Interrupt support for the DSP Global UART board

1 Introduction

Several projects this semester (Winter 2004) are going to use a RS-232 serial port daughter board as an add-on to the class's C5510 DSKs. The boards were purchased from DSP Global. The manual for the boards and the data sheet for the UART used on these boards are available on the class handouts web page. See the instructor for the software supplied with the boards. The purpose of this note is show how one might program these boards starting from scratch.

The course handout web page contains the PDF files for the manual supplied by DSP Global for their boards along with the data sheet for the Philips UART used on these boards. Also present are the text files for much of the code generated for this note.

As always, use at your own risk.

1.1 RS-232 used for asynchronous operation

- Transmitter and receiver clocks not synchronized.
- Derives from mechanical Teletype 20 ma current loop.
- RS-232 adopted in 1960 Electronics Industry Association.
- Current version is RS-232C adopted in 1969, RS-232D in 1987.
- Supports full duplex.
- Data Terminal Equipment — Data Communication Equipment based.
- Originally used a 25 pin connector. Now 9 pins is most common.
- Unbalanced lines.
- (-15 ,-5 volts) mark, (+5, +15 volts) space.
- Minimal interface uses three wires:

  transmitted data,
  received data,
  ground.

- Slew rate is limited to combat EMI.
- Electrical and/or logical handshake protocol.

1.2 The Asynchronous Serial Frame

- one start bit — to synchronize clocks
- a number of data bits
- optional parity bit (odd or even)

DRAFT

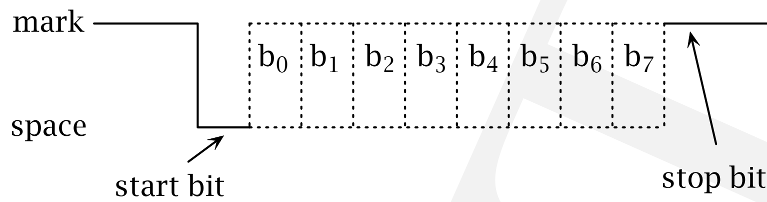

Figure 1: RS-232 serial data organization. The mark voltage level is the negative voltage. The space voltage level is the positive voltage.

- one or one and a half or two stop bits

2 RS-232 board location in memory address space

The RS-232 boards are implemented as a memory mapped peripheral. The are accessed using "normal" memory read and write operations. When accessing them from C the `far_peek` and `far_poke` functions give ready access.

The C5510 DSK assigns memory word addresses 0x400000 through 0x7FFFFF for use by add-on daughter boards. The GEL script used to initialize the DSK initializes the C5510 external memory interface in a manner compatible with the DSP Global boards. A description of the EMIF settings will be added at a later date.

The EECS 452 RS-232 daughter boards provide two serial channels. The base memory addresses used by these channels are

channel 1	0x500000
channel 2	0x500200

The DSP Global boards use the Phillips SC16C2550 dual UART chip. There are 12 8-bit registers associated with each UART.

DRAFT

register	register name	offset from base
RBR	read buffer register	0x00
THR	transmit holding register	0x00
DLL	divisor low byte	0x00
DLM	divisor high byte	0x02
IER	interrupt enable register	0x02
ISR	interrupt status register	0x04
FCR	FIFO control register	0x04
LCR	line control register	0x06
MCR	modem control register	0x08
LSR	line status register	0x0A
MSR	modem status register	0x0C
SPR	scratch pad register	0x0E

The DSP Global board uses a crystal controlled oscillator to determine the baud rate it uses. The contents of the divisor register are used to divide the oscillator frequency down to the desired baud rate.

Baud rate = 230400/(contents of DLM:DLL).

The values required for some common baud rates are:

rate	DLM	DLL
115200	0x00	0x02
57600	0x00	0x04
38400	0x00	0x06
19200	0x00	0x0C
14400	0x00	0x10
9600	0x00	0x18

Channel 1 on the DSP Global board is configured as a data terminal equipment (DTE, like as done on a PC) and uses a male 9-pin connector. Channel 2 is configured as a data communication equipment (DCE) and uses a female 9-pin connector. Channel 2 is the channel to use a standard RS-232 cable to connect directly to a PC. A special adapter is needed to successfully connect channel 1 to a PC.

Looking at the channel 1 DB9M connector from the front the pins are

```
1 2 3 4 5
 6 7 8 9
```


DRAFT

pin	function	direction
1	DCD	in to UART
2	RXD	in to UART
3	TXD	out from UART
4	DTR	out from UART
5	ground	
6	DSR	in to UART
7	RTS	out from UART
8	CTS	in to UART
9	RI	in to UART

Looking at the channel 2 DB9F connector from the front the pins are

```
5 4 3 2 1
 9 8 7 6
```

pin	function	direction
1	DTR	out from UART
2	TXD	out from UART
3	RXD	in to UART
4	DSR	in to UART
5	ground	
6	DTR	out from UART
7	CTS	in to UART
8	RTS	out from UART
9	—	not used

The logic levels use by the waveforms on the DB9 connectors are nominally ±6 volts. **These signals should not be connected directly to logic gates!**

Using the DTR waveform for on the fly timing

The state of the DTR line on the channel 1 DB9M connector (pin 4) can be used as a timing and/or event indicator. This is easily accomplished in assembler and in C using `far_peek`. The use of this waveform allows timing measurements to be made without significantly affecting program operation (as compared to the profiler). The DTR line is controlled by bit 0 in the channel modem control register (MCR).

To make use of this waveform in C use the following variable definition:

```
MCRch0 = 0x500008;
```


DRAFT

The following code snip illustrates the use of this waveform to time the overhead associated with `far_poke`.

```
        far_poke(MCR, 0x00); // put into its reset state
        MCRch0 = 0x500008;
    while (FOREVER) {
        far_poke(MCRch0, 0x01);    // set DTR on channel 0
        far_poke(MCRch0, 0x00);    // reset the DTR on channel 0
        far_poke(MCRch0, 0x00);    // reset the DTR on channel 0
        far_poke(MCRch0, 0x00);    // reset the DTR on channel 0
    }
```

Cables are available that have a DB9F connector on one end and a BNC connector on the other. The BNC center pin is connected to pin 4 of the DB9 connector. The output voltage for a space is approximately +6 volts and for a mark is approximately -6 volts.

Typically the rise and fall times of a RS-232 waveform is slew rate limited by the driver. This limits the usefulness when timing short events. The waveform produced by the above example doesn't even make it into positive territory. However it still can be useful. See Figure 2.

Note that this example also affects all other bits in the modem control register. In most cases this will not cause any problems. A more general procedure, which will allow other pins on the DB9, to be used independently is read the MCR modify the specific bit to be changed and rewrite the result back to the MCR.

The bandwidth of this waveform is somewhat limited making its use for timing very short events less useful. The `xf` flag can be use in a similar manner but the connection to the DSK to make use of this flag is ***much*** more fragile.

DRAFT

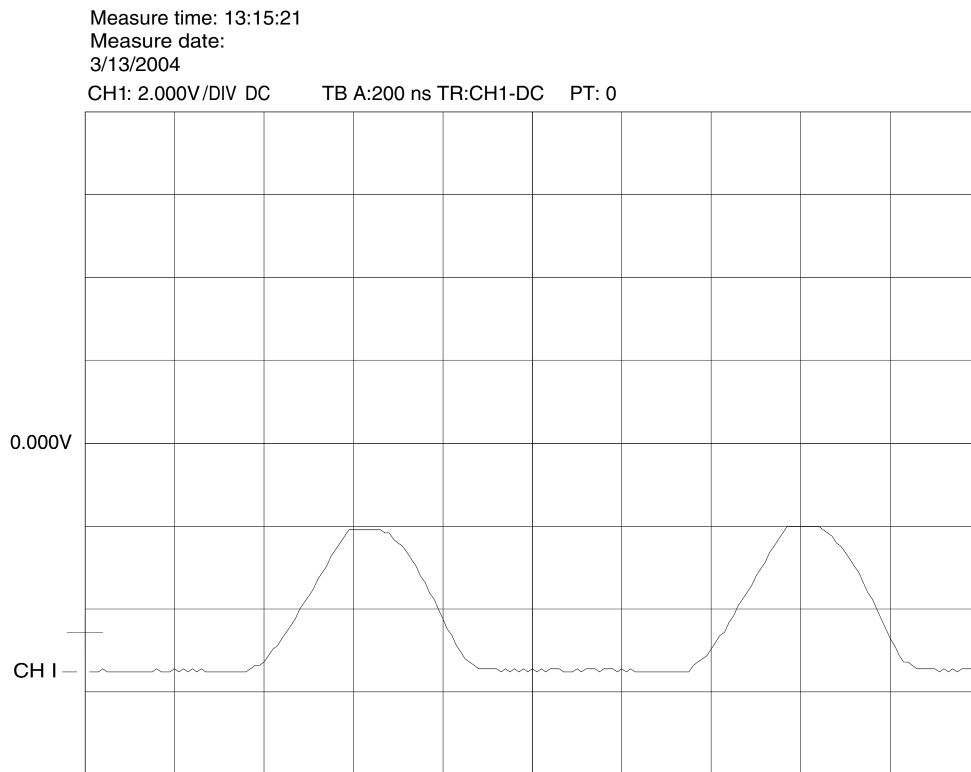

Figure 2: DTR waveform generated with 1 far_poke using a 1 bit followed by 3 far_pokes using a 0 bit. Scale: 2 volts per vertical division and 200 ns per horizontal division.

3 Initial test C test program

This was a quick-and-dirty test code intended to get some experience with the DSP Global board. Future work should use an include file with a method for distinguishing between channels.

```
/* File name: UARTtest.c

    11Mar2004 .. initial version .. KM
    28Mar2004 .. a fix and added interrupt test .. KM

*/

#include <stdio.h>
#include <math.h>
#include <extaddr.h>

#define FOREVER 1

#define UART 0x500200
#define RBR (UART+0x00)
#define THR (UART+0x00)
#define DLL (UART+0x00)
#define DLM (UART+0x02)
#define IER (UART+0x02)
#define ISR (UART+0x04)
#define FCR (UART+0x04)
#define LCR (UART+0x06)
#define MCR (UART+0x08)
#define LSR (UART+0x0A)
#define MSR (UART+0x0C)
#define SPR (UART+0x0E)

extern unsigned volatile int U1RxChar;
extern unsigned volatile int U1RxFlag;
unsigned int U1IStatus;

int test = 6;

/* baud rate is 230,400/divisor
   for 38400 baud use a value of 6
*/

int RateLow = 6, RateHigh = 0;

void simple_print(int CHar)
{
    far_poke(THR, CHar);
    while ((far_peek(LSR)&0x20)==0);
}
```


DRAFT

```
void main(void)
{
    unsigned counter;
    int temp, temp2;
    unsigned long MCRch0;

    // set up baud rate

    far_poke(LCR, 0x80); // access baud rate registers
    far_poke(DLM, RateHigh);
    far_poke(DLL, RateLow);
    far_poke(LCR, 0x07); // use 8 data and 2 stop bits

    if (test == 1) {    // can we write/read scratch register?
        far_poke(SPR, 0x53);
        temp = far_peek(SPR);
        printf("%4X\n", temp);
        return;
    }

    if (test == 2) {    // set up baud rate
        far_poke(LCR, 0x80); // access baud rate registers
        far_poke(DLM, RateHigh);
        far_poke(DLL, RateLow);
        far_poke(LCR, 0x07); // use 8 data and 2 stop bits
        temp  = far_peek(DLM);
        temp2 = far_peek(DLL);
        printf("%2X %2X\n", temp, temp2);
        return;
    }

    if (test == 3) {  // loop printing 256 character set
        far_poke(LCR, 0x80); // access baud rate registers
        far_poke(DLM, RateHigh);
        far_poke(DLL, RateLow);
        far_poke(LCR, 0x07); // use 8 data and 2 stop bits

        counter = 0;
        while (FOREVER) {
            far_poke(THR, counter++);
            while ((far_peek(LSR)&0x20)==0);
        }
    }

    if (test == 4) { // loop accepting input and echoing it
        far_poke(LCR, 0x80); // access baud rate registers
        far_poke(DLM, RateHigh);
        far_poke(DLL, RateLow);
        far_poke(LCR, 0x07); // now access xmt and rcv regs
```


DRAFT

```
        while (FOREVER) {
            while ((far_peek(LSR)&0x01)==0); // wait for character
            far_poke(THR, far_peek(RBR));     // read it and echo it
        }
    }

    if (test == 5) { // Test use of DTR on DB9M for timing
            far_poke(MCR, 0x00); // put into its reset state
            MCRch0 = 0x500008;
        while (FOREVER) {
            far_poke(MCRch0, 0x01);   // set DTR on channel 0
            far_poke(MCRch0, 0x00);   // reset the DTR on channel 0
            far_poke(MCRch0, 0x00);   // reset the DTR on channel 0
            far_poke(MCRch0, 0x00);   // reset the DTR on channel 0
        }
    }
    if (test == 6) { // Tests for developing interrupt handler
        UART1setup();
        _enable_interrupts();
        while(FOREVER) {
            while (U1RxFlag == 0);
            far_poke(THR, U1RxChar);
            U1RxFlag = 0;
        }
    }
    if (test == 7) { // Demonstrates how to print a string
        char *str;

        while(FOREVER) {
            str = "1,2,3,4,5,6,\r\n9,8,7,6,5,4\r\n";
            while (*str != 0) simple_print(*str++);
        }
    }
}
```

4 Test C UART interrupt handler

4.1 Setting up the UART interrupt vector

The `init_UART` function is used to patch the UART interrupt handler into the interrupt vector. The approach that I've taken is to not create a special interrupt vector for each project but rather to use an initialization function to link into the existing interrupt vector. More portable and less work.

First we need to decide on which interrupt to use with the DSP Global board. The only other daughter board we need to be concerned with presently is the TLV1571 EVM. Checking its schematic, the EVM uses INT0# on pin 53 of the DSK peripheral connector. This was for the C5402 DSK. A check of the C5510 DSK schematic shows that pin 53 is actually INT1#.

Presently we will only be generating support code for UART channel 2 (counting from 1). According to the DSP Global documentation we can use C5510 interrupt 0 by adding a jumper

```
E4 (Channel 2, active low) to E15 (INT0#).
```

Next we need to figure out how to patch the interrupt handler addresses into the interrupt vector.

The following information was located in the C5510 data manual.

The following registers are used with interrupts. The addresses are word addresses.

IER0	0x0000	interrupt mask register 0
IFR0	0x0001	interrupt flag register 0
IER1	0x0045	interrupt mask register 1
IFR0	0x0045	interrupt flag register 1
IVPD	0x0049	interrupt vector pointer, DSP
IVPH	0x0050	interrupt vector pointer, host

The INT0# interrupt vector entry is 8 words offset from the start of the interrupt vector.

Bit 2 of IER0 and IFR0 is used by INT0#.

The interrupt vector pointers point to the 256 byte (128 word) page that the interrupt vector is located. To access the first entry in the interrupt vector we will have to make sure that the first entry in our interrupt vector has the globally accessible name: `_resetv`. This requires minor additions to the intvec.asm code.

Next we will run a short test to make sure we can access the address of `_resetv`.

```
#include <stdio.h>

void resetv();

void UART1setup(void)
{
    long resetloc;

    resetloc = (long)resetv;
    printf("reset  %lX\n", resetloc);
    while(1);
}
```


DRAFT

The value printed out was 0x100 as it should have been. This is a byte address.

We need to set the interrupt vectors to this value shifted right by 8 bits.

The test code now becomes

```
#define IVPD ((unsigned long)0x49)
#define IVPH ((unsigned long)0x50)

void far_poke(unsigned long, unsigned);
void resetv();

void UART1setup(void)
{
long resetloc;

    resetloc = (long)resetv;
    far_poke(IVPD, (unsigned)(resetloc>>8));
    far_poke(IVPH, (unsigned)(resetloc>>8));
    while(1);
}
```

A quick run using CCS to check the contents of the interrupt vectors shows that all is as expected.

Next we create dummy interrupt handler.

```
interrupt void UART1int(void)
{
    while(1);
}
```

and place its address into the interrupt vector. The text now becomes

```
#define IVPD ((unsigned long)0x49)
#define IVPH ((unsigned long)0x50)
#define INT0 0x0008

void far_poke(unsigned long, unsigned);
interrupt void UART1int(void);
void resetv();

void UART1setup(void)
{
    unsigned long resetloc;

    resetloc = (long)resetv;
    far_poke(IVPD, (unsigned)(resetloc>>8));
    far_poke(IVPH, (unsigned)(resetloc>>8));
    far_poke((resetloc>>1)+INT0, (unsigned)((unsigned long)UART1int>>16));
    far_poke((resetloc>>1)+INT0+1, (unsigned)((unsigned long)UART1int));
```


DRAFT

```
            while(1);
        }

interrupt void UART1int(void)
{
    while(1);
}
```

We had to be careful to keep track of when using word addresses and when using byte addresses. A quick run using the program memory window shows that we have successfully modified the interrupt vector.

All that remains for the moment is to enable the INT0# interrupt and clear the associated flag. Additional code to setup the interrupt handler and any associate buffer support will be added but first we need to write the interrupt handler so we know what needs to be done.

```
        #define IER0 ((unsigned long)0x00)
        #define IFR0 ((unsigned long)0x01)
        #define IVPD ((unsigned long)0x49)
        #define IVPH ((unsigned long)0x50)
        #define INT0 0x0008
        #define INT0_BIT 0x0004

        unsigned far_peek(unsigned long);
        void far_poke(unsigned long, unsigned);
        interrupt void UART1int(void);
        void resetv();

        void UART1setup(void)
        {
            unsigned long resetloc;

            resetloc = (long)resetv;
            far_poke(IVPD, (unsigned)(resetloc>>8));
            far_poke(IVPH, (unsigned)(resetloc>>8));
            far_poke((resetloc>>1)+INT0, (unsigned)((unsigned long)UART1int>>16));
            far_poke((resetloc>>1)+INT0+1, (unsigned)((unsigned long)UART1int));
            far_poke(IER0, far_peek(IER0)|INT0_BIT);
            far_poke(IFR0, INT0_BIT);
            while(1);
        }

interrupt void UART1int(void)
{
    while(1);
}
```


DRAFT

4.2 The UART interrupt handler

We used interrupts in lab exercises 6 and 7. These were written in assembler and it was clear what they were doing. Basically an interrupt handler must preserve the machine state across its execution. With the exception on well defined changes in variable values the interrupt handler must, upon its exit, restore the state of the processor. An equally important need is to be immune to the state of the machine when the interrupt occurred.

A C interrupt handler must meet these requirements. However, the code produced by the compiler appears to be undocumented. We need to make a test.

4.2.1 First test handler

Our test interrupt handler becomes:

```
unsigned volatile U1RxChar;
unsigned volatile U1TxChar;

interrupt void UART1int(void)
{
    U1RxChar = 1;
    return;
}
```

The assembly language output produced by the C compiler is given in Figure 3

Now we need to determine that is being done. There are no doubt lessons to be learned that can be used with the lab 6 and 7 interrupt handlers. Unfortunately there is no guarantee that the code shown in Figure 3 won't change depending on what is done in the handler.

The C5510 automatically stacks and unstacks status registers 0, 1 and 2 (not in that order) so we don't have to worry this ourselves.

- The full 23-bit auxiliary register 1 is saved.
- Status register 3 is saved.
- The `mdp` register is stacked. This register is not listed in the C5510 reference guide. A check of the C5510 data manual shows there is a memory location reserved for a `mdp05` register and a location reserved for a `mdp06` register. A quick search through manual indexes for `mpd` was not successful in locating information.

  Might be this be another name for the `dp` (also called the "memory data page start address"?

DRAFT

```
;*******************************************************************************
;* INTERRUPT NAME: _UART1int                                                   *
;*                                                                             *
;*   Function Uses Regs : AR1,XAR1,SP,ST1,ST2,ST3,MDP,M40,SATA,SATD,RDM,FRCT,  *
;*                        SMUL                                                 *
;*   Save On Entry Regs : AR1,ST3,MDP                                          *
;*******************************************************************************
_UART1int:
    .line   2
        PSHBOTH XAR1
        PSH mmap(ST3_55)
        PSH mmap(MDP)
        AMAR *SP(#0), XAR1
        AND #0xfffe, mmap(SP)
        PSH AR1
        AND #0xf91f, mmap(ST1_55)
        OR #0x4100, mmap(ST1_55)
        AND #0xfa00, mmap(ST2_55)
        OR #0x8000, mmap(ST2_55)
    .ref ___bss__
        MOV #((___bss__ >> 16) & 0x7f), MDP
        AMOV #___bss__, XAR1
        AADD #-1, SP
    .line   3
        MOV #1, *abs16(#_U1RxChar) ; |41|
    .line   4
    .line   5
        AADD #1, SP
        POP mmap(SP)
        POP mmap(MDP)
        POP mmap(ST3_55)
        POPBOTH XAR1
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        RETI
                                        ; return occurs
    .endfunc    43,000000000h,0
```

Figure 3: Interrupt handler assembly language code generated by the C compiler for a trivial interrupt handler.

DRAFT

Generating the assembly listing provides the associated opcode:

```
140 000075 B556          PSH mmap(MDP)
    000077 98
```

From the mnemonic assembler manual we get the base opcode for the PSH when used with an smem.

```
1011 0101 AAAA AAAI
```

The 98 turns out to indicate that a mmap register is being accessed. The I if 1 means indirect addressing.

Our mystery instruction accesses address 0x2B which is DPH, the extended data page pointer. What ever happened to calling a spade a spade?

When the compiler mode bit (CPL) is clear relative addressing is relative to the data pointer. When the CPL is one (normally this way under C?) relative addressing is relative to the stack pointer.

- Next the stack pointer is modified. Focusing on the operations involving the stack pointer we have
  - The contents XAR1 is put onto the stack
  - The contents of the DPH is put onto the stack
  - The contents of the stack pointer is placed into XAR1.
  - The value in the stack pointer is made even.
  - The saved SP value is stacked.
    This guarantees the address in SP odd.
  - The stack pointer contents is decremented.
    This guarantees the address pointed to by SP even.
  - The stack pointer contents is incremented.
    This makes the address pointed to by SP odd.
  - The value on the top of the stack is put into the stack pointer.
    This restores the contents of the SP.
  - The value of the DPH is restored from the stack.
  - The value of XAR1 is restored from the stack.

The best guess is that the value in SP is being put into a known even/odd state so that 32-bit values can be properly allocated, if need be. None of the lines of code involved with accessing the flag register make use of the stack pointer.

Apparently the stack maintenance done in this code is *boiler plate*.

DRAFT

- Status register 1 bits are cleared and set.

  Bit 8 set to enable sign extension mode.
  Bit 14 set to enable compiler mode.

- Status register 2 bits are cleared and set.

  Bit 15 set to enable arms addressing mode.

- The data pointer is set up to point to the uninitialized (.bss) section.
- In spite of all the set up a value is directly placed into the `U1RxChar` register without making any use of either the stack or the data pointer.
- The compiler cleans up after itself and has the handler return from the the interrupt.

Keep in mind this is small memory model. If I were to write this in assembly language the code would consist of one code line

```
MOV #1, *abs16(#_U1RxChar) ; |41|
```

followed by the obligatory 6 `nops` and a `reti`.

We are ready to write and test a simple interrupt hander. For the first go we will only service interrupts from the UART receiver. When a character is received a flag register, `U1RxFlag`

4.2.2 Read a character under interrupts handler

With some diligent effort the set up code becomes

```
void UART1setup(void)
{
    unsigned long resetloc;

    resetloc = (long)resetv;
    far_poke(IVPD, (unsigned)(resetloc>>8));
    far_poke(IVPH, (unsigned)(resetloc>>8));
    far_poke((resetloc>>1)+INT0, (unsigned)((unsigned long)UART1int>>16));
    far_poke((resetloc>>1)+INT0+1, (unsigned)((unsigned long)UART1int));
    far_poke(IER0, far_peek(IER0)|INT0_BIT);
    far_poke(IFR0, INT0_BIT);
    U1RxFlag = 0;
    far_poke(IER, 0x01); // have UART generate receive interrupts
    far_poke(MCR, 0x08); // make UART int outputs active
    far_peek(RBR);       // clear receiver buffer
    return;
}
```


DRAFT

The line of code making the UART interrupt lines active by setting bit 4 in the modem control register (this is an obvious place to do this?) took several hours to find out that this action was necessary and where it was to be accomplished.

Our test code now is

```
if (test == 6) { // Tests for developing interrupt handler
    UART1setup();
    _enable_interrupts();
    while(FOREVER) {
        while (U1RxFlag == 0);
        far_poke(THR, U1RxChar);
        U1RxFlag = 0;
    }
}
```

This waits for a character to have read by the interrupt support, echoes to the UART (non-interrupt), clears the flag and then goes back to waiting.

Our simple only read a character interrupt support becomes

```
interrupt void UART1int(void)
{
    if ((far_peek(LSR)&0x01)==1) {        // test is Rx interrupt
        U1RxChar = far_peek(RBR);     // read character
        U1RxFlag = 1;                // set flag
    }
    return;
}
```

Here is the associated assembly language code generated by the C compiler *with full optimization enabled*!

```
    .sect   ".text"
    .align 4
    .global _UART1int
;******************************************************************************
;* INTERRUPT NAME: _UART1int                                                  *
;*                                                                            *
;*   Function Uses Regs : AC0,AC0,AC1,AC2,AC3,T0,T1,AR0,XAR0,AR1,XAR1,AR2,XAR2,*
;*                        AR3,XAR3,AR4,XAR4,AR5,XAR5,AR6,XAR6,AR7,XAR7,SP,BKC, *
;*                        BK03,BK47,ST1,ST2,ST3,MDP,BRC0,RSA0,REA0,BRS1,BRC1,  *
;*                        RSA1,REA1,CSR,RPTC,CDP,XCDP,TRN0,TRN1,BSA01,BSA23,   *
;*                        BSA45,BSA67,BSAC,TC1,M40,SATA,SATD,RDM,FRCT,SMUL     *
;*   Save On Entry Regs : AC0,AC0,AC1,AC1,AC2,AC2,AC3,AC3,T0,T1,AR0,AR1,AR2,   *
;*                        AR3,AR4,AR5,AR6,AR7,BKC,BK03,BK47,ST3,MDP,BRC0,RSA0, *
;*                        REA0,BRS1,BRC1,RSA1,REA1,CSR,RPTC,CDP,TRN0,TRN1,     *
;*                        BSA01,BSA23,BSA45,BSA67,BSAC                         *
;******************************************************************************
```

```
_UART1int:
        PSH dbl(AC0)
        PSH mmap(AC0G)
        PSH dbl(AC1)
        PSH mmap(AC1G)
        PSH dbl(AC2)
        PSH mmap(AC2G)
        PSH dbl(AC3)
        PSH mmap(AC3G)
        PSH T1, T0
        PSHBOTH XAR0
        PSHBOTH XAR1
        PSHBOTH XAR2
        PSHBOTH XAR3
        PSHBOTH XAR4
        PSHBOTH XAR5
        PSHBOTH XAR6
        PSHBOTH XAR7
        PSH mmap(BKC)
        PSH mmap(BK03)
        PSH mmap(BK47)
        PSH mmap(ST3_55)
        PSH mmap(MDP)
        PSH mmap(BRC0)
        PSH mmap(RSA0L)
        PSH mmap(RSA0H)
        PSH mmap(REA0L)
        PSH mmap(REA0H)
        PSH mmap(BRS1)
        PSH mmap(BRC1)
        PSH mmap(RSA1L)
        PSH mmap(RSA1H)
        PSH mmap(REA1L)
        PSH mmap(REA1H)
        PSH mmap(CSR)
        PSH mmap(RPTC)
        PSHBOTH XCDP
        PSH mmap(TRN0)
        PSH mmap(TRN1)
        PSH mmap(BSA01)
        PSH mmap(BSA23)
        PSH mmap(BSA45)
        PSH mmap(BSA67)
        PSH mmap(BSAC)
        AMAR *SP(#0), XAR1
        AND #0xfffe, mmap(SP)
        PSH AR1
        AND #0xf91f, mmap(ST1_55)
        OR #0x4100, mmap(ST1_55)
        AND #0xfa00, mmap(ST2_55)
        OR #0x8000, mmap(ST2_55)
```

DRAFT

```
    .ref ___bss__
        MOV #((___bss__ >> 16) & 0x7f), MDP
        AMOV #___bss__, XAR0
        AMAR *AR0, XAR1
        AMAR *AR0, XAR2
        AMAR *AR0, XAR3
        AMAR *AR0, XAR4
        AMAR *AR0, XAR5
        AMAR *AR0, XAR6
        AMAR *AR0, XAR7
        AMAR *AR0, XCDP
        MOV #80 << #16, AC0 ; |62|
        BSET ST3_SMUL
        BCLR ST3_SATA
        AADD #-1, SP
        OR #0x020a, AC0, AC0 ; |62|
        CALL #_far_peek ; |62|
                                        ; call occurs [#_far_peek]  ; |62|
        BTST @#0, AC0, TC1 ; |62|
        BCC L1,!TC1 ; |62|
                                        ; branch occurs ; |62|
        MOV #80 << #16, AC0 ; |63|
        OR #0x0200, AC0, AC0 ; |63|
        CALL #_far_peek ; |63|
                                        ; call occurs [#_far_peek]  ; |63|
        MOV AC0, *abs16(#_U1RxChar) ; |63|
        MOV #1, *abs16(#_U1RxFlag) ; |64|
L1:
        AADD #1, SP
        POP mmap(SP)
        POP mmap(BSAC)
        POP mmap(BSA67)
        POP mmap(BSA45)
        POP mmap(BSA23)
        POP mmap(BSA01)
        POP mmap(TRN1)
        POP mmap(TRN0)
        POPBOTH XCDP
        POP mmap(RPTC)
        POP mmap(CSR)
        POP mmap(REA1H)
        POP mmap(REA1L)
        POP mmap(RSA1H)
        POP mmap(RSA1L)
        POP mmap(BRC1)
        POP mmap(BRS1)
        POP mmap(REA0H)
        POP mmap(REA0L)
        POP mmap(RSA0H)
        POP mmap(RSA0L)
        POP mmap(BRC0)
```


DRAFT

```
        POP mmap(MDP)
        POP mmap(ST3_55)
        POP mmap(BK47)
        POP mmap(BK03)
        POP mmap(BKC)
        POPBOTH XAR7
        POPBOTH XAR6
        POPBOTH XAR5
        POPBOTH XAR4
        POPBOTH XAR3
        POPBOTH XAR2
        POPBOTH XAR1
        POPBOTH XAR0
        POP T1,T0
        POP mmap(AC3G)
        POP dbl(AC3)
        POP mmap(AC2G)
        POP dbl(AC2)
        POP mmap(AC1G)
        POP dbl(AC1)
        POP mmap(AC0G)
        POP dbl(AC0)
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        NOP       ;     avoids Silicon Exception CPU_99/100
        RETI
                                        ; return occurs
```

Not knowing what to save the C compiler saves everything. This most likely was triggered by the use of function calls within the interrupt handler.

5 "Final" buffered UART channel 2 interrupt support

What follows was built on the results of the preceding experiments. It is shown in what is nominally final (working) form.

5.1 Application level code

There are two functions. One to get characters from the UART receive (Rx) buffer and one to put characters into the UART transmit (Tx) buffer.

```
    RxGet:
        while (no character in buffer) wait
        fetch character from buffer
        advance the Rx get pointer cyclicly
```


DRAFT

```
        enter critical section
            decrement the count of values in the buffer
        exit critical section
        return character

    TxPut:
        while (no space in Tx buffer) wait
        enter critical section
            if (not waiting for an interrupt) begin
                put character into UART Tx register
                enable Tx empty interrupt
                clear waiting switch
            else
                put character into Tx buffer
                advance Tx put pointer cyclicly
                increase count of values present by 1
            end
        exit critical section
        return
```

I was a bit heavy handed on the size of the critical section. With some work it should be possible to reduce its size.

The sending of characters to the UART for transmission basically works as follows:

- A character is placed into the transmit buffer, THR.
- This clears the THR empty flag.
- Some time later the contents of the THR is transferred into the transmitter serialization register.
- This empties the THR causing the THR empty flag to become set.
- A set THR flag causes an interrupt.
- The associated interrupt handler fetches a character from the Tx buffer and places it into the THR.
- This clears the THR empty flag.
- And so on.

There are two special cases that need to be taken care of:

1. Startup. This can be folded into the following case.

2. Restarting after the Tx buffer is emptied and the interrupt occurs indicating the THR is also empty. There is no value to be placed into the THR clearing the request. This is handled below by disabling responding to the THR empty flag being set. When it is time to transmit an another character and this state exists special action is necessary. The code given below simply loads the character into the THR (clearing the flag) and then enables the associated interrupt.

DRAFT

```
// Function to fetch characters from the Rx buffer

int U2RxGet(void)
{
    int itemp;

    while (U2RxCount == 0);    // wait if no characters present

    itemp = *(U2RxAdr+U2RxAOff);             // fetch character
    if (++U2RxAOff >= U2RxSize) U2RxAOff = 0; // adv pointer cyclicly

    _disable_interrupts();     // enter a critical section
    U2RxCount--;               // reduce the number present
    _enable_interrupts();      // exit the critical section

    return (itemp);            // return the fetched character
}

// Function to put characters into the Tx buffer

void U2TxPut(int value)
{
    while (U2TxCount == U2TxSize);  // wait if no room in the buffer

    _disable_interrupts();          // enter a critical section

    if (U2TxStopped != 0) {                  // if Tx not running
        far_poke(THR, value);                // load character directly
        far_poke(IER, far_peek(IER)|0x0002); // reenable interrupt
        U2TxStopped = 0;                     // and note that is expected
    }
    else {                                   // if Tx is running
        *(U2TxAdr+U2TxAOff) = value;         // put character into buffer
        if (++U2TxAOff >= U2TxSize) U2TxAOff = 0; // adv pointer cyclicly
        U2TxCount++;                              // and increase the count
    }

    _enable_interrupts();          // exit the critical section

    return;
}
```

5.2 Interrupt level code

The comments pretty much serve as the pseudo code describing what is being done.

```
// Interrupt handler for UART channel 2

int volatile U2Flag;
```


DRAFT

```
interrupt void UART2int(void)
{
    U2Flag = far_peek(ISR);           // get the interrupt status value
    if ((U2Flag&0x0C) != 0) {         // true if Rx interrupt
       if (U2RxCount < U2RxSize) {    // ignore if no room
           *(U2RxAdr+U2RxIOff) = far_peek(RBR); // put character in Rx buffer
           if (++U2RxIOff >= U2RxSize) U2RxIOff = 0; // adv pointer cyclicly
           U2RxCount++;                                // count the character
       }
       else {                         // if no room in the Rx buffer we
           far_peek(RBR);             // fetch the character and discard it
       }
   }
   else if ((U2Flag&0x02)!=0) {      // true if Tx interrupt
       if (U2TxCount == 0) {         // true if no characters in Tx buffer
           U2TxStopped = 1;          // so note that no future interrupt
           far_poke(IER, far_peek(IER)&0xFFFD); // and disable Tx interrupt requests
       }
       else {                        // otherwise have a character to send
           far_poke(THR, *(U2TxAdr+U2TxIOff));      // so put into the Tx buffer
           if (++U2TxIOff >= U2TxSize) U2TxIOff = 0; // adv pointer cyclicly
           U2TxCount--;                                // reduce count present
       }
   }
   else {
        while(1);  // should never get here..but if we do, wait for help
   }
   return;
}
```

5.3 Initialization code

Note that the size and location of the buffers and the baud rate divisor are passed as parameters to the initialization code. This gives the user great freedom in how the support is used and avoids building constants into the UART support itself.

```
/* Function to set up the UART channel buffered Rx and Tx support.

   Call prior to globally enabling the interrupt system.

   The arguments are:

   *intbuf    A pointer to the receive (Rx) buffer.
   nin        The number of ints in the Rx buffer.
   *outbuf    A pointer to the transmit (Tx) buffer.
   nout       The number of ints in the Tx buffer.
   RateDiv    The rate divisor value baud rate is
```


DRAFT

```
              230,400/RateDiv.  For 38400 baud use a
              value of 6.
*/

void UART2setup(int *inbuf, int nin, int *outbuf, int nout, int RateDiv)
{
    unsigned long resetloc;

    // set up buffering support

    U2RxAdr = inbuf;    // address of Rx buffer
    U2RxSize = nin;     // size of Rx buffer
    U2RxAOff = 0;       // application level Rx address offset
    U2RxIOff = 0;       // interrupt level Rx address offset
    U2RxCount = 0;      // count of characters in Rx buffer
    U2TxAdr = outbuf;   // address of Tx buffer
    U2TxSize = nout;    // size of Tx buffer
    U2TxAOff = 0;       // application level TX address offset
    U2TxIOff = 0;       // interrupt level Tx address offset
    U2TxCount = 0;      // count of characters in Tx buffer
    U2TxStopped = 1;    // Tx waiting for a character to be sent

    // set up interrupt vector and interrupt registers

    resetloc = (long)resetv;
    far_poke(IVPD, (unsigned)(resetloc>>8));
    far_poke(IVPH, (unsigned)(resetloc>>8));
    far_poke((resetloc>>1)+INT0, (unsigned)((unsigned long)UART2int>>16));
    far_poke((resetloc>>1)+INT0+1, (unsigned)((unsigned long)UART2int));
    far_poke(IER0, far_peek(IER0)|INT0_BIT);
    far_poke(IFR0, INT0_BIT);

    // configure the UART channel 2

    far_poke(LCR, 0x80); // access baud rate registers
    far_poke(DLM, RateDiv>>8); // set baud rate divisor high byte
    far_poke(DLL, RateDiv);    // set baud rate divison low byte
    far_poke(LCR, 0x07); // use 8 data and 2 stop bits

    far_poke(FCR, 0x00); // insure FIFOs are off
    far_poke(LSR, 0x60); // initialize line status register
    far_poke(IER, 0x03); // have UART generate Rx and Tx interrupts
    far_poke(MCR, 0x08); // make UART int req outputs active
    far_peek(RBR);       // clear receiver buffer

    return;
}
```

DRAFT

5.4 Test program

Hopefully the code works in other situations beyond those tested here.

```
/* File name: INTtest.c

    30Mar2004 .. intial version .. KM

*/

#include <stdio.h>

#define FOREVER 1

/* Allocate UART channel 2 buffers.  The U2RxN and U2TxN define
   the number of characters contained in the Rx and Tx buffers
   respectively.
*/

#define U2RxN 12
#define U2TxN 6
int U2RxIn[U2RxN], U2TxOut[U2TxN]; // define Rx and Tx buffers for UART channel 2

void UART2setup(int *inbuf, int nin, int *outbuf, int nout, int divisor);

int test = 1;
int CHar;

/* baud rate is 230,400/divisor
   for 38400 baud use a value of 6
*/

int RateDivisor=6; // to give 38400 baud

void main(void)
{

    if (test == 1) {  // read input character and echo versions

        UART2setup(U2RxIn, U2RxN, U2TxOut, U2TxN, RateDivisor);
        _enable_interrupts();

        while (FOREVER) {
            CHar = U2RxGet();
            U2TxPut(CHar++);
            U2TxPut(CHar++);
            U2TxPut(CHar++);
            U2TxPut(CHar);
        }
    }
```

DRAFT

```
    if (test == 2) {     // sort of stress the output support

        UART2setup(U2RxIn, U2RxN, U2TxOut, U2TxN, RateDivisor);
        _enable_interrupts();

        CHar = 0;
        while (FOREVER) {
            U2TxPut(CHar++);
            U2TxPut(CHar++);
            U2TxPut(CHar++);
            U2TxPut(CHar++);
        }
    }
}
```

6 Comments

The overhead associated with the C interrupt handler rivals if not exceeds the time required by the executing code. The logic associated with both the application and the interrupt support is not very complicated and it would be quite easy to convert this into assembly language. The result should be more time and space efficient.

The above code is easily modified to create matching support for UART channel 1. Properly done, the two sets of code will be completely independent and work well side by side. The lab UART boards are set up for UART channel 1 to use C5510 interrupt 2.

DRAFT