 board.

The connection provides a frequency of up to 10 MHz. The handshake mode is used for serial transmission.

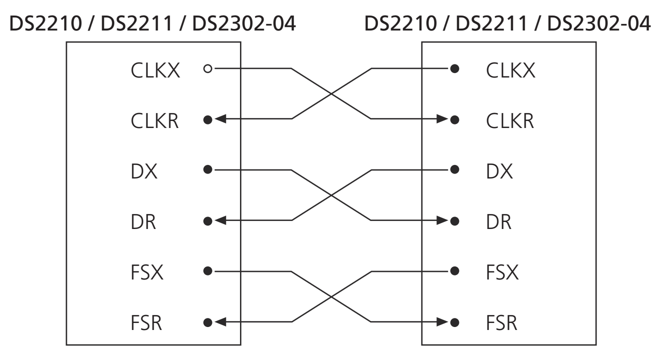
> **Note**
>
> If the serial port is initialized and used for data transmission, the digital I/O access macros (see Digital I/O on page 63) must not be used. Otherwise serial transmission will fail.

Further information

For more information on the serial interface, refer to the Texas Instruments Web site at http://www.ti.com and search for "TMS320VC33".

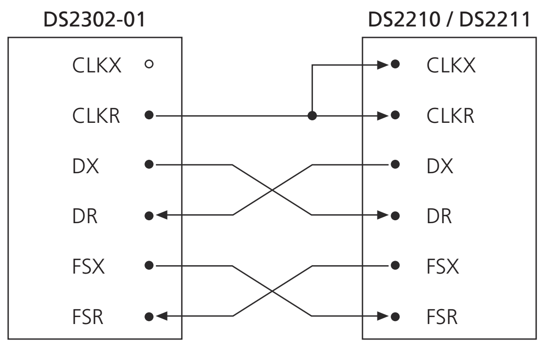
Connection schemes

The following illustration shows the scheme for DS2210 / DS2302-04 / DS2211 connections in handshake mode.



The following illustration shows the scheme for the old DS2302-01 / DS2211 connection scheme connections in handshake mode.

> **Note**
>
> On the DS2302-01, the serial interface pin CLKX0 is not available for serial transmission.



Example of Using the Serial Interface of the Slave DSP

Introduction

The examples show serial communication in polling mode and interrupt-driven mode.

Example (polling mode)

The following example shows serial communication in polling mode. The serial interface is initialized for the standard handshake mode. Transmission is performed at a frequency of 10.0 MHz for a connection to a DS2210 board running at 80 MHz.

The `serial_rx_word_poll` receive function is invoked until one data word is received. After that, the `serial_tx_word_poll` function is invoked until the data word is transmitted successfully.

```
void main(void)
{
   long data;
   /* initialize hardware system */
   init();
   /* initialize the serial interface */
   serial_init_std_handshake(1);
   /* receive data */
   while(serial_rx_word_poll((long *)&data) != SP_TRUE );
   /* transmit data */
   while(serial_tx_word_poll((long *)&data) != SP_TRUE );
}
```

Example (interrupt-driven mode)

The following example shows serial transmission in interrupt-driven mode. The serial interface is initialized for the standard handshake mode. Transmission is performed at a frequency of 10.0 MHz for a connection to a DS2210 board running at 80 MHz.

The transmit and the receive interrupts are initialized. The received data is stored in the data array `receive_data[]` by the `serial_rx_word_int` function. The data to be transmitted is available in the `transmit_data[]` array. To start interrupt-driven transmission, the `serial_tx_int_start` macro must be invoked. Each time the serial interface has sent a data word and is ready to send the next data word, a new transmit interrupt is requested. After the 100 data values are sent with the `serial_tx_word_int` function, transmission stops. No data is sent in the last execution of the transmit interrupt service routine due to x ≤ 100. To restart sending, the index x must be set to 0 and the `serial_tx_int_start` macro must be called again.

```
long transmit_data[100];
long receive_data[100];
void isr_receive() /* receive interrupt service routine */
{
   if(i >= 100)
      i = 0;
   serial_rx_word_int((long *)&receive_data[i++]);
}
void isr_transmit() /* transmit interrupt service routine */
{
   if(x < 100)
      serial_tx_word_int((long *)&transmit_data[x++]);
}
void main(void)
{
   /* initialize hardware system */
   init();
   /* initialize the serial interface */
   serial_init_std_handshake(1);
   /* initialize receive interrupt */
   serial_rx_int_init();
   /* initialize transmit interrupt */
   serial_tx_int_init();
   /* start transmission */
   serial_tx_int_start();
...
}
```

Related topics

Basics

References

serial_init_std_handshake

Macro

```
void serial_init_std_handshake(unsigned long timer_prd)
```

Include file

`Ser31.h`

Purpose

To initialize the slave DSP's serial interface for data transfer in handshake mode for the following connections:

- DS2211 – DS2210
- DS2211 – DS2211
- DS2302-04 – DS2210
- DS2302-04 – DS2211
- DS2302-04 – DS2302-04

Description

This macro automatically calls **`serial_init`** with the required parameters. For a connection to a DS2302-01, refer to serial_init_ds2302 on page 84.

> **Note**
>
> The receiving frequency via the DR0 input depends on the setting of the opposite transmitting serial interface.

Parameters

timer_prd Frequency of the serial transmission via the DX0 output. The valid values are 0x0001 … 0xFFFF. The transmission frequency is calculated as follows:

$$f_{trans} = \frac{osc_{clk}}{8 \cdot timer\_prd}$$

Where

osc_{clk} is the oscillator clock frequency of the DSP.

To ensure successful operation the transmission frequency must not exceed the value calculated below:

$$f_{trans} \leq \frac{osc_{clk,min}}{5.2}$$

Where

$osc_{clk,min}$ is the slowest oscillator clock frequency of both involved boards.

The following table shows the oscillator clock frequencies of the boards:

Board	Oscillator Clock Frequency [MHz]
DS2210	80
DS2211	150

Board	Oscillator Clock Frequency [MHz]
DS2302-01	60
DS2302-04	150

Return value

None

Related topics

References

serial_init 85

serial_init_ds2302

Macro

```
void serial_init_ds2302 (unsigned long timer_prd)
```

> **Note**
>
> Only valid for connections to the old DS2302-01.

Include file

```
Ser31.h
```

Purpose

To initialize the slave DSP's serial interface for data transfer in handshake mode of old DS2302-01 boards connected to DS2210 or DS2211 boards.

Description

This macro calls `serial_init` automatically with the required parameters. For a connection between DS2210, DS2211 or DS2302-04 boards board, refer to serial_init_std_handshake on page 83.

> **Note**
>
> - This macro must only be used with a DS2302-01 board with its serial port connected to a DS2302-04, DS2210, or DS2211 board.
> - After using this macro it is no longer possible to generate PHS-bus interrupts via the respective DS2302-01 board channel using the `phs_bus_interrupt_request()` macro.

Parameters

timer_prd Frequency of the serial transmission via the DX0 output. The valid values are 0x0001 … 0xFFFF. The transmission frequency is calculated as follows:

$$f_{trans} = \frac{osc_{clk}}{8 \cdot timer\_prd}$$

Where

osc_{clk} is the oscillator clock frequency of the DSP.

To ensure successful operation the transmission frequency must not exceed the value calculated below:

$$f_{trans} \leq \frac{osc_{clk,min}}{5.2}$$

Where

$osc_{clk,min}$ is the slowest oscillator clock frequency of both involved boards.

The following table shows the oscillator clock frequencies of the boards:

Board	Oscillator Clock Frequency [MHz]
DS2210	80
DS2211	150
DS2302-01	60
DS2302-04	150

Return value

None

Related topics

References

serial_init 85
serial_init_std_handshake 83

serial_init

Syntax

```
serial_init(
      unsigned long g_ctrl,
      unsigned long timer_prd,
      unsigned long tx_ctrl,
      unsigned long rx_ctrl,
      unsigned long timer_ctrl);
```

Include file

`Ser31.h`

Purpose

To initialize the serial interface of the slave DSP as follows:

- Clear the global control register and the timer control register to reset the serial interface.
- Initialize the serial interface registers with the specified values.
- Start the required serial interface timers according to the control register settings.
- Enable receive and transmit access.

For more information on the serial interface, refer to the Texas Instruments web site at http://www.ti.com and search for the *TMS320C3x User's Guide* (literature number SPRU031F) and *TMS320VC33 Digital Signal Processor* (literature number SPRS087E).

Note

`serial_init` is called automatically by the board-related initialization macros `serial_init_std_handshake` and `serial_init_ds2302`.

Parameters

g_ctrl Setting of the global control register

timer_prd Frequency of the serial transmission via the DX0 output. The valid values are 0x0001 … 0xFFFF. The transmission frequency is calculated as follows:

$$f_{trans} = \frac{osc_{clk}}{8 \cdot timer\_prd}$$

Where

osc_{clk} is the oscillator clock frequency of the DSP.

To ensure successful operation the transmission frequency must not exceed the value calculated below:

$$f_{trans} \leq \frac{osc_{clk,min}}{5.2}$$

Where

$osc_{clk,min}$ is the slowest oscillator clock frequency of both involved boards.

The following table shows the oscillator clock frequencies of the boards:

Board	Oscillator Clock Frequency [MHz]
DS2210	80
DS2211	150
DS2302-01	60
DS2302-04	150

tx_ctrl Setting of the transmit control register

rx_ctrl Setting of the receive control register

timer_ctrl Setting of the timer control register

Return value

None

Related topics

References

serial_init_ds2302 84
serial_init_std_handshake 83

serial_disable

Macro

```
void serial_disable
```

Include file

`Ser31.h`

Purpose

To disable and reset the serial interface.

Description

All serial interface pins are configured as digital I/O pins and set as inputs (refer to Digital I/O via Serial Port (DS2211 RTLib Reference 📖)).

Return value

None

Related topics

References

Digital I/O via Serial Port (DS2211 RTLib Reference 📖)

serial_rx_int_init

Syntax

```
void serial_rx_int_init()
```

Include file

`Ser31ir.h`

Purpose

To initialize the receive interrupt of the serial interface as follows:

- Set the corresponding interrupt vector RINT0 to point to the receive interrupt routine c_int06.
- Enable the receive interrupt and interrupts globally.

Description

The receive interrupt service routine usually contains the receive function. You must program the routine yourself.

> **Tip**
>
> You can use the alias name `isr_receive` instead of `c_int06` for the receive interrupt service routine.

For an example, refer to example 2 in Example of Using the Serial Interface of the Slave DSP on page 81.

Return value

None

Related topics

Examples

References

serial_tx_int_init

Syntax

```
void serial_tx_int_init()
```

Include file

`Ser31ix.h`

Purpose

To initialize the transmit interrupt of the serial interface as follows:

- Set the interrupt vector XINT0 to point to the receive interrupt routine c_int05.
- Enable the receive interrupt and interrupts globally.

Description

The transmit interrupt service routine usually contains the transmit function. You must program the routine yourself.

Tip

You can use the alias name `isr_transmit` instead of `c_int05` for the transmit interrupt service routine.

For an example, refer to example 2 in Example of Using the Serial Interface of the Slave DSP on page 81.

After initialization, you have to start the interrupt-driven transmission with `serial_tx_int_start` on page 89. This macro requests the first transmit interrupt by setting the respective flag in the DSP's IF register.

Return value

None

Related topics

Examples

References

serial_tx_int_start

Macro

```
void serial_tx_int_start()
```

Include file

`Ser31.h`

Purpose

To request the first transmit interrupt by setting the respective interrupt flag in the DSP's IF register. Call this macro after the initialization of the transmit interrupt to start interrupt-driven transmission.

Description

The transmit interrupt is requested by the serial interface when the port is ready to transmit a new word after a preceding transmission.

Note

Use this macro to restart transmission each time it stops.

Return value

None

Related topics

References

disable_rx_int, disable_tx_int

Macro

```
void disable_rx_int()
void disable_tx_int()
```

Include file

`Ser31.h`

Purpose

To disable the serial receive or transmit interrupt (RINT0 or XINT0). The enable bit for the interrupt RINT0 or XINT0 is cleared in the DSP's IE register to disable the corresponding interrupt.

Return value

None

Related topics

References

enable_rx_int, enable_tx_int

Macro	`void enable_rx_int()` `void enable_tx_int()`
Include file	`Ser31.h`
Purpose	To enable the serial receive or transmit interrupt.
Description	The enable bit for the interrupt RINT0 or XINT0 is set in the DSP's IE register to enable the corresponding interrupt.
Return value	None
Related topics	References disable_rx_int, disable_tx_int........ 90

serial_tx_word_poll

Syntax	`int serial_tx_word_poll(void *word)`
Include file	`Ser31.h`
Purpose	To transmit a 32-bit data word via the serial interface.
Description	The value can be of either float or long type. If the transmit buffer of the serial interface is empty, this means the port is ready to transmit, and the function writes the value to the buffer.

Note

You have to initialize the receiving serial interface before starting a transmission. Otherwise, you have to initialize the transmitting port again after the initialization of the receiving port.

Parameters

word 32-bit word to be transmitted (datatype float or long)

Return value

Transmission state; the following symbols are predefined:

Predefined Symbol	Value	Meaning
`SP_TRUE`	0	The transmission has been performed successfully.
`SP_FALSE`	1	The serial interface was not ready to transmit data.

Related topics

References

serial_rx_word_poll.. 93

serial_tx_word_int

Syntax

```
void serial_tx_word_int(void *word)
```

Include file

`Ser31.h`

Purpose

To transmit a 32-bit data word via the serial interface in a transmit interrupt service routine.

Description

The data word is written to the transmit buffer of the serial interface.

> **Note**
>
> You have to initialize and enable the transmit interrupt with `serial_tx_int_init` and enable_tx_int before using `serial_tx_word_int`.
> You have to initialize the receiving serial interface before starting a transmission. Otherwise, you have to initialize the transmitting port again after the initialization of the receiving port.

Parameters

word 32-bit word to be transmitted (datatype float or long)

Return value

None

Related topics

References

enable_rx_int, enable_tx_int 91
serial_tx_int_init 88

serial_rx_word_poll

Syntax

```
int serial_rx_word_poll(void *word)
```

Include file

`Ser31.h`

Purpose

To receive a 32-bit data word via the serial interface.

Description

If the receive buffer contains new data the buffer is read and the function returns SP_TRUE. Otherwise, the buffer is not read and the function returns SP_FALSE.

Parameters

word 32-bit word to be received (datatype can be float or long)

Return value

Transmission state; the following symbols are predefined:

Predefined Symbol	Value	Meaning
SP_TRUE	0	The transmission has been performed successfully.
SP_FALSE	1	The serial interface was not ready to transmit data.

Related topics

References

serial_tx_word_poll..91

serial_rx_word_int

Syntax

```
void serial_rx_word_int(void *word)
```

Include file

`Ser31.h`

Purpose

To receive a 32-bit data word via the serial interface in a receive interrupt service routine.

Description

The data word is read from the receive buffer of the serial interface directly.

> **Note**
>
> You have to initialize and enable the receive interrupt with `serial_rx_int_init` and `enable_rx_int` before using `serial_rx_word_int`.

Parameters

word 32-bit word to be received (datatype can be float or long)

Return value

None

Related topics

References

Floating-Point Format Conversion

Introduction

Because the TMS320VC33 uses a special floating-point format, a floating-point format conversion is necessary whenever float values are transferred between the master processor board and the DS2302.

Converting the Floating-Point Format

Introduction

When values are transferred between DS2302 and processor board, the floating-point format must be converted.

> **Note**
>
> The processor board library also contains an implementation of this functions. Refer to the RTLib Reference of your processor board.

Conversion function

The functions are described in the table below:

Syntax	Description	Parameter
`long cvtie3 (float value)`	The VC33 input value is converted into the processor board's 32-bit IEEE floating-point format. Although the function returns a parameter of *long* type, it contains the pattern of a 32-bit IEEE floating-point value. This function can be used if floating-point values has to be transferred from the DS2302 to the processor board.	*Parameter:* `value` VC33 floating-point input value *Return parameter:* 32-bit integer value containing the bit pattern of the floating point value in IEEE format
`float cvtdsp (long ieee_value)`	The 32-bit IEEE input value is converted into the VC33 floating-point format. Although the input parameter is of *long* type, it must contain the pattern of a 32-bit IEEE floating-point value. This function can be used if floating-point values have to be transferred from the processor board.	*Parameter:* `ieee_value` bit pattern of the floating point value in IEEE format *Return parameter:* floating point value in VC33 format

Time-Base Bus/Engine Position Bus

Introduction

The DS2302 (starting from board revision DS2302-04) provides a time base connector to cascade it with other I/O boards. When I/O boards are cascaded, their angular processing units (APUs) or timing I/O units are given the same time base.

Where to go from here

Information in this section

Basics of Accessing the Time-Base Bus or Engine Position Bus

Introduction

The DS2302 (starting from board revision DS2302-04) provides a time base connector to cascade it with other I/O boards (DS2210, DS2211, DS2302, DS4002, DS5001, DS5203). When I/O boards are cascaded, their angular processing units (APUs) or timing I/O units are given the same time base.

Usable I/O boards

You can connect the time-base bus (engine position bus) of the following I/O boards (the engine position bus of the DS2210 and DS2211 boards is the same as the time-base bus of the DS2302, DS4002, DS5001, and DS5203 boards):

- DS2210
- DS2211
- DS2302 (as of board revision DS2302-04)
- DS4002 (as of board revision DS4002-04)
- DS5001 (as of board revision DS5001-06)
- DS5203 (as of board revision DS5203-05)

Note

The DS2302 board cannot be used as the bus master.

Accessing the Time-Base Bus/Engine Position Bus

Introduction

You can use macros to access the time-base bus or the engine position bus.

Accessing the time-base bus or engine position bus

The following macros let you access the time-base bus or the engine position bus. They are contained in the `ds2302_advanced.h` file.

Syntax	Description	Return Value
`float apu_read_deg(void)`	To read the time base value and scales the angle to degree.	0.0 … 720.0
`float apu_read_rad(void)`	To read the time base value and scales the angle to radiant.	0.0 … (4π)
`int apu_master_available(void)`	To search for a time-base master connected to the time-base bus (engine position bus).	▪ 0: No time-base master was detected ▪ 1: A time-base master was detected
`long apu_16bit_mode_available(void)`	To check whether the time-base bus (engine position bus) is using 16-bit values or 13-bit values.	▪ 0: 13-bit values are used ▪ 1: 16-bit values are used

LED Access

Accessing the LED Function

Introduction

The DS2302-04 board provides a user programmable, green LED per channel. It could be used for debugging purposes or for indicating a certain application state.

Accessing the LED

The LED access macro is contained in the `ds2302_advanced.h` file.

Syntax	Description	Parameter
`void led_state(long value)`	To set the green status LED to the specified state.	LED_OFF or 0: LED is inactive LED_ON or 1: LED is active

Example

The following example shows how to switch the LED on:

```
led_state(LED_ON)                     /* activates LED */
```

Message Functions

Using Messages

Introduction

To simplify the debugging of slave-DSP applications, messages could be send via two different message functions from the DSP to the processor board.

Message functions

The message functions increase the memory consumption and execution time of the slave application enormous. The `msg_printf()` function more, the `print()` function less.

Both message functions use the dual-port memory of the DSP to transfer messages to the master processor board.

Note

- Do not use the `print()` function in conjunction with the `msg_printf()` function.
- Do not use the DS2302 its dual port memory offset range 0x3A00 … 0x3F7F by the application if you use the message functions.

Reading messages by the processor board

The processor board reads the messages from the DS2302's dual port memory with the `ds2302_read_msg()` function. Call this function in a background loop. The messages are transferred into the message module of the processor board and can be observed using the Message Viewer of the experiment software.

Slave-DSP messages contain prefixes like “DSP #1:”. The number depends on the channel which the message sends.

For an example on using messages, see `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\Apu`.

print function

The following table shows the function for messages with reduced capability of formating options but less memory consumption.

Syntax	Description	Parameter
`void print(const char *fmt, …)`	The `print()` function provides a simplified writing of messages into a message buffer within the dual-port memory of the DSP. The header file `print.h` must be included. This function is designed to save memory and stack consumption of the small 2K memory of C31 DSPs on DS2302-01 boards. Therefore the	–

Syntax	Description	Parameter
	`print()` function supports only fixed point parameters and a reduced capability of formating options. The following formatting is supported: ▪ Flag characters: ' ' (white space) ▪ Width specifiers: 0n, n ▪ Input size modifier: l, L, h ▪ Type characters: x, X, d, i	

Example The following code is an example.

```
print(„register = 0x%08lX“, my_register);
print(„count = %d“, count);
print(„Application started.“);
```

In the Message Viewer you receive for example the following:

```
ds1006:   18:19:46   [#5] ds1006 - RTLIB:   DSP #1: Application started. (1)(0xCA)
```

printf function

The following table shows the function for messages with all possibilities of printf formating but requiring more memory consumption.

Syntax	Description	Parameter
`msg_printf(const char *fmt, ...)`	The `msg_printf()` function allows to print messages with all possibilities of printf-formating. The messages are written into a message buffer within the dual-port memory of the DSP. The `msg_printf.h` header file must be included. Due to increased memory and stack consumption this function can only be used on the DS2302-04 boards and newer, because only these boards provide the necessary memory.	–

Example The following code is an example.

```
msg_printf("register = 0x%08lX", my_register);
msg_printf("count = %d, voltage = %f", count, voltage_flt);
msg_printf("Application started.");
```

In the Message Viewer you receive for example the following:

```
ds1006:   18:19:46   [#5] ds1006 - RTLIB:   DSP #1: Application started. (1)(0xCA)
```

Related topics

References

ds2302_read_msg (DS2302 RTLib Reference 📖)

Execution Time Profiling

Introduction

There are two header files providing macros for execution time measurement of individual code parts in timer interrupt service routines.

Where to go from here

Information in this section

Measuring Execution Times with the util2302.h Module

Introduction

Two macros for execution time measurement of individual code parts in timer interrupt service routines are contained in the `util2302.h` header file.

Basics

The macros can be used only within timer interrupt service routines and must be used together. The appropriate timer for the respective timer interrupt service routine must be specified in the parameter list of both macros for proper operation.

The macros introduce an overhead of 4 processor cycles (133 ns on a 60 MHz DS2302), which must be subtracted from the result to get the correct execution time.

Note that the macros in the new module tic3x.h described below provide more convenient execution time measurement. The macros `count_timer()` and `time_elapsed()` are still available for compatibility reasons.

Macros

The macros are described in the table below:

Syntax	Description	Parameter
`long timer_count(long timer)`	This macro returns the current contents of the timer counter register of the specified TMS320VC33's on-chip timer.	Parameter: `timer` Target timer (0 or 1) *Return parameter:*

Syntax	Description	Parameter
		Timer Counter Value
`float time_elapsed(long timer, long prev_count)`	The duration between the timer count value `prev_count` and the current timer count value is computed and returned as a floating-point value scaled in seconds. The value `prev_count` can be obtained by invoking the macro `timer_count()` at the point where execution time measurement is intended to start.	*Parameter:* *timer* Target timer (0 or 1) *Return parameter:* Elapsed time in *sec*

Example The following example shows how to use the macros.

```
void c_int09()           /* timer0 interrupt service routine */
{
  ...
  count = timer_count(0);
  y = sqrt(x);
  time = time_elapsed(0,count);
  ...
}
```

The example demonstrates how the execution time requirement of the function `sqrt()` can be evaluated. A host or master processor board program must be used to make the result visible. For some suggestions, refer to Advanced Programming Techniques on page 111.

Measuring Execution Times with the tic3x.h Module

Introduction

The `tic3x.h` header file contains definitions and macros which are used for execution time measurement.

Macros

The macros are described in the table below:

Syntax	Description	Parameter
`void tic0_init(void)` `void tic1_init(void)`	Initializes and starts timer i for execution time measurement. **Note** Do not call this macro if the respective timer is already in use, e.g., for timer interrupt generation.	–

Syntax	Description	Parameter
`void tic0_start(void)` `void tic1_start(void)`	Starts the execution time measurement.	–
`void tic0_halt(void)` `void tic1_halt(void)`	You can pause execution time measurement.	–
`void tic0_continue(void)` `void tic1_continue(void)`	Continues the execution time measurement after a pause caused by `tic0_halt()` or `tic1_halt()`.	–
`float tic0_read(void)` `float tic1_read(void)`	The macros return the exact execution time in seconds between `ticx_start()` and `ticx_read()` with consideration of the idle time between `ticx_halt()` and `ticx_continue()`.	–
`float tic0_read_total(void)` `float tic1_read_total(void)`	Reads out the total execution time in seconds without consideration of the idle time.	–
`void tic0_delay(float duration)` `void tic1_delay(float duration)`	The macros can be used to hold the program execution for a specified time. The minimum delay time is 1.4 µs. The delay time can be adjusted in 0.4 µs steps with a maximum error of +0.5 µs (compiler optimization at level -o 2 assumed). The maximum delay is 114 seconds on a 150 MHz DS2302-04 if the timer was initialized for execution time measurement using `tic0_init()` or `tic1_init()`. If the timer is already in use for timer interrupt generation, the delay value must not exceed the initialized timer period.	–

Example The following example shows how to use the macros.

```
void isr_t0()
{
  ...
  tic1_start();                          /* start execution time measurement */
  function1(arg);

  tic1_halt()                             /* halt execution time measurement */
  function2(arg);
  tic1_continue();                    /* continue execution time measurement */

  function3(arg);
  exec_time = tic1_read();            /* read execution time of fct 1 & fct 3 */
  ...
}

void main()
{
  ...
  tic1_init();                                         /* initialize timer 1 */
  ...
}
```

Using the DS2301 with the DS2302 Software Environment

Introduction

The DS230x software environment supports both DDS boards, the DS2301 and the DS2302.

Where to go from here

Information in this section

Information in other sections

DS2301 RTLib Reference
Provides detailed descriptions of the C functions needed to program RTI-specific Simulink S-functions or implement your real-time models manually via C programs (handcoding).

Using the DS2302 Software Environment for a DS2301 Board

Introduction

All functions known from the DS2301 software environment are available for the DS2301 DDS board. To use the DS2301 related functions of the *ds230x.c* source file it is necessary to include the *ds2301.h* header file instead of *ds2302.h*.

Changes Between the DS2301 and DS230x Software Environment

Introduction

The DS230x software environment differs from the DS2301 software environment.

Changes

The `ptr1.obj` and `ptr2.obj` object files containing pointer definitions have been removed. All pointer variables have been replaced by macros except for `*dac` and `*int_tbl`.

The `*dac` and `*int_tbl` pointers are declared and initialized in the `init()` function, refer to Predefined Pointer Declarations and Global Variables on page 57.

New Identifiers and Function for the DS2301 Board

Identifiers

Some new identifiers for numerical constants and a new function are available for the DS2301 DDS board:

- `DS2301_2_BASE`
- `DS2301_ALL_CH`
- `ds2301_get_board_type()`

Converting Applications for Different DDS Board Types

Introduction

A master processor board C program accessing a DS2301 can be converted to access a DS2302 board and vice versa.

Converting applications

To access a DS2302 instead of a DS2301 board, all you have to do is replace '2301' by '2302' everywhere in the function names and identifiers.

Example:

```
DS2301_INT3(BASE, DS2301_CH1, value);
```

change to

```
DS2302_INT3(BASE, DS2302_CH1, value);
```

Limitations

Note the following limitations:

If you want to access the DS2302 instead of the DS2301 board, the following functions cannot be converted:

- `ds2301_read_osr()`
- `ds2301_write_osr()`
- `ds2301_read_custom_register()`
- `ds2301_write_custom_register()`

If you want to access the DS2301 instead of the DS2302 board, the following functions cannot be converted:

- `ds2302_dsp_reset_on_ioerr()`
- `ds2302_module_reset_on_ioerr()`

Using DS2301 Applications with a DS2302

Porting DS2301 Applications to the DS2302

Introduction

C programs written for the DS2301 can be used with the DS2302 DDS board.

Limitation

There is only one limitation: Summing the analog output signals using the DS2301's Output Summing feature is not possible with the DS2302.

Migrating

The `ds2301.h` and `util2301.h` include files must be replaced by `ds2302.h` and `util2302.h`.

The macro names for accessing the digital I/O pins TCLK, DX and FSX have been changed in the DS230x software environment as described below and can be redefined by including `convert.h`.

So it is necessary to include the `convert.h` header file to redefine these macros. The DS2301 function `dig_io_init()` must be replaced by a suitable macro.

> **Note**
>
> The `convert.h` header file must be included after the `ds2302.h` header file.

The following table shows the macro changes:

Digital I/O Pin	DS2301 Macro	DS2302 Macro
TCLK	`dig_out1()`	`dig_out6()`
DX	`dig_out2()`	`dig_out1()`
FSX	`dig_out3()`	`dig_out2()`

DS2302-04 Boards Together with DS2302-01 Boards

Using DS2302-04 and DS2302-01 Boards Together

Introduction

The DS2302-04 board is completely downward compatible with the DS2302-01 board. It executes applications which are written for the older board without problems. The VC33 DSP of the DS2302-04 is object-compatible with the C31 DSP of the DS2302-01. The DS2302-04's default DSP clock setting after an application is load 60 MHz, which is expected by DS2302-01 applications.

Detecting the board version

The DS2302-04 software environment is designed to support both versions of DS2302 boards. The `init()` function of the DSP application can detect the board version of the DS2302. If a DS2302-04 board is detected, the initialization function switches the DSP clock to its maximum of 150 MHz and reinitializes the internal timer clock variables to match the increased clock setting.

Using hardware features of the DS2302-04

If you want to use the new hardware features of the DS2302-04 board, such as the time-base connector or the additional 32 KWord RAM, you must include the `ds2302_advanced.h` header file. Including this header file marks the application as a DS2302-04 board application. Due to this marking, the loader of the dSPACE experiment software and the generated loader function of `DDS2C` prevent the loading of this application to DS2302-01 boards which do not support the new features. During the build process with the `CL230x` utility, the linker command file generator `GetSizeC3x` also detects the marking and assigns the sections of the application to the additional RAM blocks RAM2 and RAM3.

PHS bus mode

The DS2302 (starting from board revision DS2302-04) provides a PHS++ interface. Unfortunately the DS2302 must use a different PHS-bus register mapping in PHS++ mode. After loading an application the DS2302-04 board is set to standard PHS-bus mode, to support applications, build for the old DS2302 board. The new master processor board software environment detects the new DS2302-04 board version and will enable the PHS++ mode during initialization. Due to the different register mapping custom software modules written for the old DS2302-01 with direct register access, such as S-functions or loader functions generated by old versions of DDS2C will not work properly. In this case you can force the DS2302-04 board to standard PHS-bus mode by using the `ds2302_phspp_init` function.

Related topics

Basics

ds2302_phspp_init (DS2302 RTLib Reference 📖)

Advanced Programming Techniques

Introduction

This section assumes that you are already familiar with the DS2302 board and the related software. You should also be familiar with the assembly programming language for the TMS320VC33 DSP.

All sampling period values listed are given for a DS2302 board with 60 MHz oscillator clock frequency.

Where to go from here

Information in this section

Improving Time-Critical C Programs

Introduction

The time-critical part of most DS2302 application programs is the timer interrupt service routine. If an application has to run at very high sampling rates, this part of the program should be kept as short as possible.

Where to go from here

Information in this section

Using the TI Optimizer

Introduction

You can use the Texas Instruments (TI) optimizer to optimize your application program.

Optimizing applications

To optimize application programs on the C language level, it is recommended to invoke the Texas Instruments optimizer with the command line option `-s` (for example, using the switch `-co` of `CL230x.EXE`). This causes the C language commands to be merged into the assembly source code as comments and thus allows direct comparison of each C command line with the corresponding assembly instructions.

Related topics

References

Variable Initialization

Initialization

Expressions that need to be computed only once should be evaluated in the initialization phase of the `main()` routine. The initialization phase is not time-critical, because it is usually executed before timer interrupts are enabled.

```
void c_int09()
{
  ...
  angle += rpm * 2.0 * PI * 60.0 * dt;
  ...
}
```

If the variable `rpm` in the above example is a constant, the entire computation of the angle increment can be moved to the initialization phase of the `main()` routine.

However, constant expressions like `2.0 * PI * 60.0` are already computed during compilation. Thus, the above term needs two multiplications and one addition during run time.

```
void c_int09()
{
  ...
  angle += d_angle;
  ...
}

void main()
{
  ...
  d_angle = rpm * 2.0 * PI * 60.0 * dt;
  ...
  timer0(...);
  for (;;)
  ...
}
```

Background Computations

Introduction

Expressions that must be computed periodically, but not synchronously to the timer interrupts, can be computed in the background.

Background computation

The angle increment from the example in the foregoing section might also be computed in the background, e.g. if *rpm* varies during run time. The update rate depends on the time available for background operations, i.e. the spare time left between subsequent timer interrupt services.

```
void c_int09()
{
  ...
  angle += d_angle;
  ...
}

void main()
{
  ...
  timer0(...);
  for (;;)
  {
    d_angle = rpm * 2.0 * PI * 60.0 * dt;
  }
}
```

Arrays and Pointers

Chained data structure

When data tables must be read out, e.g. in table look-up applications, using a chained data structure results in much faster code than accessing a simple array.

```
void c_int09()
{
  *dac = cur->value;
  cur = cur->next;
}
```

The following data structure is used in the example. The pointer next in each table element is initialized to point to the succeeding table entry and the last element in the table is linked to the first one.

```
typedef struct tbl_struct tbl_t;

struct tbl_struct
{
  long value;
  tbl_t *next;
};
```

Thus it is essential to reset the index to the table origin, because the `next` pointers connect each table entry with its successor.

The following listing is an excerpt from the corresponding assembly source code.

```
*** 3 -----------------------   *dac = cur->value;
      LDI           @_cur,AR0
      LDI           @_dac,AR1
      LDI           *AR0,R0
      STI           R0,*AR1
*** 4 -----------------------    cur = cur->next;
      LDI           *+AR0(1),R0
      STI           R0,@_cur
```

The above code might be implemented with only 3 assembly instructions in the timer interrupt service routine by removing the load and store instructions for the addresses `@_cur` and `@_dac`. This is possible if these addresses are constantly held in registers (here AR0 and AR1), and initialized in the `main()` routine (see Holding Variables in Registers on page 120). However, this requires hand optimizations on the assembly level.

```
LDI    *AR0,R0
STI    R0,*AR1
LDI    *+AR0(1),AR0
```

The same application with the data table implemented as an array is listed below.

```
void c_int09()
{
  *dac = tbl[i++];
  if (i >= 10) i = 0;
}
```

Compare the following assembly source code with the pointer table version.

```
*** 3 ----------------------- *dac = tbl[i++];
      LDI            @_i,IR0
      ADDI           1,IR0
      STI            IR0,@_i
      LDI            @_tbl,AR0
      SUBI           1,AR0
      LDI            @_dac,AR1
      LDI            *+AR0(IR0),R0
      STI            R0,*AR1
*** 4 ----------------------- i = (i >= 10) ? 0 : i;
      LDI            @_i,R0
      CMPI           10,R0
      LDIGE          0,R1
      LDILT          @_i,R1
      STI            R1,@_i
```

Although some load and store instructions might also be moved to the `main()` routine, this implementation consumes much more execution time than implementation using pointers.

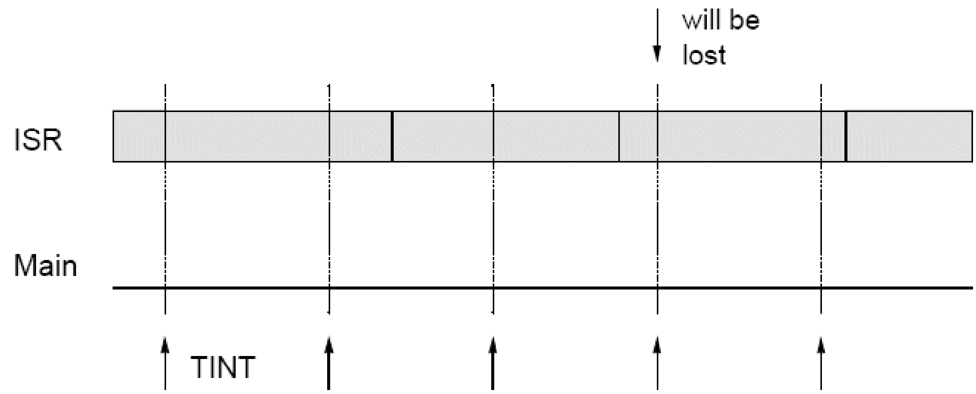
Processor Load

Introduction

You must ensure that a signal generation code can be actually executed at the given sampling rate.

Executing signal generating code

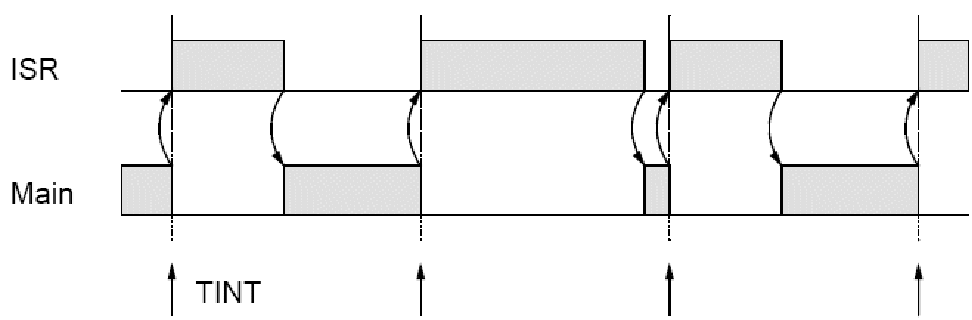
Most DS2302 standard applications use a timer interrupt service routine to execute the signal generating code at a regular sampling rate. You must ensure that this piece of code can be actually executed at the given sampling rate, i.e. that the execution of the timer interrupt service routine has already finished when the next timer interrupt is received. Otherwise, since interrupts are disabled during interrupt service by default, a pending interrupt is served only after the previous interrupt service has been finished. Because the DSP stores only one interrupt request, a timer interrupt will be lost from time to time. Also there will be no time left for any background computations (see illustration below).



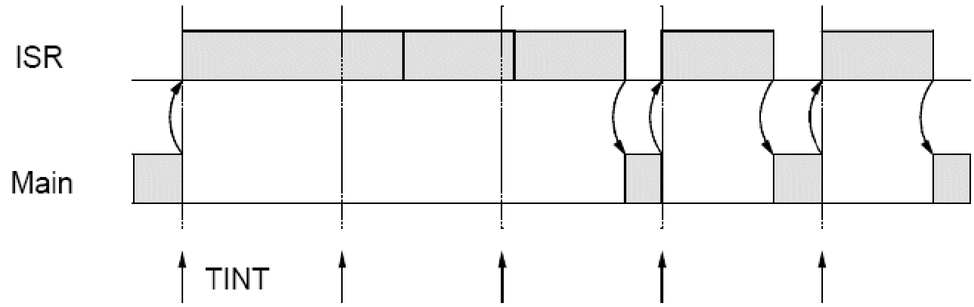
If the timer interrupt service routine consists of multiple different program branches, e.g. due to if-then-else constructions, the problem becomes more complex.

Consider the following example: A timer interrupt service routine consists of two program branches of different lengths, where the longer branch requires twice as much execution time as the short one. The program runs through the long branch only once during 1000 sampling periods.

Normally, the maximum sampling rate must be selected appropriately, so that the long program branch can be executed within a single sampling period, even though only one of 1000 timer interrupt services make full use of the entire sampling period as shown in the illustration below.



If the sampling rate is increased and the long branch no longer fits into a single sampling period, no timer interrupt is lost as long as the short program branch can be executed within a single sampling period and there is enough time left to allow the timer interrupt service to recover from a long program branch as shown in the illustration below. However, this will introduce a slight jitter to the generated signal. If this jitter is acceptable, the sampling period can be adjusted to the execution time requirement of the short program branch, resulting in a much higher sampling rate.



You can use the SPEEDCHK utility to evaluate the execution time requirements of the longest and shortest branches in the timer interrupt service routine.

Assembly Code Optimizations

Introduction

Before you start to perform optimizations on the assembly level, use SPEEDUP for automatic assembly code optimization. Because any manual optimizations must be repeated each time the C source code is modified, you should make sure that your application C source code is final.

Where to go from here

Information in this section

Information in other sections

Holding Variables in Registers

Introduction

Any unused registers can be used to hold constant addresses or variables that are frequently needed in the interrupt service routine.

Variables in registers

Variables to be hold in registers need to be loaded from memory and saved to memory. The example below shows a code sequence from an interrupt service routine.

```
LDF     @_angle,R5
CMPF    *AR6,R5
...
ADDF    @_delta_deg,R5
STF     R5,@_angle
```

Assuming that the register R5 is not used otherwise, it can be loaded from the memory location `@_angle` once in the `main()` routine. Then the current value is constantly held in `R5` and the load and save instructions can be removed. The remaining code is listed below.

```
CMPF    *AR6,R5
...
ADDF    @_delta_deg,R5
```

Delayed Branches

Introduction

The delayed branches allow execution of 3 additional instructions before the branch is actually performed.

Delayed branches

Branch instructions require 4 processor cycles. Special versions of the branch instructions, the delayed branches, allow execution of 3 additional instructions before the branch is actually performed. The delayed branch instruction itself requires only 1 processor cycle.

```
     CMPF    *AR6,R5
     BLT     L2
     LDI     *+AR6(1),R1
     STI     R1,*AR1
     ...
L2:
     ADDF    @_delta_deg,R5
```

In the listing below, the branch instruction has been replaced by the appropriate delayed branch. The point where the branch actually occurs is indicated by the original branch instruction included as a comment.

```
     CMPF    *AR6,R5
     BLTD    L2
     LDI     *+AR6(1),R1
     ADDF    @_delta_deg,R5
     NOP
***  BLT     L2 ; branch occurs
     STI     R1,*AR1
     ...
L2:
```

Note that the 3 instructions following the delayed branch are executed anyway, whether or not the branch condition is true. You must therefore place appropriate instructions there. If not all of these 3 instructions can be appropriately used, NOP instructions must be included.

A Hand-Optimized Application Program

Introduction

An application program can be optimized to reduce the sampling period.

Hand optimization

The example below shows excerpts from the assembly source file of the crankshaft sensor signal generator that comes with the DS2302 demo software. The C source code of this example can be found in <RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\crank. The following listing shows the original assembly source code of the timer interrupt service routine. PUSH and POP instructions for context save and restore have been removed by using SPEEDUP.

Note

The assembly source file described in this example is generated by the Texas Instruments Compiler Vs. 4.7. Newer compiler versions will generate different assembly source code.

```
_c_int09:
        PUSH   ST
        LDI    @_cur,AR6
        LDF    @_angle,R5
        CMPF   *AR6,R5
        BLT    L2
        LDI    @_dac,AR1
        LDI    *+AR2(1),R1
        STI    R1,*AR1
        SUBF   *+AR6(2),R5
        STF    R5,@_angle
        LDI    *+AR6(3),R1
        STI    R1,@_cur
L2:
        ADDF   @_delta_deg,R5
        STF    R5,@_angle
EPI0_1:
        POP    ST
        RETI
```

The corresponding optimized timer interrupt service routine is listed below. The optimizations applied are those discussed in the foregoing sections.

Because the registers have already been substituted by SPEEDUP and are not used otherwise, initialization of the constant address `@_dac` can be moved to the `main()` routine. Also the variables `@_angle` and `@_cur` can also be held constantly in the registers R5 and AR6, respectively.

The branch instruction has been substituted by its corresponding delayed branch (see Delayed Branches on page 121) and the ADDF instruction, which is not dependent on the branch condition, has been placed after the delayed branch. One NOP instruction has been added after the delayed branch (see Delayed Branches on page 121).

```
_c_int09:
        PUSH    ST
        CMPF    *AR6,R5 ; compare current pos against edge pos
        BLTD    L2 ; done if current pos less than edge pos
        LDI     *+AR6(1),R1 ; load output value in R1
        ADDF    @_delta_deg,R5 ; add angle increment
        NOP
        STI     R1,*AR1 ; store output value to DAC
        SUBF    *+AR6(2),R5 ; subtract angle reset value
        LDI     *+AR6(3),AR6 ; pointer points to next list element
L2:
EPI0_1:
        POP     ST
        RETI
```

The listing below is an excerpt from the `main()` routine, which has been augmented by the bold printed instructions in order to initialize the variables `@_angle` and `@_cur`, and by the address `@_dac` pointing to the D/A converter output register.

```
_main:
        ...
        PUSHF   R0
        CALL    _timer0
        SUBI    1,SP
        LDI     @STATIC_2,AR4
        LDI     @STATIC_1,AR5
        LDI     @STATIC_5,R3
        LDI     @STATIC_4,RC
        LDI     @_rpm,AR2
        LDF     @_delta_t,R4
        LDI     @_dac,AR1 ; pointer to DAC output register
        LDI     @_cur,AR6 ; pointer to current list element
        LDI     @_angle,R5 ; current angle
L10:
        ...
        B       L10
```

The table below shows the execution time requirements for the original C version of the crankshaft sensor signal generator, the SPEEDUP optimized version, and the hand-optimized version discussed above.

Optimization	Sampling Period
pure C	1.53 µs
SPEEDUP	1.13 µs
assembly	0.73 µs

DSP Synchronization by Interrupts

Where to go from here

Information in this section

Basics of Synchronizing the DSPs

Introduction

The 6 on-board DSPs can be synchronized.

Synchronization

The 6 on-board DSPs can be synchronized to each other by using the TMS320VC33's interrupt INT1. This interrupt line is connected to one of the external output pins XF0 or XF1 of another on-board DSP, as selected by the interrupt select register. The setup information for the interrupt select register can be specified for any individual application and actually loaded to the hardware by the master processor board load facility before the application is started.

An interrupt can be served either by simply polling the interrupt flag in the on-board I/O control register (IOCTL), or by installing the corresponding interrupt service routine. Both methods are described in the following chapters.

Macros and functions for using interrupts are already included in the `ds2302.h` header file and in the `ds230x.lib` object library. For details, refer to Interrupts on page 66.

Polling the INT1 Interrupt Flag (crcam)

Introduction

An application shows how two camshaft sensor signal generators are synchronized to a crankshaft sensor signal by polling an interrupt flag.

Signal generation

The example listings below are excerpts from the crankshaft/camshaft sensor signal generator that comes with the DS2302 demo applications. In this application two camshaft sensor signal generators are synchronized to a crankshaft sensor signal. All the demo programs are in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\crcam`.

This application uses three DSPs in parallel. The first channel executes a modified version of the crankshaft sensor signal generator (see Crankshaft Sensor Signal Generator (crank) on page 35). The camshaft sensor signal generators run on channels 2 and 3.

Whenever the crankshaft signal has reached its zero degree position, a pulse is issued on the XF0 line to request an INT1 interrupt on channels 2 and 3. This is simply done by using the `int_xf0()` macro from the `ds2302.h` header file.

```
/* synchronize camshaft sensor signal generator(s) */

if (cur->pos == 0.0) int_xf0();
```

The camshaft sensor signal generators must serve this interrupt request. The simplest method is to poll the appropriate interrupt flag. This can be done by using the macro `int1_pending()`. No interrupt handler is necessary in this case.

```
/* synchronization to crankshaft TOC position */

if (int1_pending())       /* poll INT1 interrupt request bit */
{
  if (angle < 10.0) angle = 0.0; /* zero angle if near 0 deg */
  int1_ack();                  /* acknowledge interrupt INT1 */
}
```

Since the crankshaft generator runs at twice the camshaft wheel speed, every second INT1 interrupt request only corresponds to the camshaft's zero degree position. 360 degrees are subtracted from the angle whenever the last output event is issued, resulting in a negative angle until the zero position is reached. Thus, the angle can simply be compared to a positive bound (here 10 degrees) to decide whether it must be reset to zero.

Keep in mind to reset the INT1 interrupt flag by using the `int1_ack()` macro.

Note

The INT1 interrupt lines for each DSP must be appropriately selected. In this case channel 1 requests INT1 interrupts on channels 2 and 3 by using its XF0 bit, so the default settings can be used.

Using an INT1 Interrupt Service Routine (knock)

Introduction

As an alternative, an interrupt service routine can be used to serve an INT1 interrupt request instead of polling the interrupt flag. This saves execution time because no polling of the interrupt flag is needed.

Triggering a knock signal

The examples below are excerpts from the crankshaft/knock signal generator. In this application a knock signal is triggered by a crankshaft sensor signal. All the demo programs are in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\knock`. Knock sensor signals are often needed in automotive applications, for example, for hardware-in-the-loop simulations in conjunction with an electronic engine control unit (ECU).

Again the first channel executes a modified version of the crankshaft sensor signal generator generating INT1 interrupts. Two interrupts are issued per revolution, corresponding to a 4 cylinder / 4 cycle engine. The trigger angles are held in variables and can be modified on-the-fly by the host or by a master processor board via the dual-port memory.

The knock sensor signal generator running on channel 2 is triggered by these interrupts. Whenever an interrupt is received, the program generates a sine wave of 7.5 kHz frequency with decreasing amplitude. This is performed evaluating the discrete-time characteristic polynomial of a sine wave with exponential damping. The characteristic polynomial is the denominator of the z-transform of the sine signal. The INT1 interrupt service routine puts the sine wave generator into its initial state. During subsequent sampling steps the system responds with a sine wave signal, fading off after a short time according to the specified damping coefficient. A Gaussian noise is added to the output signal.

For a description of the characteristic sine polynomial and the implementation on a digital processor, refer to appropriate literature. The noise generator (see PRBS and Gaussian Noise Generators (noise) on page 28) is used to generate the Gaussian noise.

The INT1 interrupt service routine is listed below. It shows how the sine wave generator is initialized each time an INT1 interrupt is received.

```
void c_int02()            /* INT1 interrupt service routine */
{
  xk = xk_init; /* put sine wave generator into initial state */
  xk1 = xk1_init;
  xk2 = xk2_init;
  int1_ack();                         /* clear interrupt bit */
}
```

The piece of code below shows the part of the timer interrupt service routine evaluating the discrete-time characteristic polynomial of the sine wave.

```
/* generate sine wave with decreasing amplitude
(xk is output) */
/* x(t) = exp(-d*omega_0*t) * sin(omega_0*t) */
xk2 = xk1;
xk1 = xk;
xk = -a1 * xk1 - a2 * xk2;
```

In order to keep the timer interrupt service as short as possible, the coefficients and initial values of the sine wave generator are calculated during the initialization phase in the `main()` routine. The corresponding code is listed below.

```
/* initialize sine wave generator */
a2 = exp(-DAMP * OMEGA_0 * TS);      /* coefficients of the */

/* characteristic polynomial */
a1 = -2.0 * a2 * cos(OMEGA_0 * TS);
a2 = a2 * a2;                      /* of a sine wave signal */

xk_init = 1.0;                            /* initial state of */
xk1_init = cos(-OMEGA_0 * TS);            /* sine generator */
xk2_init = cos(-2.0 * OMEGA_0 * TS);
```

Note that the `int1_init()` function must be called in the program's initialization part to initialize the INT1 interrupt vector.

```
...
int1_init();                /* setup INT1 interrupt service */
timer0(TS);                            /* initialize timer0 */
...
```

Host Interrupt

Using the Host Interrupt INT3

Introduction

A master processor board can trigger the INT3 interrupt of a connected to the DS2302 via PHS bus.

Triggering the INT3 interrupt

The interrupt INT3 can be triggered by the master processor board connected to the DS2302 via PHS bus. This is simply done by writing to the reserved dual-port memory location at offset 0x03FFF of the DS2302 channel(s) to be interrupted. The value being written can be read from that dual-port memory location by the respective DSP.

The master processor board cannot detect whether an INT3 interrupt request has been served by the DSP. You can implement some sort of handshake, if needed, as demonstrated in the following example.

The piece of code listed below is an excerpt from a master processor board. In this case the dual-port memory location reserved for the INT3 interrupt serves as an end-of-interrupt flag. The flag is set by the host to request an INT3 interrupt and is cleared by the DSP after the interrupt service has finished. So the master can therefore test whether an interrupt service is completed.

```
#define INT3_ADDR 0x03FFF
UInt32 eoi;
...
ds2302_read(DS2302_1_BASE, DS2302_CH1, INT3_ADDR, (UInt32 *)&eoi);
if (eoi == 0)       /* test if interrupt service is finished */
  /* request a new INT3 interrupt */
  ds2302_INT3(DS2302_1_BASE, DS2302_CH1, 1);
```

The corresponding DSP program part is listed below. The DSP must simply clear the handshake flag to signal an end-of-interrupt service to the host.

A pointer **`int3`** for accessing the reserved INT3 dual-port memory location is already declared in the **`ds2302.h`** header file.

The contents of the INT3 memory location returned by the **`int3_ack()`** macro are stored in a dummy variable and is not used here.

```
void c_int04()             /* INT3 interrupt service routine */
{
  long dummy;
  ...                           /* perform interrupt service */
  int3_ack(dummy);                  /* clear interrupt flags */
  *int3 = 0;  /* signal end-of-interrupt service to the host */
}
```

The INT3 interrupt vector must be initialized in the program's initialization part by using the function `int3_init()`.

```
...
init();                                      /* initialize hardware */
int3_init();                        /* setup INT3 interrupt service */
timer0(TS);                                    /* initialize timer0 */
...
```

Note that the contents of the INT3 dual-port memory location is undefined after power-up. It cannot be initialized by a master processor board without requesting an INT3 interrupt as long as the INT3 interrupt is enabled. However, it can be initialized by the DSP in the `main()` routine before the INT3 interrupt is enabled.

As an alternative, you can serve an INT3 interrupt request alternatively by using a polling technique for the interrupt INT1. Refer to Polling the INT1 Interrupt Flag (crcam) on page 124. However, note that the `int3_pending()` macro has to be used a little differently. Refer to Interrupts on page 66.

ControlDesk Controls DS2302 via Master Processor (fgen)

Where to go from here

Information in this section

Master Processor Board Agent Program

Introduction

A master processor agent program transfers data between a master processor and the DS2302.

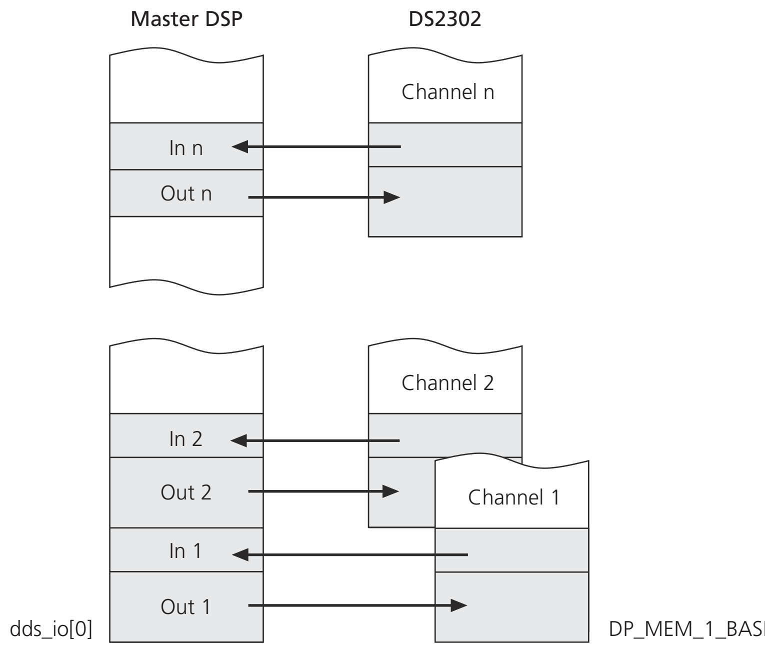
Agent program

The master processor agent program is not limited to this particular application and can be customized for different applications by modifying some of the following constants.

```
#define N_OUT          5                  /* number of DDS output parameters */
#define N_IN           1                   /* number of DDS input parameters */
#define N_PAR          (N_OUT+N_IN)                /* parameters per channel */
#define N_CH           1                    /* number of DDS channels served */
#define MSTR_BUF_OFFS  0            /* dp-mem start addr of master DSP buffer */
#define DDS_OUT_OFFS   0            /* dp-mem start addr of DDS output buffer */
#define DDS_IN_OFFS    (DDS_OUT_OFFS+N_OUT)              /* DDS input buffer */
```

The `N_OUT` constant specifies the number of parameters to be transferred to each DS2302 channel, and `N_IN` gives the number of parameters to be read from each DS2302 channel. `N_CH` specifies the number of DS2302 channels to be served by the agent program.

The data buffer on the DS2302 is located at the beginning of the dual-port memory as shown in the illustration below. The data buffer on the master processor has been allocated by the malloc function. To keep the program as simple as possible, the number of parameters is the same for each DS2302 channel.



The `dds_io` pointer points to the start of the master processor's transfer buffer.

```
dds_io = (Int32 *) malloc(sizeof(Int32) * (N_CH * N_OUT));
```

Only parameters that have been modified must be written to the DS2302. An internal array is used to hold a copy of the output parameters to detect if any parameters have been changed in the buffer.

```
long out[N_CH, N_OUT]; /* internal copy of output parameters */
```

The data transfer between the master processor and the DS2302 is performed in a timer interrupt service routine at a fixed update rate (here 100 Hz).

```
void isr_t1()                              /* timer1 interrupt service routine */
{
  ts_timestamp_type ts;
  ts_timestamp_read(&ts);
  host_service(1, &ts);

  for (ch = 0; ch < N_CH; ch++)
  {
    for (i = 0; i < N_OUT; i++)
    {
      if (out[ch, i] != dds_io[ch*N_PAR+i])
      {
        ds2302_write(
          DS2302_1_BASE, 0x01 << ch, DDS_OUT_OFFS+i,
          &(dds_io[ch*N_PAR+i]));
        out[ch,i] = dds_io[ch*N_PAR+i];
      }
    }
    for (i = 0; i < N_IN; i++)
    {
      ds2302_read(
        DS2302_1_BASE, ch+1, DDS_IN_OFFS+i,
        &(dds_io[ch*N_PAR+N_OUT+i]));
    }
  }

  /* calculate frequency value to be displayed in ControlDesk */
  freq_display = RTLIB_CONV_FLOAT32_FROM_TI32(dds_io[5]);
  if(freq_display < 1.0)
  {
    freq_display *= 1000.0;
    freq_scale = 2;
  }
  else if(freq_display > 999.9)
  {
    freq_display /= 1000.0;
    freq_scale = 3;
  }
  else
  {
    freq_scale = 1;
  }
}
```

The routine starts with the first channel. Each output parameter that has changed in the buffer meanwhile is written to the DS2302. Then the input parameters are read from the DS2302 and are updated in the master processor board's data buffer. This is repeated for further DS2302 channels depending on the `N_CH` constant.

All output parameters are read from the respective DS2302 dual-port memories at program start to initialize the master processor board's data buffer. Thus, that the DS2302 application program be loaded before the initial values are retrieved from the DS2302. Otherwise the data buffer of the agent program will be initialized with random data.

```
void main()
{
  ...

  /* read initial output values from DS2302 dual-port memory */
  for (ch = 0; ch < N_CH; ch++)
  {
    for (i = 0; i < N_OUT; i++)
    {
      ds2302_read(
        DS2302_1_BASE, ch+1, DDS_OUT_OFFS+i,
        &(dds_io[ch*N_PAR+i]));
      out[ch,i] = dds_io[ch*N_PAR+i];
    }
  }

  ...
  RTLIB_SRT_START(TS, isr_t1);

  for (;;)                                  /* background task */
  {
    /* select signal wave form */
    if(signal_step == 1)
    {
      dds_io[0] += 1;
      if(dds_io[0] > 4)
        dds_io[0] = 1;
      signal_step = 0;
    }

    /* select frequency range */
    if(freq_step == 1)
    {
      freq_fact /= 10.0;
      if(freq_fact < 0.01)
        freq_fact = 0.01;
      freq_step = 0;
    }
    else if(freq_step == 2)
    {
      freq_fact *= 10.0;
      if(freq_fact > 10000.0)
        freq_fact = 10000.0;
      freq_step = 0;
    }

    /* write frequency value and factor to buffer */
    dds_io[1] = RTLIB_CONV_FLOAT32_TO_TI32(ampl);
    dds_io[2] = RTLIB_CONV_FLOAT32_TO_TI32(frequency);
    dds_io[3] = RTLIB_CONV_FLOAT32_TO_TI32(freq_fact);
    dds_io[4] = RTLIB_CONV_FLOAT32_TO_TI32(offs);

    RTLIB_BACKGROUND_SERVICE();
  }
}
```

ControlDesk-Controlled DDS Function Generator

Introduction

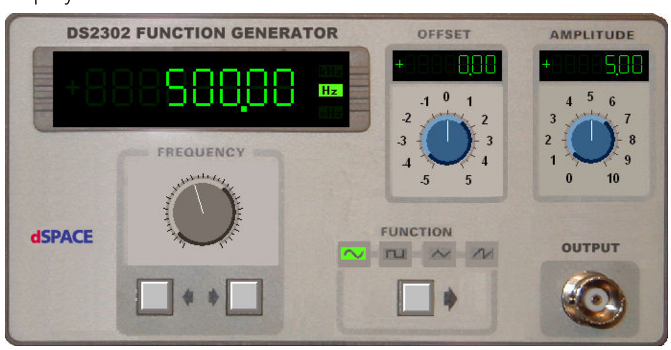
A function generator serves as an example program to demonstrate such a ControlDesk-controlled DS2302 application. The function generator allows you to switch between different waveforms, such as sine-wave, square-wave, triangular, and sawtooth signals. The signal frequency, amplitude, and offset are also variable and can be changed via the layout.

ControlDesk project

To work with this demo, a backup file (`agent100<x>.ZIP`) for ControlDesk is installed in `<RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\fgen`. It is not necessary to unzip the backup file, you can open it directly in ControlDesk (see Open Project + Experiment from Backup (ControlDesk Project and Experiment Management)).

ControlDesk layout

The predefined layout provides a set of push-buttons to select the waveform and two knobs to adjust the amplitude and offset (see illustration below). Another knob is used in conjunction with two push buttons to alter the signal frequency within the range 2 Hz … 20 KHz. The current frequency is shown in a digital display.



The output signal is fed to the D/A converter and can be observed by using an oscilloscope.

The function generator contains code performing output signal saturation. If the output value exceeds the maximum D/A converter range (i.e. ±10 V) it is saturated on the output limits (i.e. -10 V or +10 V).

Using the Digital I/O Lines (incr)

Introduction

A demo application using three digital outputs for simulating an incremental sensor is discussed below. It comes with the DS2302 software in <RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\incr.

Incremental Sensor Simulation

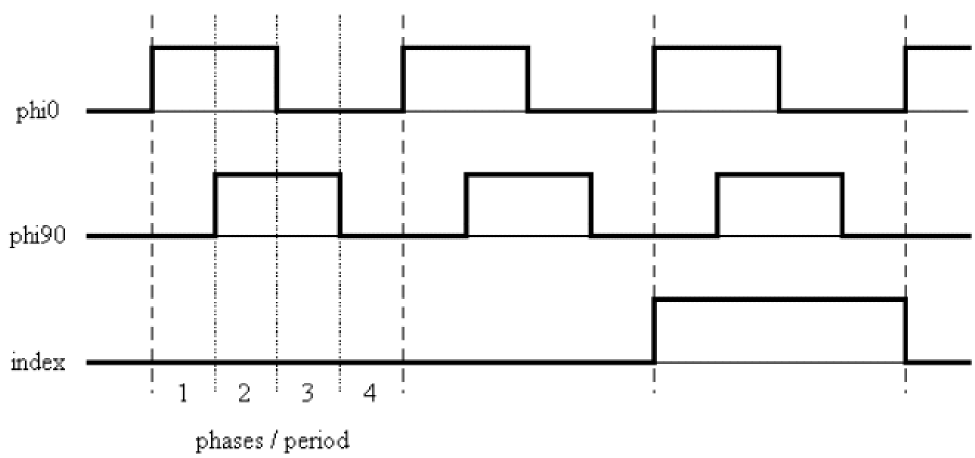
Introduction

A demo program that generates two digital outputs for the incremental sensor pulses *phi0* and *phi90* and one digital output for the index pulse.

Incremental sensor simulation

This demo program uses two digital outputs for the incremental sensor pulses *phi0* and *phi90* and one digital output for the index pulse. The illustration below shows the three incremental sensor signals for a positive spin direction. For a negative spin direction the rising edge of the signal *phi0* occurs 90 degrees after the rising edge of *phi90*.

The first pulse period in the illustration below is divided into four phases showing one of the four possible state combinations of the signals *phi0* and *phi90* each. A complete pulse period represents one incremental sensor bin. The index pulse is generated once per revolution for the duration of one pulse period.



The pulse pattern for each single output event (i.e. one of the four phases) is stored in a structure of the following type.

```
/* output pattern data structure */
typedef struct pattern_struct pattern_t;
struct pattern_struct
{
  float pos;                        /* edge angle position */
  float phi_reset;                    /* angle reset value */
  long bin_inc; /* bin count increment for index pulse gen */
  long phi0;                  /* state of phi0 output line */
  long phi90;                /* state of phi90 output line */
  pattern_t *prev;     /* link to previous pattern element */
  pattern_t *next;         /* link to next pattern element */
};
```

Four of these structure elements are needed to hold the pattern information for a complete pulse period. Each structure element is linked to its succeeding and preceding element by pointers so that the list can be run through in both directions. The last element is linked to the first one and vice versa to obtain a closed loop. The parameter phi_reset contained in each structure element is used to reset the current angle whenever the end of a pulse period is reached. This parameter must contain the value `2π/NO_OF_BINS` in the last list element and the value 0 otherwise. The `bin_inc` parameter is used to count the bins for index pulse generation, because the index pulse must be generated once per revolution. This parameter must contain the value 1 in the first list element and 0 otherwise. The C code for initializing the pattern structures is listed further on.

The timer interrupt service routine for generating the incremental sensor output signals is listed below.

```
void c_int09()        /* timer0 interrupt service routine */
{
  if (delta_phi >= 0)                  /* check direction */
  {                                 /* positive direction */
    if (phi >= cur->pos) /* check whether to put next edge */
    {
      dig_out6(cur->phi0);       /* put phi0 output signal */
      dig_out2(cur->phi90);     /* put phi90 output signal */

      /* reset angle at end of pattern */
      phi -= cur->phi_reset;
      bin_count += cur->bin_inc;       /* update bin count */
      if (bin_count >= NO_OF_BINS) bin_count = 0;
      dig_out1(bin_count == 0);    /* generate index pulse */
      cur = cur->next;      /* select next pattern element */
    }
  }
  else
  {                                 /* negative direction */
    if (phi <= cur->pos) /* check whether to put next edge */
    {
      dig_out6(cur->phi0);       /* put phi0 output signal */
      dig_out2(cur->phi90);     /* put phi90 output signal */
      cur = cur->prev;  /* select previous pattern element */

      /* reset angle at end of pattern */
      phi += cur->phi_reset;
      bin_count -= cur->bin_inc;       /* update bin count */
      if (bin_count < 0) bin_count = NO_OF_BINS - 1;
      dig_out1(bin_count == 0);    /* generate index pulse */
    }
  }
  phi += delta_phi;                /* update current angle */
}
```

The routine consists of two similar blocks, one for a positive spin direction and another for a negative direction. The actual direction depends on whether the current value of `delta_phi` is positive or negative. The differences between the two directions are that the angle reset and the bin increment values have different signs, and that the bin counter is reset to `NO_OF_BINS-1` instead of 0 for a negative direction. The `dig_out6()` (TCLK0) and `dig_out2()` (FSX0) macros are used to output the signals *phi0* and *phi90*, respectively. The `dig_out1()` macro (DX0) is used for the index pulse because the DX line is the digital output with the slowest edge rise time.

The digital I/O pins must be configured by calling the `init_dig_out1(1)`, `init_dig_out2(1)` and `init_dig_out6(1)` functions once to use the DX0, FSX0 and TCLK0 lines for digital output.

```
init();                            /* initialize hardware */
init_dig_out1(1);               /* initialize digital I/O */
init_dig_out2(1);
init_dig_out6(1);
```

The pattern structure list must be initialized in the main() routine before the timer interrupts are enabled. This is done by the piece of code listed below.

```
/* initialize output pattern data structure */
for (i = 0; i < 4; i++)
{
  pattern[i].pos = (float) i * PI2 / NO_OF_BINS / 4;
  pattern[i].phi0 = (i % 4) < 2;
  pattern[i].phi90 = ((i % 4) > 0) && ((i % 4) < 3);
  pattern[i].phi_reset = 0;
  pattern[i].bin_inc = 0;
  if (i > 0)
    pattern[i].prev = &pattern[i-1];
  if (i < 3)
    pattern[i].next = &pattern[i+1];
}
pattern[0].prev = &pattern[3];
pattern[3].next = &pattern[0];
pattern[3].phi_reset = PI2 / NO_OF_BINS;
pattern[0].bin_inc = 1;
cur = &pattern[0];
```

Most of the parameters contained in the list can be initialized in a loop. The pos structure member contains the actual angle position for the different pattern combinations of the signals *phi0* and *phi90*. Initialization of the `phi_reset` and `bin_inc` parameters and the link between the last and the first list element must be performed separately. The last instruction in the above listing sets the `cur` pointer to point to the first list element.

The angle increment value `delta_phi` is continually computed from the value `*omega`. The `*omega` variable is placed in the dual-port memory and can be changed by the master processor board.

```
...
/* initialize angle speed and angle increment */
*omega = OMEGA_DEF;
omega_max = PI2 / 4 / NO_OF_BINS / delta_t;
delta_phi = *omega * delta_t;
timer0(TS);                 /* initialize and start timer0 */
for (;;)                             /* background process */
{
  speed_check(0);           /* speed-check code for timer0 */
  if (*omega > omega_max)   /* update angle increment with */

    /* saturation for anti-windup */
    delta_phi = omega_max * delta_t;

  else if (*omega < -omega_max)
    delta_phi = -omega_max * delta_t;
  else
    delta_phi = *omega * delta_t;
}
```

The maximum value of `*omega`, called `omega_max`, depends on the current sampling rate and the number of bins per revolution. If `*omega` exceeds `omega_max`, movement through the pattern list cannot follow the current angle, which will continually grow in this case. To avoid such an angle windup, `delta_phi` is saturated if `*omega` exceeds `omega_max`.

Superposition of Signals (Fourier)

Introduction

The following demo application shows how a square wave signal is synthesized by adding appropriate sine wave signals based on a Fourier series. It comes with the DS2302 software in <RCP_HIL_InstallationPath>\Demos\DS100<x>\IOBoards\DS2302\fourier.

Fourier Synthesis of a Square Wave Signal

Introduction

A slave application generates 6 sine wave signals as the harmonics of a Fourier series of a square wave signal.

Fourier series

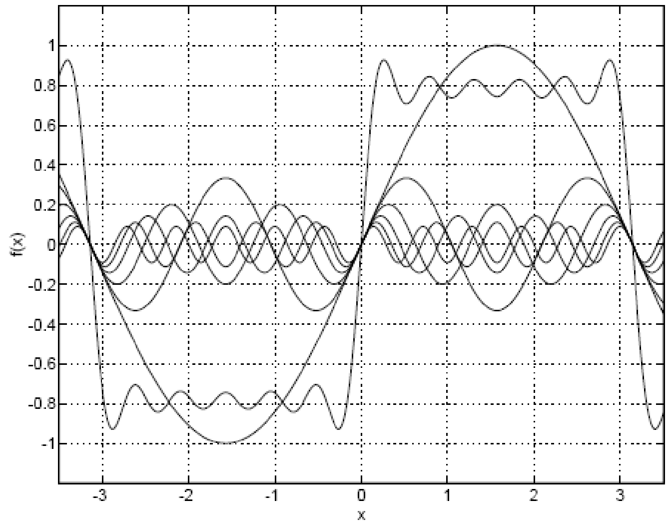
The Fourier series of a square wave signal is given by the equation

$$f(x) = \sum_{i=0}^{\infty} \frac{1}{2i-1} \sin(2i-1)x .$$

Truncation of the series after the 6th term yields

$$f(x) = \sin x + \frac{1}{3}\sin 3x + \frac{1}{5}\sin 5x + \frac{1}{7}\sin 7x + \frac{1}{9}\sin 9x + \frac{1}{11}\sin 11x .$$

The first 6 components and the resulting sum are shown in the illustration below.



Implementation

Each DS2302 channel is used to generate a single sine wave signal, i.e. channel 1 generates the basic sine wave and channels 2 … 6 generate the first 5 harmonics as given by the equations above.

The output signals of channels 2 … 6 are transferred to channel 1 via dual-port memory and added to the output value of channel 1, so the resulting square wave is available at the output of channel 1.

Since it is mandatory that all sine waves keep in phase, the individual programs must be synchronized. In this case channels 2 … 6 are synchronized to channel 1 by using the same method as used in the crankshaft/camshaft example described in DSP Synchronization by Interrupts on page 124.

You can comment out the line

```
int_xf0();     /* synchronize channels 2..6 */
```

in the `sin5_1.c` file to test the necessity of synchronization. Use the `clsqr.bat` batch file to compile the source files and download the application to the DS2302. The individual sine wave signals will get slowly out of phase due to rounding errors caused by floating-point arithmetic, and thus the shape of the output signal will change.

The application source file for the sine wave generator `sin5_1.c` is slightly different from the other five generators due to the synchronization code. The generators for the harmonics are created from a single source file (`sin5_x.c`) by using the symbolic constant `HARM` specified in the compiler command line (cf. the batch file `clsqr.bat`).

On-Board Dual-Port Memory Data Transfer

Transferring Data with the On-Board DPMEM

Introduction

The dual-port memory (DPMEM) of each DSP contained on the DS2302 board can be accessed by the master processor board as well as by every on-board DSP.

Transferring data

The dual-port memories can be used for on-board data transfer between the DSPs as well as for processor board to DS2302 data transfer.

> **Note**
>
> The last 16 dual-port memory locations in the offset range 0x3FF0 … 0x3FFF are used by the software environment of the DSP. Do not use it within your application. If you are using messages, an additional memory range is reserved. Refer to Message Functions on page 100.

Appropriate definitions to access the individual dual-port memories from each DSP are already in the `ds2302.h` header file that comes with the DS2302 software.

The local dual-port memory of each channel can be accessed at address 0x400000 (as seen from the DSP). This address is defined as the symbolic constant `DP_MEM_BASE`.

In addition, each DSP contained on the DS2302 board can access the dual-port memories of the other five channels. They are accessible at addresses 0x040000 … 0x2C0000 (channel 1 … channel 6). You can use the symbolic constants `DP_MEM_1_BASE` … `DP_MEM_6_BASE` instead.

Note that each DSP can access its local dual-port memory at two different addresses, i.e. at address 0x400000 (`DP_MEM_BASE`) and at the address where it can be accessed by the other on-board DSPs (`DP_MEM_x_BASE`).

The `ds2302.h` header file contains pointer definitions (`dp_mem` and `dp_mem_1` … `dp_mem_6`) which point to the beginnings of the dual-port memory sections. They can be used to access the dual-port memories like arrays. The pointers are of union type so they can be used in conjunction with integer values as well as with floating-point values. Some examples are given below.

```
dp_mem[0].f = angle;               /* write float value to 1st dp-mem location */

angle1 = dp_mem_1[0].f;                 /* read value from 1st dp-mem location */
                                       /* of channel 1 by another on-board DSP */

count = dp_mem[3].i;            /* read integer value from 3rd dp-mem location */

dp_mem_5[10].i = count;               /* write value into 10th dp-mem location */
                                                             /* of channel 5 */

for (i = 0; i < 10; i++)                         /* copy first 10 dp-mem values */
  dp_mem[i].f = dp_mem_3[i].f;               /* of channel 3 into local dp-mem */
```

You may of course declare your own individual pointers to arbitrary dual-port memory locations.

```
float *rpm = (float *) (DP_MEM_1_BASE + 5);
...
d_phi = *rpm * 2.0 * PI / 60.0 * d_t;
```

In the above example, the value `*rpm` is obtained from the 5th dual-port memory location of channel 1.

Note that an initialized pointer requires 3 words of memory in the `.cinit` section and an additional word in the `.bss` section. Thus it is recommended to use symbolic pointers instead.

```
#define rpm ((float *)(DP_MEM_BASE + 5))
```

Dual-Rate Applications

Introduction

The TMS320VC33's second timer can be used to implement a second timer interrupt service at a different sampling rate.

In some cases it is useful to run different program parts at different sampling rates.

Implementing Dual-Rate Applications

Introduction

To run different program parts at different sampling rates, the TMS3201VC33's second on-chip timer can be used to implement a second timer interrupt service routine.

Dual-rate application

The following pieces of code can be used as a basic frame implementing a dual-rate application.

```
void c_int09()           /* timer0 interrupt service routine */
{
  ...                   /* execute code at high sampling-rate */
}

void c_int10()           /* timer1 interrupt service routine */
{
  global_enable();             /* make routine interruptable */
  ...                   /* execute code at slow sampling-rate */
  global_disable();              /* disable interrupts again */
}

void main()
{
  ...
  timer0(10e-6);            /* initialize timer0 for 100 KHz */
  timer1(1e-3);               /* initialize timer1 for 1 KHz */
  ...
}
```

Note

Interrupts must be globally enabled within the slower timer interrupt service routine to allow interruption by the fast timer interrupt. This is necessary especially if the slower timer interrupt service routine is very time-consuming and if no jitter is allowed in the fast timer interrupt service routine. Interrupts are globally disabled by default in interrupt service routines. Interrupts can be enabled by the `global_enable()` macro and should be disabled again before leaving the interrupt service routine.

X/Y Image Generator (draw)

Generating X/Y Signals for Drawing

Introduction

This application generates x/y signals for drawing a two-dimensional image on an oscilloscope screen, in this case the dSPACE logo is used. The aim is to show the capabilities of the DDS board. There is no reference to the slave applications usually needed in simulation environments.

System requirements

This application requires a DS2302 DDS board and a processor board connected to the DS2302 via PHS bus.

The analog DS2302 outputs OUTA and OUTB must be connected to the horizontal and vertical inputs of an oscilloscope in X/Y mode, respectively. Adjust both amplitudes to 1 V/cm.

Running the demo

To start the demo, simply download the processor board application `draw<xxxx>` to the processor board by typing

```
down1005 draw1005.ppc
```

or

```
down1006 draw1006.x86
```

or

```
down1007 draw1007.ppc
```

The DS2302 applications are encoded in the master processor board object module and is downloaded to the DS2302 by the master processor board automatically.

Program modules

The demo comprises the following program modules

- `DRAW100<x>`: Master processor board program, downloading of DS2302 applications, x/y image table initialization
- `DRAW_X`: DS2302 channel A program: x/y data interpolation, x data scaling and output
- `DRAW_Y`: DS2302 channel B program: y data scaling and output
- `DRW_ZOOM`: DS2302 channel C program: periodic zooming
- `DRW_NOIS`: DS2302 channel D program: additive Gaussian noise
- `DRW_SPIN`: DS2302 channel E program: vertical spinning

The `DRW_ZOOM`, `DRW_NOIS`, and `DRW_SPIN` modules perform dynamic scaling and adding of noise. They are optional and each of them can be used individually.

Execution Time Profiling

Measuring Execution Times

Introduction

You can measure the execution times using macros.

Measuring execution times

The `tic3x.h` header file contains the `ticx_start()` and `ticx_read()` macros for execution time measurement of arbitrary code parts in timer interrupt service routines. A simple master processor board program can be used to display the result.

In the example below, the minimum and maximum execution times of a block of code are evaluated and stored in the first two dual-port memory locations. An excerpt from the DSP program is listed below.

If the macros do not correspond to the currently active timer (i.e. timer0 in the example), the `ticx_init()` macro must be executed first in the `main()` function to initialize and start the appropriate timer.

Note

If you include the `tic3x.h` header file, the `tic3x.obj` object module containing some global variables is linked from the object library. This requires 26 words of memory. If memory constraints become a problem, it may be better to use the older execution time measurement macros `timer_count()` and `time_elapsed()`, defined in the `util2302.h` header file.

```
float *min_time = (float *) (DP_MEM_BASE);
float *max_time = (float *) (DP_MEM_BASE + 1);

void c_int09()           /* timer0 interrupt service routine */
{
  float exec_time;
  ...
  tic0_start();          /* start execution time measurement */
  ...            /* code subject to execution time profiling */
  exec_time = tic0_read();             /* read execution time */
  if (exec_time < *min_time)     /* update min and max values */
    *min_time = exec_time;             /* in dual-port memory */
  else if (exec_time > *max_time)
    *max_time = exec_time;
  ...
}
```

The following declarations and instructions can be used in a host program to read the execution time values from the DS2302's dual-port memory and print them on the screen.

```
#define MIN_ADDR (DS2302_DP_MEM_BASE)
#define MAX_ADDR (DS2302_DP_MEM_BASE + 1)
UInt32 val;
Float32 min, max;
...
DS2302_read(board_index, DS2302_CH1, MIN_ADDR, &val);
min = DSP_cvt_ti_to_ieee(val);
printf("DS2302 minimum execution time: %e sec\n", min);
```

```
DS2302_read(board_index, DS2302_CH1, MAX_ADDR, &val);
max = DSP_cvt_ti_to_ieee(val);
printf("DS2302 maximum execution time: %e sec\n", max);
```

The execution time information can also be read by a processor board and observed via ControlDesk.

For execution time measurement of the entire timer interrupt service routine, the SPEEDCHK utility is more convenient. Refer to SPEEDCHK Utility on page 148.

Utilities

Where to go from here

Information in this section

SPEEDCHK Utility

Introduction

The SPEEDCHK utility is for evaluating execution time information about the timer interrupt service routine in DS2302 application programs.

Where to go from here

Information in this section

Basics of SPEEDCHK

Introduction

The SPEEDCHK utility is for evaluating execution time information about the timer interrupt service routine in DS2302 application programs.

Basics

The resolution is 1 timer tick (i.e. 26.66 ns for a 150 MHz DS2302) and the maximum error is +1 timer tick.

Since many application programs comprise different program paths of different lengths, SPEEDCHK evaluates the minimum and maximum execution times. The minimum and maximum number of timer ticks are actually computed by the macro `speed_check()` on the DSP (refer to Preparing Application Programs for SPEEDCHK on page 149) and transferred to the host through dual-port memory locations at offset 0x03FFD and 0x03FFE of each channel.

Preparing Application Programs for SPEEDCHK

Introduction

To use the SPEEDCHK utility, you must include some code in the background loop of a DS2302 application program.

Using SPEEDCHK

To use the SPEEDCHK utility, some code must be included in the background loop of a DS2302 application program. This is simply done by including the `speed_check(i)` macro from the `util2302.h` header file in the forever loop of the `main()` routine. The i parameter selects the appropriate TMS320VC33's on-chip timer (i = 0 for timer0, i = 1 for timer1). It must match the timer actually used to generate the sampling clock interrupts (cf. the following example for timer0).

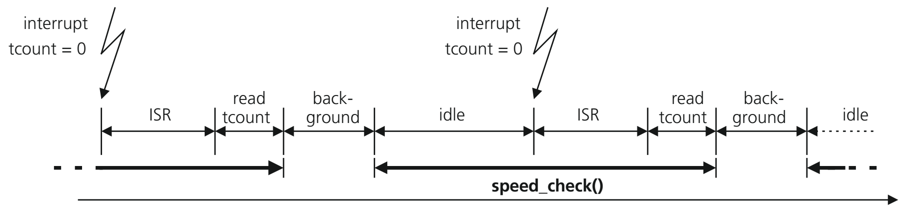
```
timer0(TS);                               /* initialize timer0 */

for (;;)
{
  speed_check(0);       /* include SPEEDCHK code for timer0 */
}
```

Note

- The `speed_check()` macro contains the assembly instruction `idle`, which waits for an interrupt. Thus any additional code in the background loop is executed only once each time an interrupt is received. The `idle` instruction also sets the GIE bit in the VC33's ST register, which enables the global interrupts.
- `speedchk` does not work properly if any other interrupt except for a single timer interrupt is used in the application.

The following illustration shows how speedchk works:



The timer counts up until the counter reaches the initialized compare value. After that, an interrupt is generated and the timer counter (tcount) is cleared to '0'. If the ISR execution has been finished, the `speed_check()` macro reads in the background the current timer counter value and knows then the execution time of the ISR, which is written into the dual-port memory. After that, the background loop is executed until the next `speed_check()` call, which again will execute the IDLE instruction, which is waiting for the next timer interrupt.

The SPEEDCHK Host Program

SPEEDCHK host program

The SPEEDCHK host program reads the minimum and maximum number of timer ticks from one or all of the DS2302 channel's dual-port memories. The timer ticks and corresponding execution time values are displayed on the screen.

Note

Ensure that no master processor board accesses the DS2302 while SPEEDCHK is invoked. Because the interface is switched to host access without arbitration, this may cause unpredictable system behavior and even may block the system.

Tip

Instead of the SPEEDCHK host program, you can use the `ds2302_speedchk()` master processor function to observe the execution times. For further information, refer to ds2302_speedchk (DS2302 RTLib Reference 📖).

Using SPEEDCHK

The program is invoked by typing

`speedchk [channel] [/p portbase]` (PC/AT bus version)

`speedckn [channel] [/p portbase]` (PC/AT network version)

where the optional parameter `channel` can specify one of the DS2302's channels 1 … 6 . If no channel is specified, SPEEDCHK lists the execution time values for all 6 channels.

The optional parameter `/p portbase` can be used to specify a non-default DS2302 I/O port base address. This is necessary if you have changed the DS2302's DIP switch setting to a non-default I/O-port base address. If no port base address parameter is specified, SPEEDCHK uses the default I/O-port base address 0x0320.

If a DS2302 channel does not execute the `speed_check()` macro or does not run any application program at all, SPEEDCHK displays the message "evaluation failure" for that particular channel.

SPEEDUP Utility

Introduction

To optimize assembly code.

Where to go from here

Information in this section

Basics of SPEEDUP

Basics

The SPEEDUP utility is available to automatically perform optimizations on the assembly level in DS2302 application programs. The optimizations include removing of unnecessary context save and restore instructions from the timer interrupt service routine. Some unnecessary code for floating-point-to-integer type conversion in conjunction with data output to the D/A converter is also removed. The optimizations are discussed in detail in the following subsections.

Invoking SPEEDUP

Introduction

You can start the SPEEDUP utility manually.

Starting the SPEEDUP utility manually

Normally, the SPEEDUP utility is automatically invoked by the compile/link tool `cl230x.exe` if the command line option '/s' is specified. However, if you need a different behavior you can invoke the program `speedy.exe` directly. The calling syntax is

```
speedy [-v] [-k] [-o outfile] <asmfile>
```

where the optional parameter '-v' selects generation of verbose information about register usage, subroutine calls and register replacements performed by SPEEDUP.

The parameter '-k' allows removed assembly instructions to be kept as comments.

The resulting optimizer output is written to a file named speedup.out by default. An arbitrary output file name can be specified by using the command line parameter '-o outfile'.

Note

The assembly source file must be specified including the suffix .asm.

Context Save and Restore

Introduction

Most DS2302 application programs consist of a single interrupt service routine and contain only little or no code at all in the background loop. Thus most of the context save and restore instructions (PUSH / POP) performed at the beginning and at the end of interrupt service routines are dispensable and can be removed to save execution time.

Removing PUSH and POP instructions

SPEEDUP removes the PUSH and POP instructions from interrupt service routines (for example, `c_int09()` or `c_int10()`). If the command line option '-k' is used, the removed instructions are kept as comments, as shown in the following example.

```
_c_int09:
            PUSH     ST
*o*         PUSH     R0
*o*         PUSHF    R0
*o*         PUSH     R1
*o*         PUSHF    R1
*o*         PUSH     R2
*o*         PUSHF    R2
*o*         PUSH     AR0
*o*         PUSH     AR1
*o*         PUSH     AR2
            ...
*o*         POP      AR2
*o*         POP      AR1
*o*         POP      AR0
*o*         POPF     R2
*o*         POP      R2
*o*         POPF     R1
*o*         POP      R1
*o*         POPF     R0
*o*         POP      R0
            POP      ST
            RETI
```

Interrupt service routines are searched for registers that are already used by the `main()` routine or by another interrupt service routine. If any register conflicts are detected, registers are substituted by other unused registers in interrupt service routines, if possible. If no remaining unused register is available, the context save / restore of particular registers remains unaffected by SPEEDUP.

Register substitution in interrupt service routines is performed in order of their appearance in the assembly source code. Thus, the first interrupt service routine is assigned the highest optimization priority.

The saving of the status register (ST) is not affected by SPEEDUP.

Any instructions that use the auxiliary register AR3, which is used as the frame pointer in Texas Instruments C compiler generated programs, are not changed by SPEEDUP.

Detailed information about the individual register substitution actually performed can be obtained by using the command line switch '-v' of the SPEEDUP program `speedy.exe`.

Floating-Point-to-Integer Type Conversion

Introduction

The SPEEDUP utility can detect and optimize the casting of floating point values to integer values.

Note

SPEEDUP can detect and optimize the conversion sequence only, if a TI compiler version 4.7 or lower is used.

Casting floating point to integer value

Whenever a floating-point value is cast to an integer value, a FIX instruction is used to perform floating-point to integer type conversion. The C compiler uses the FIX instruction for such conversions, which rounds towards negative infinity, followed by a 4-instruction sequence to correct negative values.

In DS2302 application programs, floating-point-to-integer type conversion is frequently needed in conjunction with data output to the on-board D/A converters. In this case the correction of negative values is not important due to the D/A converter's limited precision.

If SPEEDUP detects such a code sequence in conjunction with the keyword *@_dac* in the following load instruction, the extra instructions are removed (cf. the example below).

```
            FIX     R1,R3
*o*         NEGF    R1
*o*         FIX     R1
*o*         NEGI    R1
*o*         LDILE   R1,R3
            LDI     @_dac1,AR2
            STI     R3,*AR2
```

Note

The correction sequence for floating-point-to-integer type conversion can also be suppressed for faster execution by using the CL30 option *-mc*. However, this affects every floating-point-to-integer type conversion.

Optimization Limitations

Limitations

When using SPEEDUP, note the following limitations:

- If an application program contains function calls, SPEEDUP uses the exact register usage information for local functions and for the run-time support arithmetic routines from the run-time library `rts30.lib` (div, mod, etc.). All other functions are assumed to use all registers.

- The register usage of local functions is evaluated in order of their appearance, i.e. if a function is called prior to its declaration, usage of all registers is assumed.
- The SPEEDUP utility has been designed to be applied to Texas Instruments C compiler generated assembly source code. In the case of hand-coded assembly programs or inline assembly statements, macro definitions and substitution symbols must not be used in conjunction with SPEEDUP.
- SPEEDUP should be applied only to application programs that consist of a single assembly source file, except for the standard object modules from the object library `ds230x.lib`. In the case of modular programs, special care must be taken that no externally linked object code can be interrupted by interrupt service routines, that have been optimized by SPEEDUP. Otherwise register conflicts can cause unpredictable system behavior.
- Interrupt service routines optimized by SPEEDUP are no longer reentrant. So whenever interrupts are enabled in an interrupt service routine in order to make it interruptable, you must ensure that the interrupt service routine is never interrupted by itself. Otherwise registers will be corrupted, which will cause unpredictable results. In the case of timer interrupt service routines, the sampling rate must be appropriately chosen to make sure that one interrupt service has finished, before the next interrupt is received. To be on the safe side, first select a sufficiently large sampling period and then use SPEEDCHK to evaluate the actual execution time.

Details on the TMS320VC33 DSP Hardware

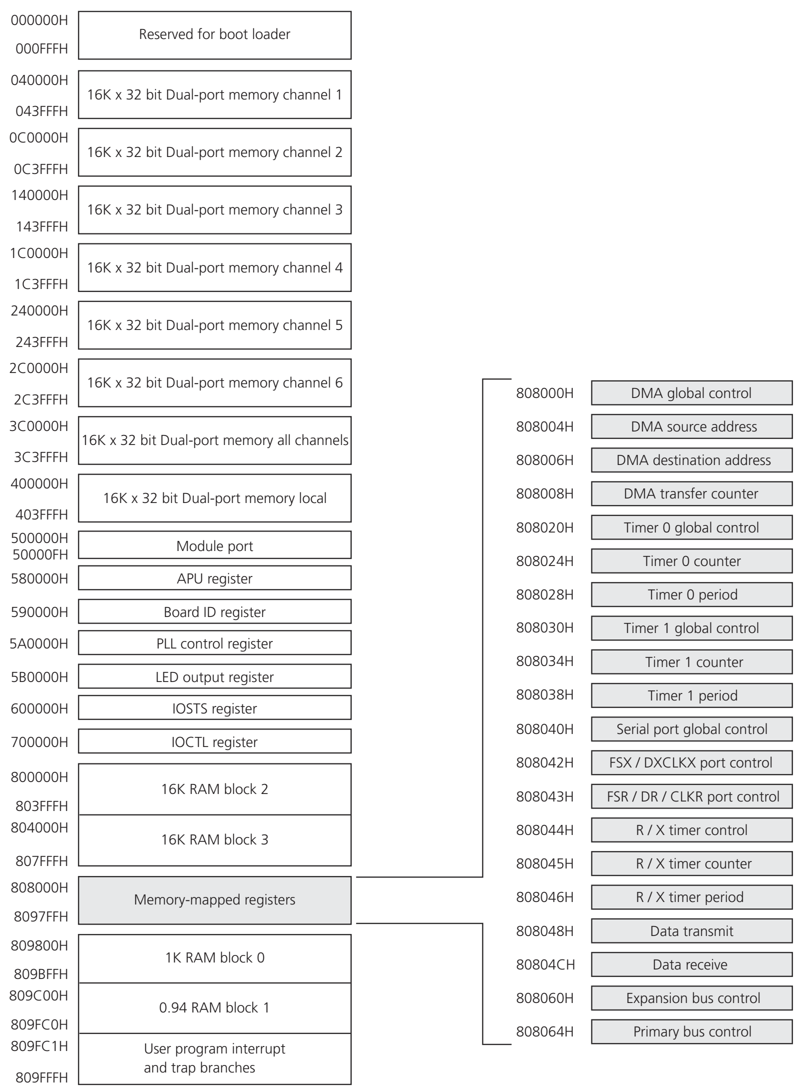
Where to go from here

Information in this section

DSP Memory Map

Introduction

The TMS320VC33 supports a linear address space of 16 M 32-bit words. The DS2302 contains 16 KWords of two wait state dual-port RAM located at addresses 400000H ... 403FFFH. In addition, the six 4 KWords on-board dual-port memories are mapped from addresses 040000H ... 2C3FFFH. The module port is located in the address range 500000H ... 50000FH. The I/O control register (IOCTL) is located at address 700000H and the I/O status register (IOSTS) at address 600000H. The illustration below shows the complete TMS320VC33 memory map.



PHS Bus Interrupt Generation

PHS bus interrupt generation

The CLKX0 signal of the TMS320VC33's serial port is used for PHS bus interrupt generation. The CLKX0 pin must be initialized for output. The Interrupt Control Unit must be initialized to edge-trigger mode for proper operation. The CLKX0 signal is set to high by default. To generate an interrupt, the CLKX0 signal must be set to zero for at least 100 ns. Then the CLKX0 signal must be reset to one.

External DSP Interrupts

Introduction

The TMS320VC33 has four interrupt lines. One of them can be used as an external interrupt input.

Interrupt lines

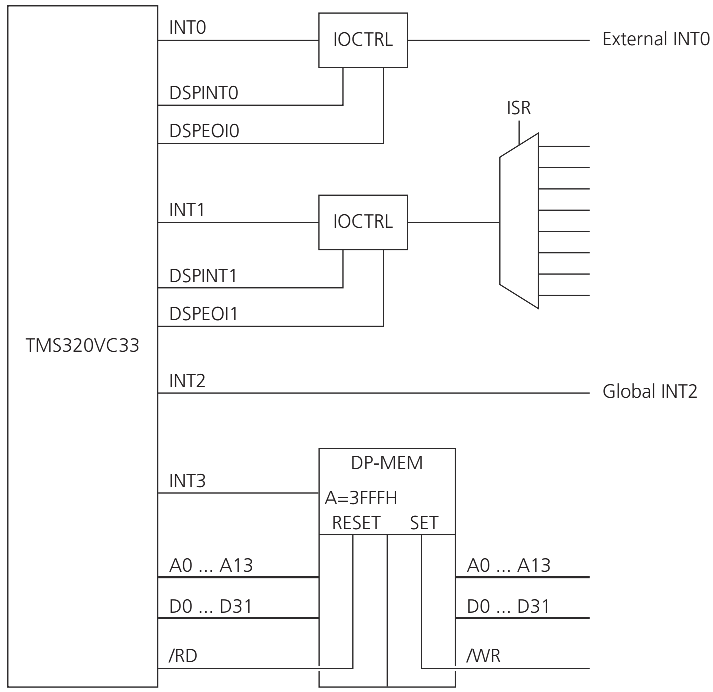
INT0 is the interrupt input with the highest priority. It is available on the I/O connector P4 and can be used as an external interrupt input.

INT1 is connected to one of the eight interrupt sources, depending on the contents of the interrupt select register.

INT2 is a global interrupt which enables the host to generate an interrupt to all 6 DSPs simultaneously.

INT3 is activated by writing to dual-port memory location 3FFFH.

The following illustration shows the external DSP interrupts.



The table below shows the corresponding DSP interrupt lines.

Source	**Interrupt line**
External interrupt input	INT0
One of eight on-board interrupt sources	INT1
Global host interrupt	INT2
Other DSPs, PC or PHS bus master	INT3

For further information, refer to Digital I/O (PHS Bus System Hardware Reference 📖).

I/O Control Register (IOCTL)

Introduction

The I/O control register (IOCTL) is a 2-bit read/write register used to control and query the states of the two interrupt lines (INT0 and INT1) of each TMS320VC33. The IOCTL register is located at memory address 600000H. The end-of-interrupt bits can be set to release the TMS320VC33's interrupt input. The DSPINTx flags display the state of the interrupt lines.

Reading the IOCTL register

The I/O-control register contains two DSPINT flags.

	D31	D30	D29 - D0
IOCTL	DSPINT1	DSPINT0	unused

The table below shows the format of the IOCTL register when read.

Bit	Name	Function
30	DSPINT0	TMS320VC33 interrupt line 0. DSPINT0 = 1 indicates an active interrupt request. DSPINT0 = 0 indicates that the DSP has finished the interrupt service. DSPINT0 is set when an interrupt is pending on the external interrupt input. DSPINT0 is cleared by setting the DSPEOI0 bit in the IOCTL register.
31	DSPINT1	TMS320VC33 interrupt line 1. DSPINT1 = 1 indicates an active interrupt request. DSPINT1 = 0 indicates that the DSP has finished the interrupt service. DSPINT1 is set when an interrupt was generated by one of the other DSPs. DSPINT1 is cleared by setting the DSPEOI1 bit in the IOCTL register.

Writing the IOCTL register

The IOCTL register contains the two end-of-interrupt bits.

	D31	D30	D29 - D0
IOCTL	DSPEOI1	DSPEOI0	unused

The table below shows the format of the IOCTL register during write operations.

Bit	Name	Function
30	DSPEOI0	End of interrupt line. Writing a 1 resets the DSPINT0 flag in the IOCTL register and the respective DSPINT0 line. Writing a zero has no effect. Before leaving the corresponding interrupt routine with the RETI instruction, this flag must be set to clear the TMS320VC33's interrupt input.
31	DSPEOI1	End of interrupt line. Writing a 1 resets the DSPINT1 flag in the IOCTL register and the respective DSPINT1 line. Writing a zero has no effect. Before leaving the corresponding interrupt routine with the RETI instruction, this flag must be set to clear the TMS320VC33's interrupt input.

I/O Status Register (IOSTS)

Introduction

The I/O status register (IOSTS) is a 2-bit read-only register used to monitor the external input INT0 and the external input INT0 from the adjacent channel. The IOSTS register is located at memory address 700000H. The table below shows the contents of the I/O status register for each channel.

	D31	D30	D29 - D0
Channel 1	INT0 2	INT0 1	unused
Channel 2	INT0 1	INT0 2	unused
Channel 3	INT0 4	INT0 3	unused
Channel 4	INT0 3	INT0 4	unused
Channel 5	INT0 6	INT0 5	unused
Channel 6	INT0 5	INT0 6	unused

APU Register

Introduction

The APU interface register gives information on the APU interface.

Valid board version

The information on the APU register is only valid for DS2302-04 boards.

Memory address

The APU Register is located at memory address 0x580000.

APU interface register

The following illustration shows the APU interface register:

	D31 … D18	D17	D16	D15 … D0
APU	Reserved	PH VAL	X16	Data

Bit(s)	Name	DSP	Description
31 … 18	Reserved	R/–	The bits are reserved. Write accesses do not have an effect, read accesses deliver unpredictable results.
17	PHVAL		Master available ▪ 0: APU bus master is active ▪ 1: APU bus master is NOT active
16	X16		16 bit mode support ▪ 0: 13 bit mode

Bit(s)	Name	DSP	Description
			▪ 1: 16 bit mode
15 … 0	Data		16-bit APU data ▪ Data[15] = MSB ▪ Data[0] = LSB When APU interface is used in 13-bit mode, Data[3] bit represents the LSB and Data[2 … 0] bits have no function.

The APU interface of the DS2302-04 board only acts as a slave device. So, the APU register is read-only for all DSP.

Board ID Register

Introduction

The board ID register contains information on the board type and revision.

Board version

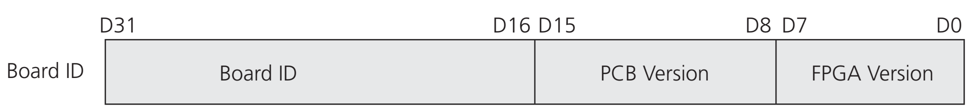
The information on the board ID register is only valid for DS2302-04 boards.

Memory address

The board ID register is located at memory address 0x590000.

ID register

The following illustration shows the board ID register:



Bit(s)	Name	DSP	Description
31 … 18	Board ID	R/–	Fix to 0x2302
15 … 8	PCB Version		PCB version; starts with 0x04 for DS2302-04 boards.
7 … 0	X16		Currently flashed FPGA version

PLL Control Register

Valid board versions

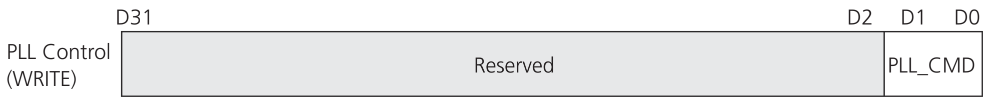
The information on the PLL control register is only valid for DS2302-04 board.

Memory address

The PLL control register is located at memory address 0x5A0000.

Write

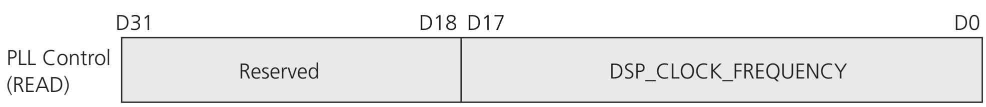
The following illustration shows the PLL control register (write):



Bit(s)	Name	DSP	Description
31 … 2	Reserved	–/–	–
1 … 0	PLL_CMD	–/W	PLL command ▪ 00: sets the DSP clock to 30.0 MHz ▪ 01: sets the DSP clock to 37.5 MHz ▪ 10: sets the DSP clock to 50.0 MHz ▪ 11: sets the DSP clock to 75.0 MHz Note: The DSP internally runs with the double frequency.

Read

The following illustration shows the PLL control register (read):



Bit(s)	Name	DSP	Description
31 … 2	Reserved	–/–	–
1 … 0	DSP_CLOCK_FREQUENCY	R/–	Current DSP clock frequency (kHz)

LED Output Register

Valid board version

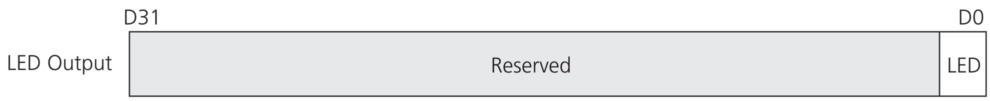
The information on the LED output register is only valid for DS2302-04 boards.

Memory address

The LED output register is located at memory address 0x5B0000.

LED output register

The following illustration shows the LED output register:



Bit(s)	Name	DSP	Description
31 … 1	Reserved	R/–	The bits are reserved. Write accesses do not have an effect, read accesses deliver unpredictable results.
0	LED	R/W	Control bit for (high active) green DSP LED

S

T

U

X