TMS320 DSP
DESIGNER'S NOTEBOOK

Accessing TMS320C54x Memory-Mapped Registers in C– C54XREGS.H

APPLICATION BRIEF: SPRA260

Leor Brenman
Digital Signal Processing Products
Semiconductor Group

Texas Instruments
May 1994

TEXAS INSTRUMENTS

TRADEMARKS

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Examples

Accessing TMS320C54x Memory-Mapped Registers in C– C54XREGS.H

Abstract

This document describes how to access the ’C54x memory-mapped registers in C. Several code examples are given.

Design Problem

How do I access the 'C54x memory-mapped registers in C?

Solution

Accessing most of the 'C54x registers from C is easily accomplished using pointers since most of the registers are memory mapped. The most common reason for accessing memory-mapped registers is to control the 'C54x peripherals. Refer to the`TMS320C54x User's Guide` for a list of the memory-mapped registers and their associated addresses. As an example, the 'C54x serial port 0 control register, SPC0, memory mapped at address 0x0022, could be declared in C as follows:

```
volatile unsigned int *SPC0_REG =
                    (volatile unsigned int *) 0x0022;
```

Note the volatile modifier since this register changes independent of program control. The register can be written to and read from as follows:

```
*SPC0_REG = 0xc8          ; /* Load SPC0 with 0xc8 */
currentXRDYValue = *SPC0_REG & 0x800
                       ; /* Check XRDY bit of SPC0 */
```

However, this does not lead to very readable code. By using bit-field data structures to describe the bit fields of the register, more readable code can be developed. For example, consider the data structure shown in Figure 1, SPC_REG, for the serial port control registers, SPC0 and SPC1.

Example 1. Code Listing

```
typedef union
{
   unsigned int intval;
   struct
   {
      unsigned int free      :1 ; /*Free run*/
      unsigned int soft      :1 ; /*Soft*/
      unsigned int rsrfull   :1
                             ; /*Rec Shift Reg Full*/
      unsigned int xsrempty  :1
                             ; /*Xmt Shift Reg Emty*/
      unsigned int xrdy      :1 ; /*Transmit Ready*/
      unsigned int rrdy      :1 ; /*Receive Ready*/
      unsigned int in1       :1 ; /*Input 1*/
      unsigned int in0       :1 ; /*Input 0*/
      unsigned int rrst      :1 ; /*Receive Reset*/
      unsigned int xrst      :1 ; /*Transmit Reset*/
      unsigned int txm       :1 ; /*Transmit Mode*/
      unsigned int mcm    :1 ; /*Clock Mode*/
      unsigned int fsm    :1 ; /*Frame Synch Mode*/
      unsigned int fo     :1 ; /*Format*/
      unsigned int dlb    :1 ; /*Dig Loopback Mode*/
      unsigned int r_0    :1 ; /*Reserved*/
   } bitval;
} SPC_REG;
```

The bit XRDY can now be read as follows:

```
volatile SPC_REG *spc0Ptr =
                       (volatile SPC_REG *) 0x0022;
currentXRDYValue = spc0Ptr->bitval.xrdy;
```

The TMS320 BBS contains the self-extracting file, C54XREGS.EXE. This file contains a C header file, C54XREGS.H, that can be included (i.e., #included) in your C programs to assist in accessing 'C54x peripheral registers as well as all of the 'C54x memory-mapped registers, where appropriate bit-field data structures are also defined. The remainder of this document will describe its usage.

To use C54XREGS.H simply include the file in your C program. Each memory-mapped register has two entities associated with it: (1) a macro that defines its address and (2) a type definition that describes the bit fields and the memory-mapped register. The macros for the address have two components for each register: one for the actual address and one to typecast the address as a pointer to a data structure that defines the memory-mapped register. The following code segment describes the address macros for the serial port 0 control register:

```
#define SPC0_BASE 0x0022
#define SPC0_ADDR ((volatile SPC_REG *)
                              ((char *) SPC0_BASE))
```

Two different methods have been used to type define the registers. For registers with bit fields, such as the serial-port-control registers, SPC0 and SPC1, and interrupt-mask register, IMR, data structures have been created that comprise a union of a 16-bit integer component, named `intval`, and a bit-field component, named `bitval`. The bit-field data structure for the serial port control register given above is such an example. Registers that have no bit field definition such as the serial port 0 receive register, DRR0, are defined as either signed or unsigned integers or characters.

To access registers defined as bit-field data structures, use the following syntax:

```
/*Set FSM, XRST, and RRST bits of the SPC0 register*/
SPC0_ADDR->intval = 0xc8;
```

To increase the readability of such assignments, macros for setting the bits have also been defined. The following example illustrates the use of these macros to accomplish the same thing:

```
/*Set FSM, XRST, and RRST bits of the SPC0 register*/
SPC0_ADDR->intval = FSM | XRST | RRST;
```

The previous example sets the serial port for Frame Sync Mode and resets the transmit and receive sides of serial port 0. Additional macros have been defined such that the user only need type SPC0 instead of SPC0_ADDR->intval. Therefore the last example could be expressed as follows:

```
SPC0 = FSM | XRST | RRST;
```

Alternatively, the bit-fields could have been used as follows to accomplish the same task:

```
SPC0_ADDR->bitval.fsm = 1;
SPC0_ADDR->bitval.xrst = 1;
SPC0_ADDR->bitval.rrst = 1;
```

To access registers that are not defined as bit-field data structures, use the following syntax:

```
*DXR0 = outputValue;
```

The previous example writes `outputValue` to the serial port 0 transmit register.

To declare a pointer to the serial port 0 control register, use the following syntax:

```
volatile SPC_REG *spcr0 = SPC_ADDR;
```

The register is accessed as follows:

```
spcr->intval = FSM | XRST | RRST;
fsmbit = spcr->bitval.fsm;
```