Texas Instruments

Application Report

SPRA378 - April 2002

Stack Mode Initialization/Reset on the TMS320C55x DSP

Monideep Mukherjee *C5000 Hardware Applications*

ABSTRACT

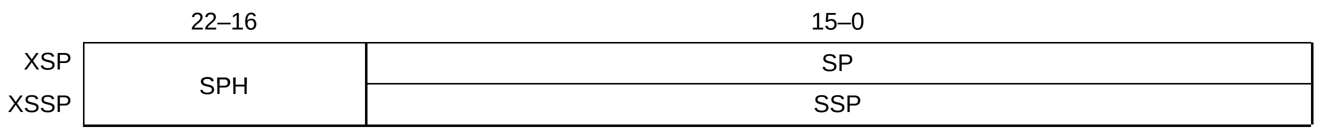
This document contains information and examples of how to initialize and use the several stack modes that are available on the TMS320C55x™ DSP.

Contents

List of Figures

1 Data and System Stacks

The TMS320C55x includes two 16-bit software stacks known as the user (or data) stack and the system stack. The user stack is the normal stack where the user can push and pop values. The user stack includes a stack pointer (SP) that contains the lower 16 bits of the return address for calls or interrupts. It also holds the values of anything pushed onto the stack. The system stack exists due to the compatibility with the TMS320C54x™ series of processors. The system stack incorporates a system stack pointer (SSP) that contains the *upper* bits of the return address for calls and interrupts. Once the user sets up the system stack, it is left alone because it does not contain any user-pushed data. Figure 1 shows the relationship between the system and user stack pointers.



Legend: SPH – High 7 bits of XSP and XSSP (this is not modified during normal stack operations)
XSP – 23-Bit Extended Data Stack Pointer
XSSP – 23-Bit Extended System Stack Pointer

Figure 1. Stack Pointers

2 Stack Configurations

The TMS320C55x DSP provides three possible stack configurations with slow and fast returns. The difference between the fast-return and slow-return process is how the CPU saves and restores the value of two internal registers: the program counter (PC) and the loop context register.

The PC holds the 24-bit address of the 1 to 6 bytes of code being decoded in the I unit. When the CPU performs an interrupt or call, the current PC value (return address) is stored, and then PC is loaded with the start address of the interrupt service routine or called routine. When the CPU returns from the routine, the return address is transferred back to the PC, so that the interrupted program sequence can continue as before.

In the slow-return process, the return address and the loop context are stored to the stacks in memory. When the CPU returns from the subroutine, the speed at which these values are restored is dependent on the speed of the memory accesses.

In the fast-return process, the return address is saved in the return address register (RETA) and the loop context is saved in the control-flow context register (CFCT). The RETA and CFCT registers can be read or written as a pair with the 32-bit load and store instructions.

2.1 Dual 16-Bit Stack With Fast Return

In this mode, the data and the system stack are independent. When an access is made to the data stack, the stack pointer (SP) is modified, but the system stack pointer (SSP) is not. The RETA and CFCT registers are used to implement a fast return. This mode is the optimal stack mode operation: it saves both cycles and memory. Note that for nested calls, RETA and CFCT are both saved.

2.2 Dual 16-Bit Stack With Slow Return

In this mode, the data stack and the system stack are independent; when the data stack is accessed, the stack pointer is modified, but the system stack pointer is not. Note that the RETA and CFCT registers are not used. This mode is slower than the Dual 16-Bit Stack With Fast Return mode; however, it gives the added advantage of simplifying stack unwinding because only memory is involved, no registers are used.

2.3 32-Bit Stack With Slow Return

The data stack and the system stack act as a single 32-bit stack; when the data stack is accessed, SP and SSP are modified by the same increment. RETA and CFCT are not used. Note that if the SP is modified directly, the SSP is not automatically updated. The user must keep the two pointers aligned. This configuration is the default stack mode and supports C54x™ compatibility.

3 Stack Mode Initialization

The stack mode is set during power up or reset; therefore, the bits in the reset vector determine what mode of operation the stack performs. If the Texas Instruments (TI) bootloader is used, the ROM code, by default, sets it to C54x compatibility, which means the 32-Bit Stack With Slow Return mode is used.

C54x is a trademark of Texas Instruments.

The stack mode can be set in two different ways:

- booting from ROM
- booting from a new vector table with the mode set in the Reset Vector location initiated with a software reset

3.1 Booting From External ROM

If the system boots from external ROM, then the 32-bit reset vector location has to have one of the following formats:

- For Dual 16-Bit Stack With Fast Return

  Reset Vector = xx 00 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
  - Bits 29 and 28 must be set to zero
  - Bits [23 – 0] contain the 24-bit start address of the reset interrupt service routine
  - Bits [30,31,27 – 24] are don't cares
- For Dual 16-Bit Stack With Slow Return

  Reset Vector = xx 01 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
  - Bit 29 must be set to 0b and bit 28 must be set to 1b
  - Bits [23 – 0] contain the 24-bit start address of the reset interrupt service routine
  - Bits [30,31,27 – 24] are don't cares
- For a 32-Bit Stack With Slow Return

  Reset Vector = xx 10 xxxx xxxx xxxx xxxx xxxx xxxx xxxx
  - Bit 29 must be set to 1b and bit 28 must be set to 0b
  - Bits [23 – 0] contain the 24-bit start address of the reset interrupt service routine
  - Bits [30,31,27 – 24] are don't cares

Please note that the reset vector address refers to the location that the CPU automatically goes to when a software or hardware reset occurs. The reset interrupt service routine is the least 24 bits of the reset vector and is where the CPU begins execution of the reset interrupt service routine.

3.2 Using a Soft Reset With a Vector Table to Initialize the Stack

A second way to initialize the stack mode is to set it in the in the reset vector, in a vectors.asm file.

```
;Vectors.asm
      .sect "vectors"
rsv:  .ivec ISR_start_location, STACK_MODE
```

where STACK_MODE is

```
C54X_STK:   32-bit stack (default)
USE_RETA:   16-bit stack using RETA
NO_RETA:    16-bit stack with out RETA
```

Figure 2. Stack Mode Initialization