Application Report
SPRA668 - May 2000

Interfacing TLC320AC01 to the TMS320C54x Serial Port

C5000 Applications Team
Digital Signal Processing Solutions

ABSTRACT

Most DSP systems transfer data through peripherals. These peripherals include parallel and serial ports. This application report describes how the serial ports are initialized and how the TLC320AC01 (’AC01) analog interface circuit (AIC) interfaces to the TMS320C54x™ serial port. This application report also describes the various issues involved such as stack, context switching, interrupt priorities, and different addressing modes for collecting the samples during the interrupt processing.

Contents

List of Examples

1 Context Switching

Before you execute a routine, you must save its context and restore the context after the routine has finished. This procedure is called context switching, and involves pushing the PC onto the stack. Context switching is useful for subroutine calls, especially when making extensive use of the auxiliary registers, accumulators, and other memory-mapped registers.

Due to system and CPU requirements, the order of saving and restoring can vary. Some repeat instructions, such as RPTB, are interruptible. To nest repeat block instructions, you must ensure that the block-repeat counter (BRC), block-repeat start address (RSA), and block-repeat end address (REA) registers are saved and restored.

You must also ensure that the block-repeat active flag (BRAF) is properly set. Since the block-repeat flag can be deactivated by clearing the BRAF bit of the ST1 register, the order in which you push the block-repeat counter and ST1 is important. If the BRC register is pushed onto the stack prior to ST1, any PC discontinuity in RPTB can give a wrong result, since BRAF is cleared in ST1. Thus, you must restore BRC before restoring the ST1 register.

A context save complements the restored contents. To ensure the integrity of the code, determine what contents must be restored so that no sequencing is lost.

Example 1. Context Save and Restore for TMS320C54x

```
   .title   “CONTEXT SAVE/RESTORE on SUBROUTINE or INTERRUPT
CONTEXT_RESTORE   .macro
   POPM  PMST     ;Restore PMST register
   POPM  RSA      ;Restore block repeat start address
   POPM  REA      ;Restore block repeat end address
   POPM  BRC      ;Restore block repeat counter
   POPM  IMR      ;Restore interrupt mask register
   POPM  BK    ;Restore circular size register
   POPM  ST1      ;Restore ST1
   POPM  ST0      ;Restore ST0
   POPM  AR0      ;Restore AR0
   POPM  AR1      ;Restore AR1
   POPM  AR2      ;Restore AR2
   POPM  AR3      ;Restore AR3
   POPM  AR4      ;Restore AR4
   POPM  AR5      ;Restore AR5
   POPM  AR6      ;Restore AR6
   POPM  AR7      ;Restore AR7
   POPM  T     ;Restore temporary register
   POPM  TRN      ;Restore transition register
   POPM  BL    ;Restore lower 16 bits of accB
   POPM  BH    ;Restore upper 16 bits of accB
   POPM  BG    ;Restore 8 guard bits of accB
   POPM  AL    ;Restore lower 16 bits of accA
   POPM  AH    ;Restore upper 16 bits of accA
   POPM  AG    ;Restore 8 guard bits of accA
   .endm
CONTEXT_SAVE   .macro
   PSHM  AG    ;Save 8 guard bits of accA
   PSHM  AH    ;Save upper 16 bits of accA
   PSHM  AL    ;Save lower 16 bits of accA
   PSHM  BG    ;Save 8 guard bits of accB
   PSHM  BH    ;Save upper 16 bits of accB
   PSHM  BL    ;Save lower 16 bits of accB
   PSHM  TRN      ;Save transition register
   PSHM  T     ;Save temporary register
   PSHM  AR7      ;Save AR7
   PSHM  AR6      ;Save AR6
   PSHM  AR5      ;Save AR5
   PSHM  AR4      ;Save AR4
   PSHM  AR3      ;Save AR3
   PSHM  AR2      ;Save AR2
   PSHM  AR1      ;Save AR1
   PSHM  AR0      ;Save AR0
   PSHM  ST0      ;Save ST0
   PSHM  ST1      ;Save ST1
   PSHM  BK    ;Save circular size register
   PSHM  IMR      ;Save interrupt mask register
   PSHM  BRC      ;Save block repeat counter
   PSHM  REA      ;Save block repeat end address
   PSHM  RSA      ;Save block repeat start address
   PSHM  PMST     ;Save PMST register
   .endm
```

2 Interrupt Handling

The ’54x CPU supports 16 user-maskable interrupts. The vectors for interrupts not used by a ’54x device can function as software interrupts, using the INTR and TRAP instructions. TRAP and INTR allow you to execute any of the 32 available ISRs. You can define other locations in the interrupt vector table. The INTR instruction sets the INTM bit to 1, clears the corresponding interrupt flag to 0, and makes the $\overline{\text{IACK}}$ signal active, but the TRAP instruction does not. INTR and TRAP are nonmaskable interrupts.

When a maskable interrupt occurs, the corresponding flag is set to 1 in the interrupt flag register (IFR). Interrupt processing begins if the corresponding bit in IMR register is set to 1 and the INTM bit in the ST1 register is cleared. The IFR register can be read and action taken if an interrupt occurs. This is true even when the interrupt is disabled. This is useful when not using an interrupt-driven interface, such as in a subroutine call when INT1 has not occurred.

When interrupt processing begins, the PC is pushed onto the stack and the interrupt vector is loaded into the PC. Interrupts are then disabled by setting INTM = 1. The program continues from the address loaded in the PC. Since all interrupts are disabled, the program can be processed without any interruptions, unless the ISR reenables them. Except for very simple ISRs, it is important to save the processor context during execution of the routine.

During the time the ’AC01 is reset, the DSP initializes the serial port and sets up the interrupt. To set up the interrupts, it performs the following operations:

- Enables unmasked interrupts by clearing the interrupt mode bit (INTM)
- Clears prior receive interrupts by writing the current contents of the appropriate receive interrupt flag in the IFR back to the IFR
- Enables receive interrupts by setting the appropriate receive interrupt flag in the interrupt mask register (IMR)

The initialization of the IMR and IFR registers and the INTM bit is included in the serial port and the ’AC01 initialization.

Example 2 processes the receive interrupt 1 service routine. The routine collects 256 samples in the first buffer and changes the address to the second buffer for the next 256 samples while processing the first buffer.

Example 2. Receive Interrupt Service Routine

```
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include ”INTERRPT.INC”
   .include ”main.inc”
RCV_INT1_DP    .usect      ”rcv_vars”,0
d_index_count     .usect      ”rcv_vars”,1
d_rcv_in_ptr      .usect      ”rcv_vars”,1            ; save/restore input bffr ptr
d_xmt_out_ptr     .usect      ”rcv_vars”,1            ; save/restore output bffr
ptr
d_frame_flag      .usect      ”rcv_vars”,1
input_data     .usect      ”inpt_buf”,K_FRAME_SIZE*2 ; input data array
output_data    .usect      ”outdata”,K_FRAME_SIZE*2  ; output data array
   .def        receive_int1
   .def        d_frame_flag
   .def        RCV_INT1_DP
   .def        input_data,output_data
   .def        d_xmt_out_ptr
   .def        d_rcv_in_ptr
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;  Functional Description
;   This routine services receive interrupt1. Accumulator A, AR2 and AR3
;  are pushed onto the stack since AR2 and AR3 are used in other applications.
;  A 512 buffer size for both input and output.
;  After every 256 collection of input samples a flag is set to process the
;  data. No circular buffering scheme is used here.
;  After collecting 256 samples in the 1st bffr, then the second buffer
;  address is loaded and collect data in the second buffer while processing
;  the first buffer and vice versa.
;––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
   .asg        AR2,GETFRM_IN_P                   ; get frame input data pointer
   .asg        AR3,GETFRM_OUT_P                  ; get frame output data pointer
   .asg        AR2,SAVE_RSTORE_AR2
   .asg        AR3,SAVE_RSTORE_AR3
   .sect       ”main_prg”
receive_int1:
      PSHM     AL
      PSHM     AH
      PSHM     AG
      PSHM     BL
      PSHM     BH
      PSHM     BG
; AR2, AR3 are used in other routines, they need to be saved and restored
; since receive interrupt uses AR2 and AR3 as pointers
      PSHM     SAVE_RSTORE_AR2                   ; Since AR2 and AR3 are used
      PSHM     SAVE_RSTORE_AR3                   ; in other routines, they need
      PSHM     BRC
      LD       #RCV_INT1_DP,DP                   ; init. DP
      MVDK     d_rcv_in_ptr,GETFRM_IN_P          ; restore input bffr ptr
      MVDK     d_xmt_out_ptr,GETFRM_OUT_P        ; restore output bffr ptr
      ADDM     #1,d_index_count                  ; increment the index count
      LD       #K_FRAME_SIZE,A
      SUB         d_index_count, A
      BC       get_samples,AGT                   ;check for a frame of samples

frame_flag_set
      ADDM  #1,d_int_count
      ST    #K_FRAME_FLAG,d_frame_flag        ; set frame flag
      ST    #0,d_index_count                  ; reset the counter
      LD    #input_data+K_FRAME_SIZE,A        ; second input bffr starting addr
      LD    #output_data+K_FRAME_SIZE,B       ; second output bffr starting addr
      BITF  d_int_count,2                     ; check for 1st/2nd  bffr
      BC    reset_buffer,NTC
```

Example 2. Receive Interrupt Service Routine (Continued)

```
      SUB      #K_FRAME_SIZE,A                  ; 1st input address
      SUB      #K_FRAME_SIZE,B                  ; 1st output address
      ST    #K_0,d_int_count
reset_buffer
      STLM  A,GETFRM_IN_P                    ; input buffer address
      STLM  B,GETFRM_OUT_P                ; output buffer address
get_samples
      LDM      DRR1,A                           ; load the input sample
      STL      A,*GETFRM_IN_P+                  ; write to buffer
      LD    *GETFRM_OUT_P+,A                 ; if not true, then the filtered
      AND      #0fffch,A                        ; signal is send as output
      STLM  A,DXR1                           ; write to DXR1
      MVKD  GETFRM_IN_P,d_rcv_in_ptr         ; save input buffer ptr
      MVKD  GETFRM_OUT_P,d_xmt_out_ptr       ; save out bffr ptr
      POPM  BRC
      POPM  SAVE_RSTORE_AR3                  ; restore AR3
      POPM  SAVE_RSTORE_AR2                  ; restore AR2
      POPM  BG
      POPM  BH
      POPM  BL
      POPM  AG
      POPM  AH
      POPM  AL
      POPM  ST1
      POPM  ST0
      RETE                                ; return and enable interrupts
      .end
```

3 Interrupt Priority

Interrupt prioritization allows interrupts that occur simultaneously to be serviced in a predefined order. For instance, infrequent but lengthy ISRs can be interrupted frequently. In Example 3, the ISR for the INT1 bit includes context save and restore macros. When the routine has finished processing, the IMR is restored to its original state. Notice that the RETE instruction not only pops the next program counter address from the stack, but also clears the INTM bit to 0. This enables all interrupts that have their IMR bit set.

Example 3. Interrupt Service Routine (ISR)

```
   .title   “Interrupt Service Routine”
   .mmregs
int1:
   CONTEXT_STORE  ; push the contents of accumulators and registers on stack
   STM   #K_INT0,IMR ; Unmask only INT0~
   RSBX  INTM  ; Enable all Interrupts
;
; Main Processing for Receive Interrupt 1
.
.
.
   SSBX  INTM  ; Disable all interrupts
   CONTEXT_RESTORE   ; pop accumulators and registers
   RETE     ; return and enable interrupts
   .end
```

There is a potential conflict between the INTM bit disable and context restore. If an interrupt 0 (INT0) occurs during context restore, the macro CONTEXT_RESTORE is executed before servicing INT0. This can trigger an INT0. If INTM is cleared during the context restore, it branches to the INT0 service routine. If you reenable the interrupts when INTM returns from INT0, a conflict occurs, because INTM is set to 0 and its original contents are lost. To preserve the contents of the INTM bit, do not enable the interrupts when INTM returns from the INT0 service routine. During interrupt priorities, preserve the INTM and IMR bits for the system requirements.

4 Circular Addressing

Circular addressing is an important feature of the ’54x instruction set. Algorithms for convolution, correlation, and FIR filters can use circular buffers in memory. In these algorithms, the circular buffers implement a sliding window that contains the most recent data. As new data comes in, it overwrites the oldest data. The size, the bottom address, and the top address of the circular buffer are specified by the block size register (BK) and a user-selected auxiliary register (ARn). A circular buffer size of R must start on a K-bit boundary (that is, the K LSBs of the starting address of the circular buffer must be 0), where K is the smallest integer that satisfies $2^K > R$.

Circular addressing can be used for different functions of an application. For example, it can be used for collecting the input samples in a block. It can also be used in processing samples in blocks and data in the output buffer. In Example 4, a frame of 256 samples is collected from the serial port to process the data using the circular addressing mode. The output from the processed block is sent to the D/A converter through the serial port register using circular buffers. A ping-pong buffering scheme is used. While processing the first buffer, samples are collected in the second buffer, and vice versa. The real-time operation of the system is not disturbed and no data samples are lost.

Example 4. Circular Addressing Mode

```
; TEXAS INSTRUMENTS INCORPORATED
   .mmregs
   .include    ”INTERRPT.INC”
   .include    ”main.inc”
RCV_INT1_DP    .usect   ”rcv_vars”,0
d_index_count     .usect   ”rcv_vars”,1
d_rcv_in_ptr      .usect   ”rcv_vars”,1               ; save/restore input bffr ptr
d_xmt_out_ptr     .usect   ”rcv_vars”,1               ; save/restore output bffr ptr
d_frame_flag      .usect   ”rcv_vars”,1
input_data     .usect   ”inpt_buf”,K_FRAME_SIZE*2  ; input data array
output_data    .usect   ”outdata”,K_FRAME_SIZE*2   ; output data array
   .def        receive_int1
   .def        d_frame_flag
   .def        RCV_INT1_DP
   .def        input_data,output_data
   .def        d_xmt_out_ptr
   .def        d_rcv_in_ptr
;-------------------------------------------------------------------------------------
;     Functional Description
;   This routine services receive interrupt1. Accumulator A, AR2 and AR3
;  are pushed onto the stack since AR2 and AR3 are used in other applications.
;  A 512 buffer size of both input and output uses circular addressing.
;  After every 256 collection of input samples a flag is set to process the
;  data. A PING/PONG buffering scheme is used such that upon processing

;  PING buffer, samples are collected in the PONG buffer and vice versa.
;---------------------------------------------------------------------------------
```

Example 4. Circular Addressing Mode (Continued)

```
   .asg  AR2,GETFRM_IN_P                     ; get frame input data pointer
   .asg  AR3,GETFRM_OUT_P                    ; get frame output data pointer
   .asg  AR2,SAVE_RSTORE_AR2
   .asg  AR3,SAVE_RSTORE_AR3
   .sect ”main_prg”
receive_int1:
   PSHM  AL
   PSHM  AH
   PSHM  AG
   PSHM  BL
   PSHM  BH
   PSHM  BG
; AR2, AR3 are used in other routines, they need to be saved and restored
; since receive interrupt uses AR2 and AR3 as pointers
   PSHM  SAVE_RSTORE_AR2                     ; Since AR2 and AR3 are used
   PSHM  SAVE_RSTORE_AR3                     ; in other routines, they need
   PSHM  BRC
   STM      #2*K_FRAME_SIZE,BK                  ; circular buffer size of in,out
                                          ; arrays
   LD    #RCV_INT1_DP,DP                     ; init. DP
   MVDK  d_rcv_in_ptr,GETFRM_IN_P            ; restore input circular bffr ptr
   MVDK  d_xmt_out_ptr,GETFRM_OUT_P          ; restore output circular bffr ptr
   ADDM  #1,d_index_count                    ; increment the index count
   LD    #K_FRAME_SIZE,A
   SUB      d_index_count, A
   BC    get_samples,AGT                     ;check for a frame of samples
frame_flag_set
   ST    #K_FRAME_FLAG,d_frame_flag          ; set frame flag
   ST    #0,d_index_count                    ; reset the counter
get_samples
   LDM      DRR1,A                              ; load the input sample
   STL      A,*GETFRM_IN_P+%                    ; write to buffer
   LD    *GETFRM_OUT_P+%,A                ; if not true, then the filtered
   AND      #0fffch,A                           ; signal is send as output
   STLM  A,DXR1                              ; write to DXR1
   MVKD  GETFRM_IN_P,d_rcv_in_ptr            ; save input circular buffer ptr
   MVKD  GETFRM_OUT_P,d_xmt_out_ptr          ; save out circular bffr ptr
   POPM  BRC
   POPM  SAVE_RSTORE_AR3                     ; restore AR3
   POPM  SAVE_RSTORE_AR2                     ; restore AR2
   POPM  BG
   POPM  BH
   POPM  BL
   POPM  AG
   POPM  AH
   POPM  AL
   POPM  ST1
   POPM  ST0
   RETE                                   ; return and enable interrupts
   .end
```