

DSP32C External Interrupts IREQ1,2

This note covers using and programming the DSP32C external interrupts IREQ1,2. While the external IREQ interrupts are discussed, the same techniques can be applied to other DSP32C interrupt sources such as the SIO serial port. Physically, the external interrupts IREQ1,2 are available either on the 32 bit parallel port header or their own separate header block depending on the specific SR DSP32C board you have. They provide a good way for external devices to get the attention of the DSP32C for asynchronous tasks like data transfer.

The operation of IREQ1,2 is fairly simple. As soon as a low level is detected on one of the IREQ1,2 pins the DSP32C branches to an interrupt service routine (isr). Before interrupts are enabled, an interrupt vector table must be set up to point to the interrupt service routines, or filled with two instruction "quick interrupts", and the CAU register r22 (ivtp) must point to this table. The code fragments below show the basic programming techniques, while the app note disk has two complete programs, `intrupt.dsp` and `quick.dsp`, that are ready to run. Also see pages 2-31 and 7-29 in the AT&T DSP32C Information manual for more information about interrupts.

The steps in setting up an interrupt are as follows:

After power up or a software reset, all DSP32C interrupts are disabled. To become functional, a particular interrupt must be explicitly enabled by setting a corresponding bit in the DSP32C PCW register. These bit assignments are best programmed with defined constants, where typical constants are listed in the following code fragment. Note that the PCW register also controls wait state settings and other processor features. Be careful to preserve the other bit settings when changing the interrupt enable status for a particular interrupt.

```
constant
    ENABLE_INTREQ1      = 0x8000,
    ENABLE_PDF          = 0x4000,
    ENABLE_PDE          = 0x2000,
    ENABLE_IBF          = 0x1000,
    ENABLE_OBE          = 0x0800,
    ENABLE_INTREQ2      = 0x0400;
```

To declare an interrupt vector table in DSPASM, you might at first want to allocate it as a static data array, but a better way is to define it as a routine. That way any branch or "quick interrupt" instructions that need to be programmed can be directly encoded. Furthermore, the name of the interrupt vector table routine is a symbolic constant that can be later used to set the value of r22 (ivtp). An example of an interrupt vector table is:

```
IVTP_TABLE {
    nops_off;
    goto isr_INTREQ1;          /* interrupt vector #1 */
```

```

    nop;
    goto isr_PDF;           /* interrupt vector #2 */
    nop;
    goto isr_PDE;           /* interrupt vector #3 */
    nop;
    goto isr_IBF;           /* interrupt vector #4 */
    nop;
    goto isr_OBE;           /* interrupt vector #5 */
    nop;
    goto isr_INTREQ2;       /* interrupt vector #6 */
    nop;
    nops_on;
}

```

The interrupt vector table layout is covered on page 2-23 in the AT&T DSP32C Information manual, and is comprised of entries with two instruction per entry. Normally, the table entries are a branch to an interrupt service routine followed by a nop. However, for two instruction "quick interrupts" where the isr is a single instruction, they can also be the instruction followed by an ireturn. Note that DSPASM automatic nop generation has been turned off so every instruction in the table can be explicitly programmed.

Besides setting up the interrupt vector table, you'll need to program interrupt service routines. The isr routines should execute any interrupt tasks, and then terminate with an ireturn to resume mainline execution. The ireturn instruction not only returns to the mainline code, it also reenables further interrupts on that particular signal. Until an ireturn is executed, interrupts on that signal remain disabled. The interrupt state is indicated in hardware by the signals on the IACK1,2 pins.

A DSPASM main program and an interrupt service routine code fragment for IREQ1 might be:

```

main {
    register

        value    = r1,
        ivtp     = r22; /* DSP32C dedicated ivtp pointer */

    /* Program the PCW register for zero wait states. */
    value = 0; pcw = value;

    /* Point ivtp to the interrupt vector table. */
    ivtp = IVTP_TABLE;

    /* Enable INTREQ1 and wait for interrupts. */
    value = pcw;

        value |= ENABLE_INTREQ1; pcw = value;

    wait:

        /* Put any foreground mainline code in here. */

        goto wait;
}

```

```

isr_INTREQ1 {
    register c = r2;

    /* Increment the count. */
    c = *count; c += 1; *count = c;

    /* Return to the main line code. */
    ireturn;
}

long count = 0;

```

This program increments the count variable every time an interrupt occurs. To run it, download and start execution by hand in DSPMON, and then bring the IREQ1 pin low. The count variable will start incrementing and continue incrementing only while the IREQ1 pin is low.

Note that while the DSP32C automatically saves and restores the state of the floating point DAU aN registers across an interrupt, CAU rN register values are not. Any changes made to CAU registers in an interrupt service routine will remain after returning to the mainline code. This is good for passing arguments in registers, but if you need to preserve CAU registers across calls, save and restore on entry and exit to the isr. Corrupting mainline CAU registers with an isr is an easy error to make and can be hard to debug! Particularly if you are using named registers.

The code on the app note disk has the above code fragments as a complete program, `intrupt.dsp`. There is also a "quick interrupt" example in `quick.dsp`.