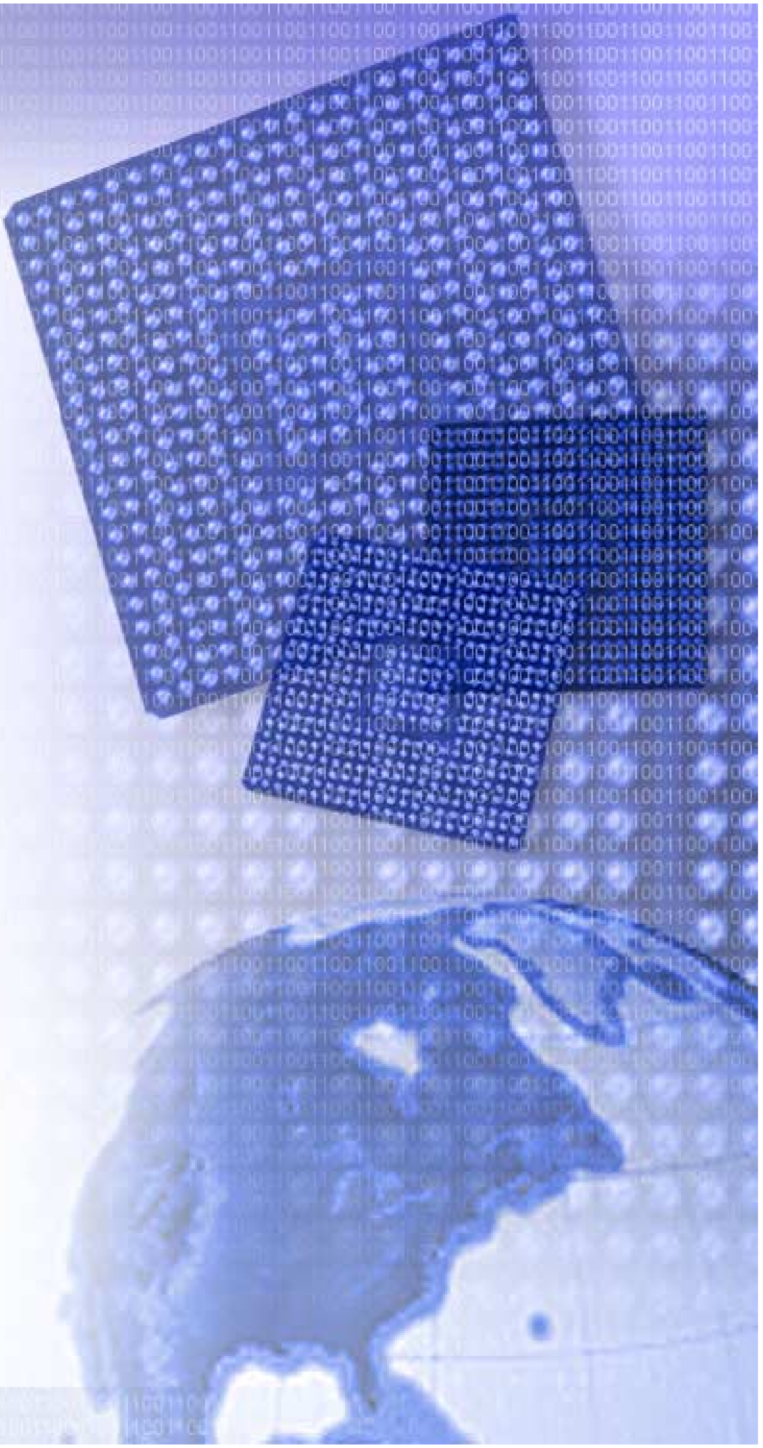
KCPSM3 Reference Design

UART Real Time Clock

Ken Chapman

Xilinx Ltd

September 2003 Rev.1



Limitations

This module is **<u>not</u>** supported by general Xilinx Technical support as an official Xilinx Product.
Please refer any issues initially to the provider of the module.

Any problems or items felt of value in the continued improvement of KCPSM3 and this reference design would be gratefully received by the author.

Ken Chapman
Staff Engineer - Applications Specialist
email: chapman@xilinx.com

The author would also be pleased to hear from anyone using any version of PicoBlaze or the UART macros with information about your application and how these macros have been useful.

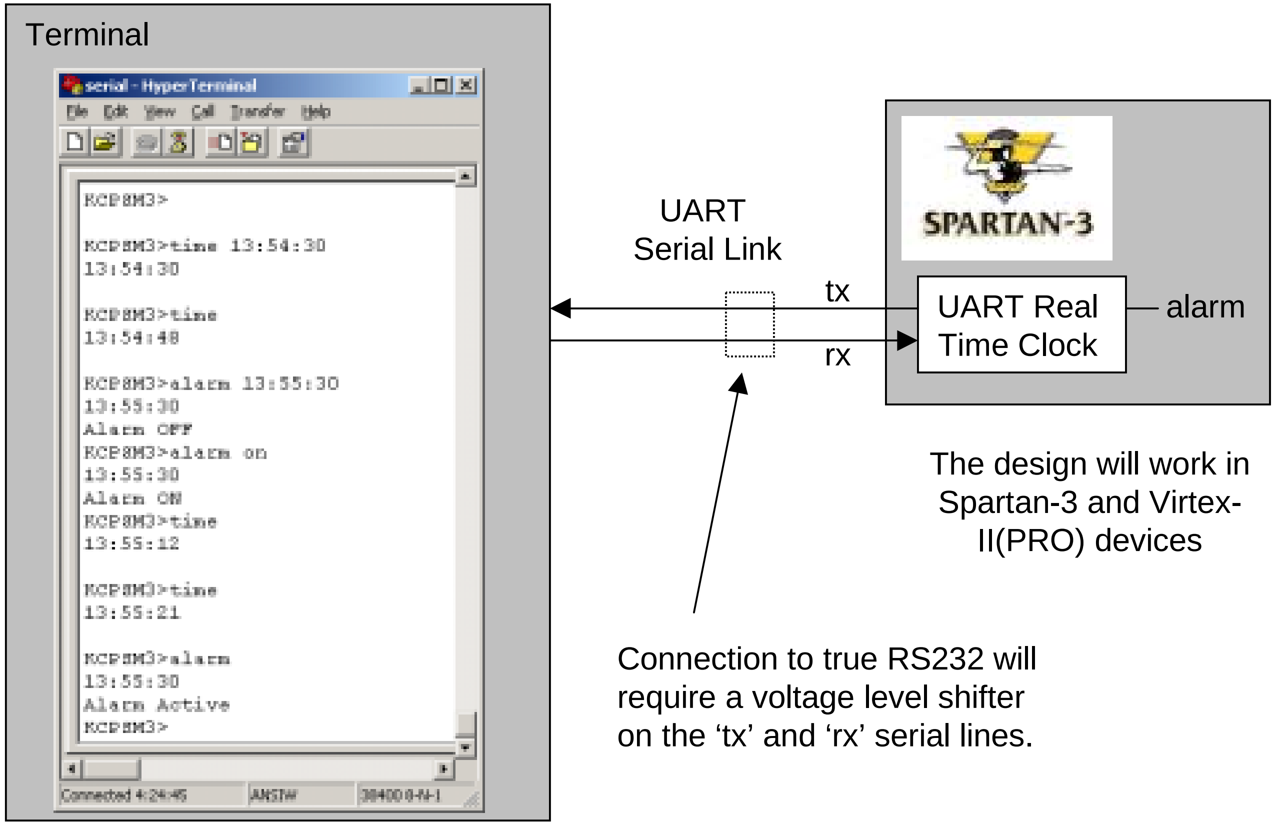
What is a UART Real Time Clock ?

The main purpose of this package is as a reference design for the KCPSM3 variant of PicoBlaze although it is hoped that some people may use it directly in there own product designs as a suitable 'IP core'. The design as a whole implements a simple real time clock maintaining time in hours, minutes and seconds together with the ability to set an alarm. The unusual feature of the design is that a UART serial communication is used to set an observe the time and alarm via simple text commands and messages.

HyperTerminal on a PC provides a suitable terminal to send commands and observe messages.

The design even supports some minor editing facilities during command entry.

The design may also be connected as a UART linked peripheral to a MicroBlaze or PowerPC processor in order to provide an independent real time clock facility. The 'alarm' signal may then be used to interrupt the host processor.



```
KCPSM3>

KCPSM3>time 13:54:30
13:54:30

KCPSM3>time
13:54:48

KCPSM3>alarm 13:55:30
13:55:30
Alarm OFF
KCPSM3>alarm on
13:55:30
Alarm ON
KCPSM3>time
13:55:12

KCPSM3>time
13:55:21

KCPSM3>alarm
13:55:30
Alarm Active
KCPSM3>
```

The design will work in Spartan-3 and Virtex-II(PRO) devices

Connection to true RS232 will require a voltage level shifter on the 'tx' and 'rx' serial lines.

Reference Design

As a reference design, the supplied VHDL and assembler code illustrates a design based on the KCPSM3 variant of PicoBlaze and demonstrates the following…...

Hardware VHDL design

- Connection of KCPSM3 to the Program ROM.
- Connection of UART macros supplied with KCPSM3 - Input and output ports and baud rate timing.
- Interrupt generation with a fixed interval timer and use of interrupt acknowledge signal.

Software KCPSM3 design

- Use of CONSTANT directives to define ports, bits within ports, scratch pad memory locations and useful values.
- Interrupt Service Routine (ISR).
- Use of scratch pad memory.
- Communications with UART receiver and transmitter.
- ASCII string handling.
- Use of COMPARE and TEST instructions which are new with the KCPSM3 variant of PicoBlaze.

Useful PSM subroutines include….

- UART communications.
- ASCII to decimal and decimal to ASCII conversions.
- Lower case to upper case character conversion.
- UART character string reading and editing.
- Real time clock.
- List of ASCII constants.

Understanding the Design

To begin to use the design as a reference, it is useful to understand what it does and ideally to try it out with some suitable hardware and a PC. The design is supplied as just two files…..

The hardware definition is supplied in a VHDL file called ‘uart_clock.vhd. This will need to be added to a suitable project and either used directly as the top level of your project or as a macro within a larger design. To complete the hardware definition you will need to add all the VHDL files from the KCPSM3 (PicoBlaze) package, which also includes the UART macros.

The KCPSM3 software is provided as ‘uclock.psm’ which you will need to assemble using the KCPSM3 assembler.

Recommendation

Using the KCPSM3 documentation for guidance, ensure that you can assemble the supplied PSM program and synthesize the complete design into a Spartan3 or Virtex-II(PRO) device. These files are known to work and will help you to resolve any design flow issues.

If you are fortunate enough to have some suitable hardware to actually try the design, you will need to connect the UART to a couple of pins. A User Constraints File (UCF) can be used to specify the physical pins and provide timing constraints.

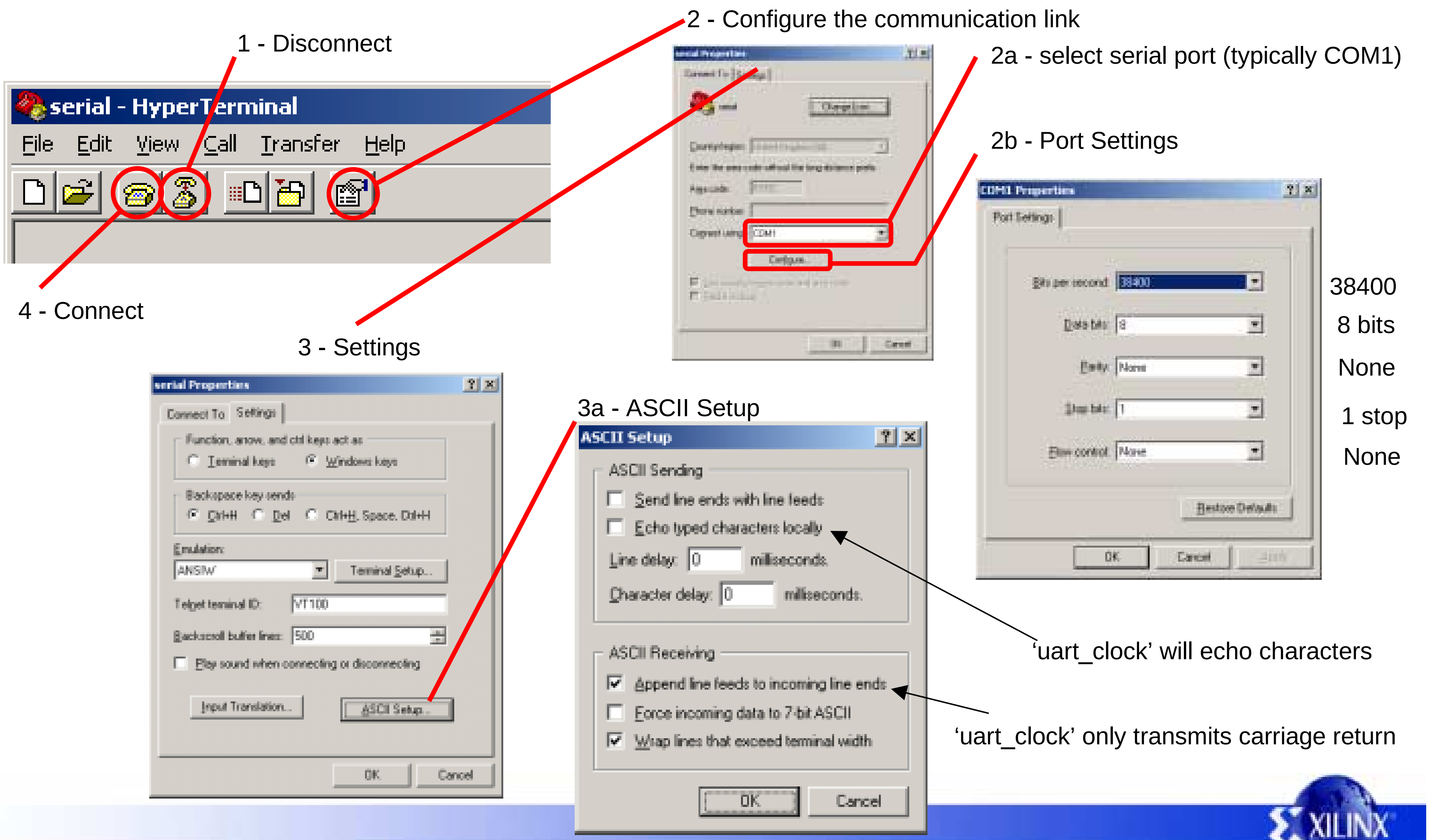
```
TIMESPEC TS01 = FROM : FFS  : TO : FFS  : 18 ns;
TIMESPEC TS02 = FROM : RAMS : TO : FFS  : 18 ns;
TIMESPEC TS03 = FROM : FFS  : TO : RAMS : 18 ns;
TIMESPEC TS04 = FROM : RAMS : TO : RAMS : 18 ns;
TIMESPEC TS05 = FROM : FFS  : TO : PADS : 18 ns;
TIMESPEC TS06 = FROM : PADS : TO : FFS  : 18 ns;
TIMESPEC TS07 = FROM : PADS : TO : RAMS : 18 ns;
NET "tx"    LOC = "A10";
NET "rx"    LOC = "A7";
NET ”alarm" LOC = "J15";
```

These specifications should provide 100% coverage.
(18ns is constant with a 55MHz clock)

It is nice to make ‘alarm’ available on a pin (ideally an LED) but you could remove it from the design or leave it disconnected.

HyperTerminal Setup

HyperTerminal running on a PC can provide a very simple terminal for communication with 'uart_clock'. Here are the required settings for the design and how to configure HyperTerminal.


1 - Disconnect
2 - Configure the communication link
2a - select serial port (typically COM1)
2b - Port Settings
4 - Connect
3 - Settings
3a - ASCII Setup
serial - HyperTerminal
File Edit View Call Transfer Help

serial Properties – Connect To: Connect using: COM1 – Configure...

COM1 Properties – Port Settings:

Bits per second:	38400	38400
Data bits:	8	8 bits
Parity:	None	None
Stop bits:	1	1 stop
Flow control:	None	None

serial Properties – Settings:

Function, arrow, and ctrl keys act as: Terminal keys [✓ Windows keys]

Backspace key sends: [✓ Ctrl+H] Del Ctrl+H, Space, Ctrl+H

Emulation: ANSIW

Telnet terminal ID: VT100

Backscroll buffer lines: 500

☐ Play sound when connecting or disconnecting

ASCII Setup

ASCII Sending

☐ Send line ends with line feeds

☐ Echo typed characters locally — 'uart_clock' will echo characters

Line delay: 0 milliseconds.

Character delay: 0 milliseconds.

ASCII Receiving

☑ Append line feeds to incoming line ends — 'uart_clock' only transmits carriage return

☐ Force incoming data to 7-bit ASCII

☑ Wrap lines that exceed terminal width

Commands

The UART Real Time Clock design understands some simple ASCII commands and even supports some editing during their entry using the backspace key on your keyboard. A command is only competed when carriage return is entered. The design is ready to accept a command when the "KCPSM3>" prompt is displayed.

The 'uclock' program is able to interpret upper and lower case characters by converting commands to upper case before analysing them. Incorrect commands will result in a "Syntax Error" message and incorrect time values will be indicated by an "Invalid Time" message. Although it is unlikely to occur when using HyperTerminal, an "Overflow Error" message will be generated if commands are transmitted faster than the design can process them (i.e. the UART receiver buffer becomes full).

TIME The current time will be displayed in the form hh:mm:ss

TIME hh:mm:ss Allows the time to be set. The values of 'hh', 'mm' and 'ss' must provide a valid time using a 24-hour clock. The new time will then be displayed.

```
KCPSM3>time
16:54:28

KCPSM3>time 17:23:56
17:23:56
```

ALARM The current alarm time will be displayed in the form hh:mm:ss together with the current status of the alarm (OFF, ON, or Active)

ALARM hh:mm:ss Allows the alarm time to be set. The values of 'hh', 'mm' and 'ss' must provide a valid time using a 24-hour clock. The new alarm time and current status will then be displayed.

```
KCPSM3>alarm 07:15:00
07:15:00
Alarm OFF
KCPSM3>alarm 17:28:55
17:28:55
Alarm OFF
KCPSM3>
```

ALARM ON Enables the alarm to become active. The current alarm time and status will be displayed

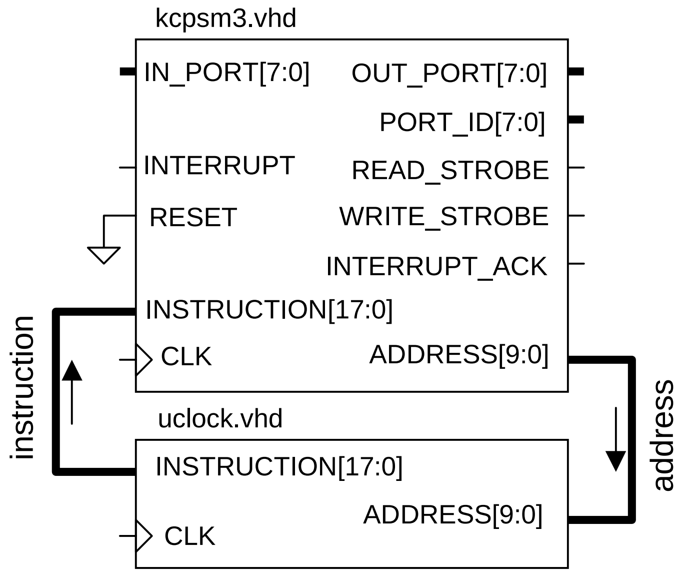
ALARM OFF Disables the alarm from becoming active and will cancel the alarm if already active. The current alarm time and status will be displayed

```
KCPSM3>alarm on
17:28:55
Alarm ON
KCPSM3>alarm
17:28:55
Alarm Active
KCPSM3>alarm off
17:28:55
Alarm OFF
KCPSM3>
```

Note - All commands and responses are defined by the KCPSM3 program and is therefore easy to modify. The program is less than 50% of the available program space available allowing the potential for many more commands and features.

The Hardware - Program ROM

The connection of the program ROM is a straight forward. The KCPSM3 processor and the ‘uclock’ program are instantiated and signals used to make direct connections. In this design the reset port is not required so it is tied to ground.

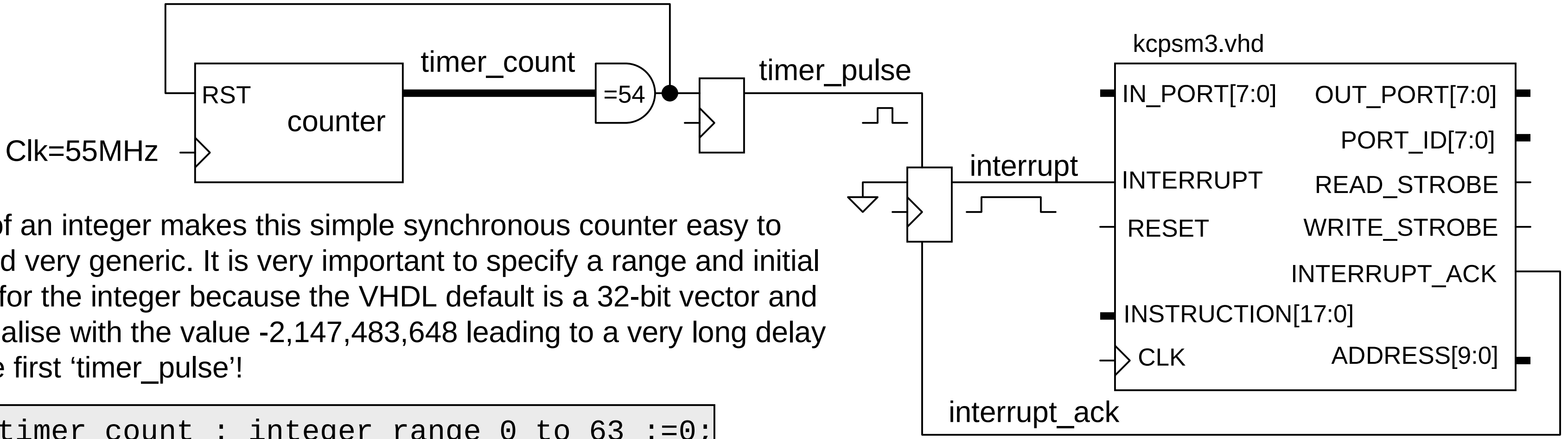


```
processor: kcpsm3
   port map(      address => address,
             instruction => instruction,
                 port_id => port_id,
            write_strobe => write_strobe,
                out_port => out_port,
             read_strobe => read_strobe,
                 in_port => in_port,
               interrupt => interrupt,
           interrupt_ack => interrupt_ack,
                   reset => ‘0’,
                     clk => clk);

 program_rom: uclock
   port map(      address => address,
             instruction => instruction,
                     clk => clk);
```

The complete reference design is connected to a single clock source ‘clk’. We will see this signal used in the more generic VHDL code, but here it is again a direct connection to the instantiated components.

The Hardware - Interrupt

The interrupt in this design is used to provide a regular timing reference for the clock in the form of 1µs pulses. This is achieved by counting system clock cycles. In the example code, the counter has 55 states (0 to 54) to be consistent with a 55MHz clock. This is a functional diagram of the logic and may not represent the exact implementation style used by the synthesis tool. However, the code is specifically written to use synchronous controls and provide pipelining where appropriate.



The use of an integer makes this simple synchronous counter easy to modify and very generic. It is very important to specify a range and initial condition for the integer because the VHDL default is a 32-bit vector and would initialise with the value -2,147,483,648 leading to a very long delay before the first 'timer_pulse'!

```
signal timer_count : integer range 0 to 63 :=0;
```

```
if clk'event and clk='1' then
  if timer_count=54 then
    timer_count <= 0;
    timer_pulse <= '1';
   else
    timer_count <= timer_count + 1;
    timer_pulse <= '0';
  end if;
end if;
```

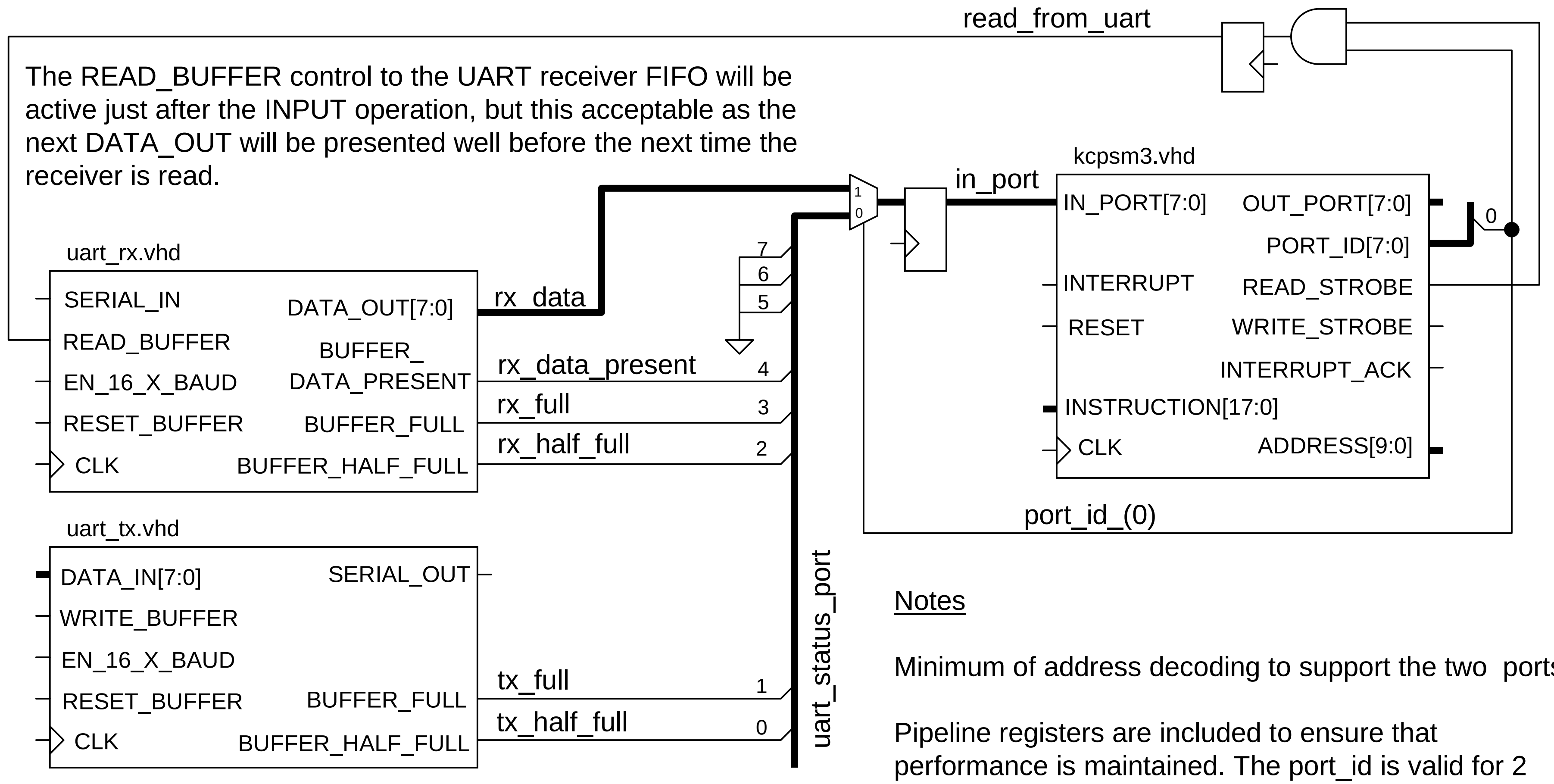
Hint - SRL16E based shift registers will often result in a more efficient clock divider.

Using 'interrupt_ack' ensures that each pulse is processed by KCPSM3 even when interrupts are temporarily disabled.

```
if clk'event and clk='1' then
  if interrupt_ack = '1' then
    interrupt <= '0';
   elsif timer_pulse = '1' then
    interrupt <= '1';
   else
    interrupt <= interrupt;
  end if;
end if;
```

The Hardware - Inputs

The UART macros provide the only inputs to KCPSM3. There is the data received from the ‘rx’ serial input and captured by the UART_RX macro, but there is also the status signals associated with the FIFO buffers within the UART receiver and transmitter macros which KCPSM3 will need to monitor. This leads to a simple multiplexer and address decoding logic.



Notes

Minimum of address decoding to support the two ports.

Pipeline registers are included to ensure that performance is maintained. The port_id is valid for 2 clock cycles during an INPUT operation.

The Hardware - Inputs

The input logic is straight forward to define in VHDL, but it would be very easy to increase the complexity of the logic with resulting higher cost and lower performance. Note the minimum decoding implied by the code and the inclusion of pipeline registers. There is no need for a global reset, as is the case with the majority of designs.

```
uart_status_port <= "000" & rx_data_present & rx_full & rx_half_full & tx_full & tx_half_full;
```

Grouping the individual UART status signals into a bus makes the code easy to read. The MSB's have been forced low in this example, but these could also be masked or ignored in the KCPSM3 program.

```
if clk'event and clk='1' then

  case port_id(0) is

    -- read UART status at address 00 hex
    when '0' =>    in_port <= uart_status_port;

    -- read UART recieve data at address 01 hex
    when '1' =>    in_port <= rx_data;

    when others =>    in_port <= "XXXXXXXX";

  end case;


  -- Form read strobe for UART receiver FIFO buffer.
  read_from_uart <= read_strobe and port_id(0);

end if;
```

The 'port_id' is valid for 2 clock cycles during an INPUT operation so the multiplexer can be pipelined.

Use the minimum number of 'port_id' bits to control the multiplexer. In this design only bit0 is ever used.

Avoid using the common syntax….
```
      when others => null;
```
In designs were there are logical 'others', the 'null' will result in feedback to latch the valid last state resulting in unnecessary functionality with associated cost.

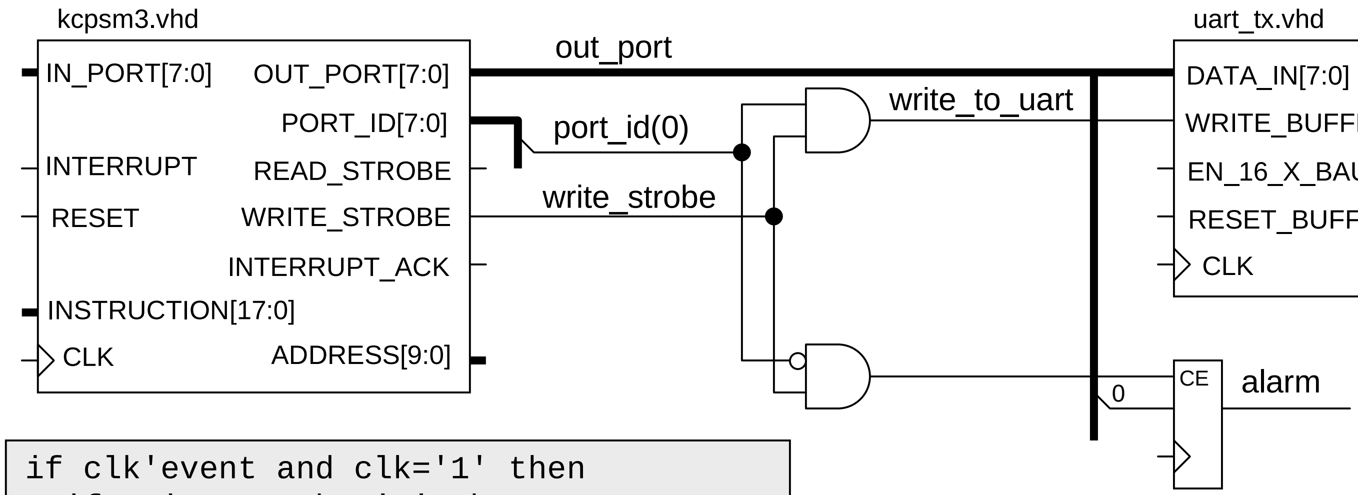
It is vital that 'read_strobe' is used to qualify the 'port_id' when reading the FIFO. 'port_id' changes during execution of all instructions and not just I/O operations.

The Hardware - Outputs

Data is output to the UART transmitter and one simple register. Again the minimum of address decoding can be employed.

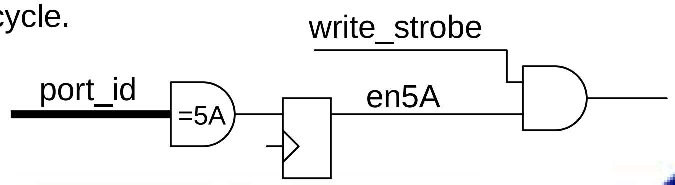
Since the FIFO inside the UART macro effectively provides the destination of the data, the design only needs to provide the address decoding. Since 'write_strobe' is only active for one clock cycle, this decode must be performed combinatorially.

```
write_to_uart <= write_strobe and port_id(0);
```



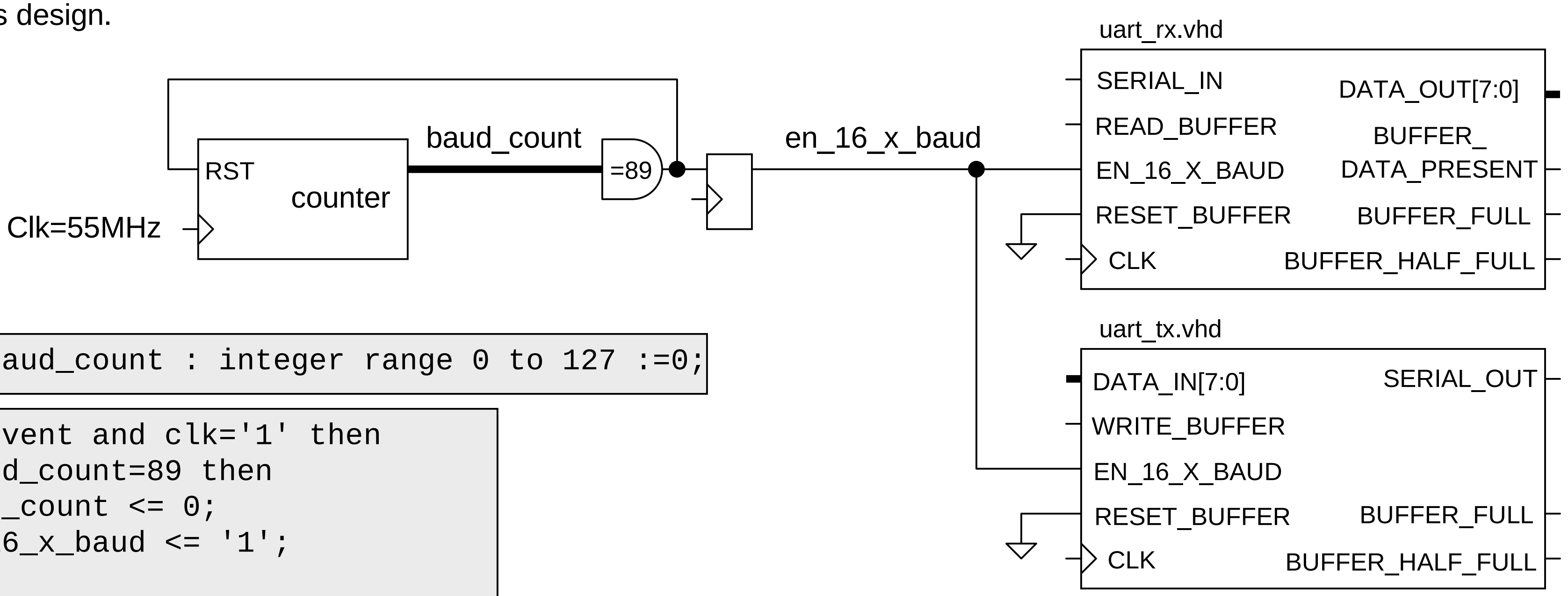
```
if clk'event and clk='1' then
  if write_strobe='1' then

    -- Alarm register at address 00 hex
    if port_id(0)='0' then
      alarm <= out_port(0);
    end if;

  end if;
end if;
```

By keeping the 'port_id' decoding to an absolute minimum, it should not be necessary to add a pipeline stage in the 2-clock cycle 'port_id' path. When more complex decoding is required, the code can be modified to perform the bulk of the address decoding in the first clock cycle and then qualify with the 'write_strobe' using a simple AND gate in the second cycle.



The Hardware - UART

In the reference design provided, the UARTs are set up to operate at 38400 BAUD with a 55MHz reference clock. Documentation for the UART macros is provided separately, so here we just look at the connections and baud rate generation used in this design.



```
signal baud_count : integer range 0 to 127 :=0;
```

```
if clk'event and clk='1' then
  if baud_count=89 then
    baud_count <= 0;
    en_16_x_baud <= '1';
   else
    baud_count <= baud_count + 1;
    en_16_x_baud <= '0';
  end if;
end if;
```

$$\text{Clock division} = \frac{\text{clk\_rate}}{\text{BAUD} \times 16} = \frac{55{,}000{,}000}{38400 \times 16}$$

= 89.52 We can round this up to 90

Therefore 'baud_count' will count from 0 to 89.

In a very similar way to the interrupt timer, a simple counter can be used to generate the pulses required to define the UART communication rate. Make sure you define the integer range and initial condition. An SRL16E based shift register could also be used in this situation.

KCPSM3 Program Code

As a reference design, this document is not intended to teach how to write software or explain how the individual instructions for KCPSM3 operate. Coding styles are very much a personal choice, but it is always useful to know what is possible and to consider some useful techniques which enable code to be easier to write, manage and understand.

In the following pages, excerpts of the supplied 'uclock.psm' program are selected to highlight particular coding styles and illustrate how the features and instruction set of KCPSM3 can be used. There will be cases where more efficient code could be written and there will be cases where more readable code could be written, but it is hoped that even in the form supplied it will inspire you with some ideas of your own.

Finally, the program has been written using multiple sub routines. Since some of these provide useful functions such as communications with the UART macros, these may also be directly useful in other designs.

Hint - It will be useful to have a copy of the KCPSM3 documentation when reviewing the code.

Before you continue…..

Assemble the 'uclock.psm' program and become familiar with the files which KCPSM3 generates.

Take a look at the 'uclock.log' file…....

How much of the available program memory has been used?

Where is the interrupt service routine (ISR) located?

Four registers have been given special names. What are they?

Take a look at the 'constant.txt' file…....

Observe the relatively long list of constants that were declared.

Notice how many have the same value.

How many minutes are there in an hour (in hexadecimal)?

Register Names

sC	
sD	
sE	
sF	

Software - I/O Ports

The CONSTANT directive is provided to allow values to be given a name which are then used in the program code. Although optional, they are probably the most significant way to make your code easier to understand and portable. Is is therefore not surprising to find that the program begins with a list of constants.

Defining the I/O ports

Providing each port with a name really helps to make the code understandable and helps define the boundary between hardware and software where it is all too easy to make a mistakes. The **port address** values assigned here directly reflect the hardware.

```
CONSTANT UART_status_port, 00          ;UART status input
CONSTANT tx_half_full, 01              ;  Transmitter     half full - bit0
CONSTANT tx_full, 02                   ;    FIFO               full - bit1
CONSTANT rx_half_full, 04              ;  Receiver        half full - bit2
CONSTANT rx_full, 08                   ;    FIFO               full - bit3
CONSTANT rx_data_present, 10           ;              data present - bit4
;
CONSTANT UART_read_port, 01            ;UART Rx data input
;
CONSTANT UART_write_port, 01           ;UART Tx data output
;
CONSTANT alarm_port, 00                ;Alarm output
CONSTANT alarm_control, 01             ;     bit0
```

It is often useful to identify particular **bit positions** within a port. The UART status bits are an excellent example of this. Each of the 5 status bits are identified with the hexadecimal code for which only the corresponding bit position is high. It is a good idea to use the same names as used in the VHDL code to further solidify the bridge between hardware and software design.

rx_full = 08_{16} = 00001000_2

$\downarrow$

```
uart_status_port <= "000" & rx_data_present & rx_full & rx_half_full & tx_full & tx_half_full;
```

Software - Scratch Pad Memory

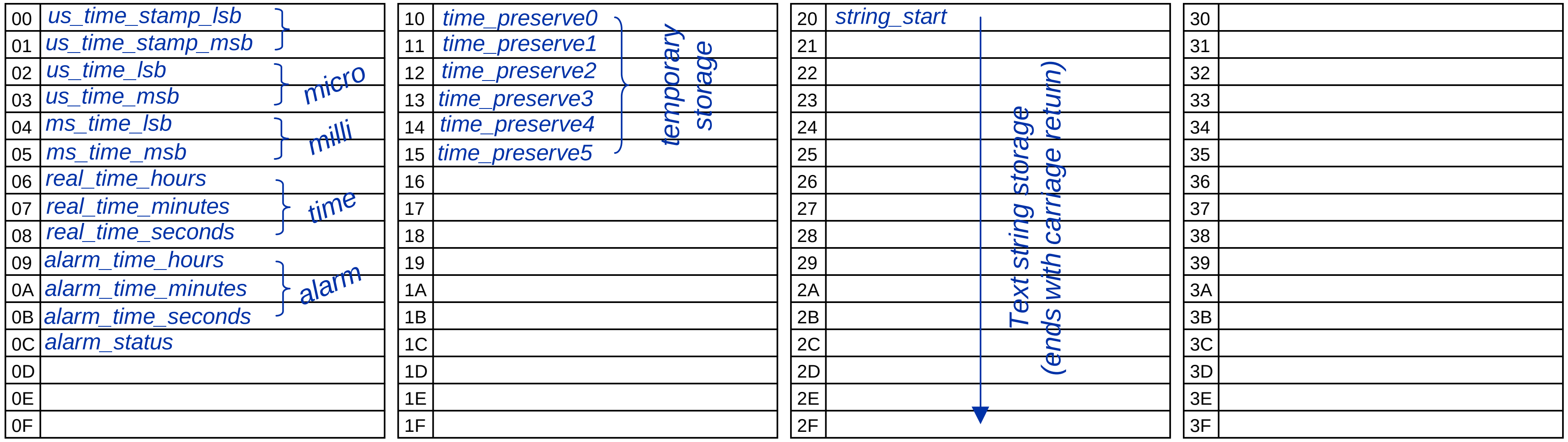
It useful to have a picture of how the scratch pad memory will be used. This ‘memory map’ will help you allocate locations and be a useful reference to the names you have allocated as you write your code. Below is the map for the ‘uclock’ program. At the end of this document you will find a set of clean resource charts for you to help with development of your own programs.

Defining the Scratch Pad Memory Locations

Constant values are now used to define the address within scratch pad memory at which a value is stored or at which the start of a block of data will be stored. Again, constants can be used to identify the special meaning of bits at some locations.

```
;Scratch Pad Memory Locations
;
CONSTANT us_time_stamp_lsb, 00           ;16-bit micro-second timestamp
CONSTANT us_time_stamp_msb, 01
```

Scratch Pad Memory



00	us_time_stamp_lsb	10	time_preserve0	20	string_start	30	
01	us_time_stamp_msb	11	time_preserve1	21		31	
02	us_time_lsb	12	time_preserve2	22		32	
03	us_time_msb	13	time_preserve3	23		33	
04	ms_time_lsb	14	time_preserve4	24		34	
05	ms_time_msb	15	time_preserve5	25		35	
06	real_time_hours	16		26		36	
07	real_time_minutes	17		27		37	
08	real_time_seconds	18		28		38	
09	alarm_time_hours	19		29		39	
0A	alarm_time_minutes	1A		2A		3A	
0B	alarm_time_seconds	1B		2B		3B	
0C	alarm_status	1C		2C		3C	
0D		1D		2D		3D	
0E		1E		2E		3E	
0F		1F		2F		3F	

Software - Constants

The final use of the CONSTANT directive is to define actual constant values which will be used to make the code easier to read and write. Note that these constants do not really need to be located at the beginning of the program. If they are significant to a given sub routine, then this is a good place to put them and will make the code more portable.

Defining useful values

Constants are used to define useful values where they are needed in the program. They are particularly good at helping to document the code automatically especially as the KCPSM3 assembler only directly supports hexadecimal values.

```
;Useful constants for real time clock operations
;
CONSTANT count_1000_lsb, E8              ;lower 8-bits of 1000 count value
CONSTANT count_1000_msb, 03              ;upper 8-bits of 1000 count value
CONSTANT hours_in_a_day, 18              ;24 hours in a day
CONSTANT minutes_in_an_hour, 3C          ;60 minutes in an hour
CONSTANT seconds_in_a_minute, 3C         ;60 seconds in a minute
```

$\} \ 1000_{10} = 03E8_{16}$

Constant values are applied globally to the program regardless of where they are defined. Therefore the very end of the program is a good place to define useful constant values even if they are not all required by the program.

```
;ASCII table
;
CONSTANT character_a, 61
CONSTANT character_b, 62
CONSTANT character_c, 63
CONSTANT character_d, 64
```

The end of 'uclock.psm' defines many of the characters in the ASCII table. These make it easier to define and interpret text messages. It is unlikely that every character is used in the program (and it isn't) but the complete definition is useful to copy to other programs and makes modifications to the program simple.

Note - The KCPSM3 assembler currently supports a maximum of 300 constants within a program, but this really should be adequate for any program. 'uclock.psm' defines just over 100 constants including the table of ASCII characters.

Software - Registers

KCPSM3 has 16 general purpose registers providing significant flexibility when writing programs. Indeed, many users of the original PicoBlaze (KCPSM) have reported that all the variables required by the application could be contained in these 16 registers without need for external memory. Under such conditions it is vital to manage the allocation of registers very well to prevent destruction of data by inadvertent reuse of a particular register in another part of the program. The NAMEREG directive is an ideal way to allocate registers and help prevent this undesirable situation.

The 'uclock.psm' only defines the names for 4 of the registers. Although any register can be renamed, by allocating the registers from the highest downwards, the lowest register numbers always remain. The lower registers with default names are then ideal as 'working registers' in sub routines. Adopting this style will make much of your code highly portable.

```
;Special Register usage
;
NAMEREG sF, UART_data             ;used to pass data to and from the UART
;
NAMEREG sE, store_pointer         ;used to pass location of data in scratch pad memory
;
;Two registers to form a 16-bit counter used to count
;interrupt pulses generated at 1us intervals.
;
NAMEREG sD, int_counter_lsb       ;lower 8-bits
NAMEREG sC, int_counter_msb       ;upper 8-bits
```

When writing sub routines it is recommended that you add comments naming the registers which are used. Some may be used to transfer information between the subroutine and the invoking level of code. It is likely that other registers will only be used as 'working registers' within the sub routine to perform the required operations. It is these 'working registers' that you need to be the most careful with to ensure that they do not modify valuable data.

```
               ;Registers used s0, s1, s2, 'store_pointer' and 'UART_data'.
               ;
transmit_time: LOAD store_pointer, real_time_hours      ;locate current time in memory
```

Software - Code Strategy

The larger block RAM provided with Spartan-3 devices means that KCPSM3 can execute programs up to 4 times longer than with the original KCPSM PicoBlaze in a Spartan-II(E) device. Longer programs can naturally lead to more complexity as they implement more functionality. It is also likely that more variables and data will be required by such applications and this was the main motivation for including the 64-bytes of Scratch Pad Memory in KCPSM3.

The introduction of scratch pad memory within a PicoBlaze has tended to change the overall coding strategy. Although some registers are still reserved for special tasks using the NAMEREG directive, the majority of variables and data is now stored in the scratch pad memory and accessed and manipulated using 'working registers'. Obviously there is both a code and performance overhead to this technique but the flexibility and portability of code is highly desirable.

Allocating Registers

The 'uclock.psm' program allocates registers using NAMEREG for the following reasons:-

a) Time critical access - 'int_counter_lsb' and 'int_counter_msb' need to be updated quickly during an ISR.
b) Code independence - The ISR uses only 'int_counter_lsb' and 'int_counter_msb' and guarantees that no other variables are modified during this event that could occur at any time.
c) Frequently used - 'UART_data' is used in many parts of the program and makes the code more understandable.
d) Code interaction and portability - 'store_pointer' is used to pass the location of information in scratch pad memory to subroutines.

Allocating Scratch Pad Memory

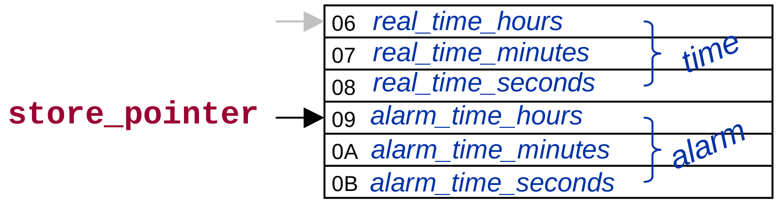
The scratch pad memory is then used for the majority of data and variables associated with the clock application. Clearly there is far more information than could be held within registers alone. The memory is used in 3 different ways by the 'uclock' program:-

a) Non time critical variables - The variables defining the real time and alarm time.
b) Temporary storage - The 'update_time' subroutine requires several 'working registers' and is invoked from various places within the program. The 'time_preserve' locations are used to preserve the contents of those 'working registers' during execution and then restore them on completion.
c) Data Buffer - The 'string_start' constant identifies the beginning of a memory block used to buffer text strings received from, or to be transmitted to the UART.

Software - Code Strategy

The register and scratch pad memory strategy is nicely demonstrated at various points within the ‘uclock’ program. The ‘time_to_ASCII’ sub routine takes the values for hours, minutes and seconds and converts them into a an ASCII text string ready to be transmitted on the UART. The code excerpts below show how the strategy is applied with this routine…..

There are currently 2 time values which need to be transmitted, the real time and the alarm time. Rather than have a separate sub routine for each, the same routine can be used by passing a pointer to the appropriate time value using a named register.



Notice how the named constants make all this code easier to understand.

```
      transmit_time: LOAD store_pointer, real_time_hours      ;locate current time in memory
                     CALL time_to_ASCII
```

```
transmit_alarm_time: LOAD store_pointer, alarm_time_hours     ;locate alarm time in memory
                     CALL time_to_ASCII
```

```
FETCH s0, (store_pointer)   ;read hours value
CALL decimal_to_ASCII       ;convert to ASCII
```

```
ADD store_pointer, 01       ;move to minutes
FETCH s0, (store_pointer)   ;read minutes value
CALL decimal_to_ASCII       ;convert to ASCII
```

```
ADD store_pointer, 01       ;move to seconds
FETCH s0, (store_pointer)   ;read seconds value
CALL decimal_to_ASCII       ;convert to ASCII
```

Within the ‘time_to_ASCII’ sub routine, the time values are accessed from the memory by incrementing the pointer address supplied. Although this increases the code length slightly compared with directly accessing each location with a constant address (i.e. FETCH s0,real_time_seconds), the overall program code is more compact due to the complete reuse of the sub routine.

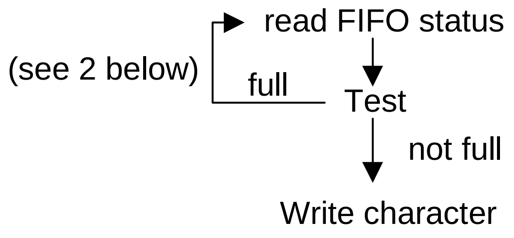
Software - UART Transmit

At the lowest level of the code, the software must communicate with the hardware. With CONSTANT directives defining the actual port addresses to be used, and in some cases, the meanings of specific bits, this code can be much more pleasant to write. Obviously the code which interacts with hardware often needs to consider how the external logic behaves in order to operate efficiently as well as correctly.

```
LOAD UART_data, character_K
CALL send_to_UART
```

The ‘uclock’ program includes a simple sub routine to send a byte of data (or character) to the UART transmitter. The named register ‘UART_data’ is used to pass data to the sub routine.

All this routine really needs to do is to write the data contained in the ‘UART_data’ register to the appropriate port which has been named ‘UART_write_port’. This is achieved with just one OUTPUT instruction. However, the ‘uart_tx’ macro contains a FIFO buffer, and it is therefore wise to determine if there is space in the buffer to accept the new character before attempting to write it.



```
send_to_UART: INPUT s0, UART_status_port
              TEST s0, tx_full
              JUMP Z, UART_write
              CALL update_time
              JUMP send_to_UART
  UART_write: OUTPUT UART_data, UART_write_port
              RETURN
```

‘tx_full’ constant to test correct bit

<u>What if the FIFO buffer really is full?</u>

The program will need to wait until there is space, which could be as long as it takes the UART to transmit one whole character (260µs at 38400 BAUD). This could be an undesirable situation if it prevents other processing from being performed.
‘uclock’ demonstrates 2 solutions to this issue….

1) Interrupts are used to ensure that the 1µs reference time pulses are never missed.
2) The subroutine includes a call to the ‘update_time’ subroutine each time it encounters the full condition. This routine executes in under 4 µs (55MHz clock) which is insignificant to the UART operation but ensures that the real time is accurate and the alarm will still be activated. Your programs could include a call to a similar subroutine.

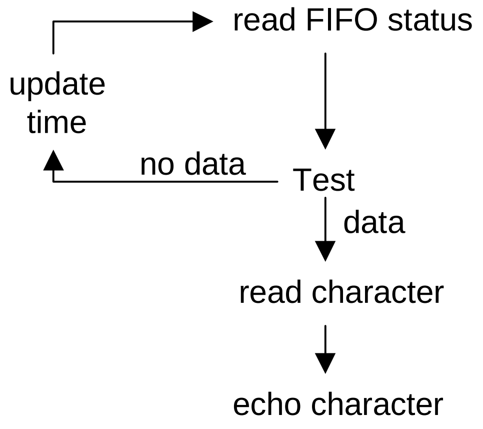
Software - UART Receive

Not surprisingly, the 'read_from_UART' routine has to deal with very similar issues to the 'send_to_UART' routine. In this case the program may need to wait indefinitely for information to be available to read from the FIFO buffer contained in the 'uart_rx' macro and it it therefore critical that the 'uclock' program uses interrupts and an embedded call to the 'update_time' routine to ensure that everything doesn't come grinding to a halt!

```
CALL read_from_UART     ;obtain and echo character
STORE UART_data, (s1)
```

The character data obtained from the UART receiver will be passed back from the routine using the 'UART_data' register assigned for the purpose.

Once a character has been obtained, the routine immediately calls 'send_to_UART' to transmit back the same character to the monitor which performs a 'remote echo' function.



```
read_from_UART: INPUT s0, UART_status_port
                TEST s0, rx_data_present
                JUMP NZ, read_character
                CALL update_time
                JUMP read_from_UART
read_character: INPUT UART_data, UART_read_port
                CALL send_to_UART
                RETURN
```

'**rx_data_present**'
constant to test correct bit

This routine does not consider the situation in which the receiver FIFO becomes full with the associated risk of data being lost. 'uclock' handles this an an exception error elsewhere in the code. You may decide to treat this as a further cause for interrupt with associated changes to the hardware.

Software - Interrupt

The 'uclock' program really doesn't have to do very much when it receives an interrupt but it is absolutely critical to the clock application because the pulses occurring every 1µs are the fundamental timing reference. The code illustrates the general configuration required for the software to behave correctly.

```
ENABLE INTERRUPT     ;enable the 1us interrupts
```

Interrupts must be enabled to work. In 'uclock', this occurs just after initialisation of registers and memory.

```
;Interrupt vector
;
ADDRESS 3FF
JUMP ISR
```

The interrupt will invoke a 'CALL 3FF' operation requiring an appropriate JUMP instruction to be placed at the most significant address of the program memory. The ADDRESS directive ensures this JUMP 'vector' is located it this position.

By their very nature, interrupts can occur at any time relative to the execution of the main program code. It is vital that any actions which take place during the execution of the Interrupt Service Routine (ISR) do not have an adverse effect on the main program. This is where it is useful to reserve some registers purely for use by the ISR.

```
     ADDRESS 3FC
ISR: ADD int_counter_lsb, 01
     ADDCY int_counter_msb, 00
     RETURNI ENABLE
```

Each interrupt is used to increment a 16-bit counter formed by two assigned registers. This counter allows up to 65,536µs of real time to be recorded. The ISR can be located anywhere in the program memory, but in 'uclock' it has been forced to assemble in the locations adjacent to the interrupt vector purely to keep associated code together. The ISR ends with the RETURNI instruction which prepares KCPSM3 to capture the next pulse.

The first part of the process to 'update_time' requires the current value of the interrupt counter (µs) to be captured. Because the counter is formed by two registers, this capture requires two (STORE) instructions. It is possible that in interrupt could occur during this capture such that the LS-Byte and MS-Byte values do not correspond to the same value (i.e. count value 3AFF could be captured as 3BFF as the counter increments to 3B00). To prevent this, interrupts are temporarily disabled. The hardware ensures that an interrupt pulse will not be missed during this brief period.

```
DISABLE INTERRUPT
STORE int_counter_lsb, us_time_stamp_lsb
STORE int_counter_msb, us_time_stamp_msb
ENABLE INTERRUPT
```

Software - ASCII codes

It is not the intention of this document to explain every line of the 'uclock' program. However, some of the code may be directly applicable to the applications as well as providing worthwhile examples of KCPSM3 assembler. The following routines are used to manipulate ASCII codes.

Lower to Upper Case Conversion

A quick review of the ASCII codes reveals that the letters 'A' to 'Z' and 'a' to 'z' appear in blocks. Close inspection reveals that these codes are identical except for bit 5 which effectively identifies the letter as upper or lower case. The 'upper_case' subroutine determines if the character contained in register 's0' is in the range 61 to 7A signifying a lower case letter and then converts it to upper case.

```
'A' = 41 = 0100 0001
'Z' = 5A = 0101 1010

'a' = 61 = 0110 0001
'z' = 7A = 0111 1010

      DF = 1101 1111
```

```
upper_case: COMPARE s0, 61
            RETURN C
            COMPARE s0, 7B
            RETURN NC
            AND s0, DF
            RETURN
```

Equivalent to 's0-61', so values *below* 61 will result in a carry which will reject all character codes below the letter 'a'.

Equivalent to 's0-7B', so values *above* 7A will not produce a carry which will reject all character codes above the letter 'z'.

Forces bit 5 to be low

ASCII number to value Conversion

```
'0' = 30 = 0011 0000
'9' = 39 = 0011 1001
```

The ASCII codes for numbers '0' to '9' are conveniently arranges in a block such that the 4 least significant bits directly correspond to the numerical value. This means that the conversion process simply requires that the 4 most significant bits need to be masked to zero. The '1char_to_value' subroutine performs this simple conversion but includes the test for character codes in the correct range using the carry flag to indicate failure.

```
1char_to_value: ADD s0, C6
                RETURN C
                SUB s0, F6
                RETURN
```

30 + C6 = F6
39 + C6 = FF

Character codes greater than 39 will result in a sum greater than FF and generate a carry.

F6 - F6 = 00
FF - F6 = 09

The character codes which were less than 30 will now result in an underflow with associated carry flag. Valid characters have correct numerical value.

Software - String Input

The clock receives commands from the UART link as ASCII characters. To make the interface more human friendly, the commands can be entered and modified using the backspace key before confirming entry using the carriage return. KCPSM3 performs the task of controlling the display during command entry and forms the ASCII string in scratch pad memory.

The 'receive_string' sub routine commences with the definition of the scratch pad memory locations. The first location is defined as a constant, but the length of the string has been included in the code and is used to compute the last location.

```
   receive_string: LOAD s1, string_start        ;locate start of string
                   LOAD s2, s1                  ;compute 16 character address
                   ADD s2, 10
```

The receiver FIFO buffer is tested to see if it has become full. If this is the case, then it is assumed that characters have been missed and an 'Overflow_Error' message is transmitted and all the existing data in the FIFO is discarded. It is never easy to decide the best way to deal with an unexpected situation, but assembler code in KCPSM makes these situations relatively easy to identify and adapt to.

```
receive_full_test: INPUT s0, UART_status_port   ;test Rx_FIFO buffer for full
                   TEST s0, rx_full
                   JUMP NZ, read_error
```

The routine is then ready to enter the normal sequence of events with register 's1' acting as a pointer to the scratch pad memory. Each character is read using the 'read_from_UART' subroutine described earlier which also echoes the received character to the UART transmitter to control the display and provide feedback. The received character is then stored in the scratchpad memory.

```
                   CALL read_from_UART          ;obtain and echo character
                   STORE UART_data, (s1)        ;write to memory
```

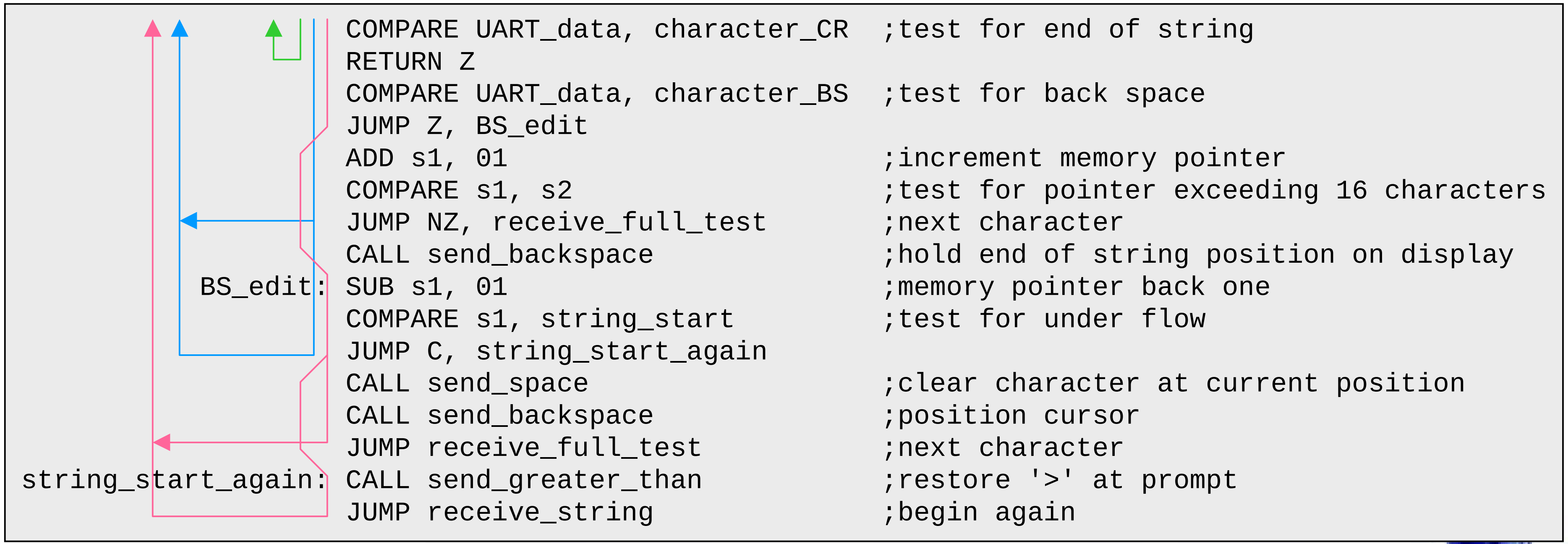
Software - String Input

After each character is received and stored, it is tested for ‘carriage return’ or ‘backspace’ so that the appropriate action can be performed.

Carriage Return - This signifies the end of the string and the routine is complete.

Normal character - Increments the memory pointer ready for the next character. It must also check to see if this is the maximum length for a string, in which case, it decrements the memory pointer and transmits a backspace so that the monitor indicates that no more characters can be entered (the last character just gets over written on the display).

Backspace - Decrements the the memory pointer so that the backspace just stored is ignored and the previous character will be overwritten by the next input. The echoed character has already controlled the display cursor position, but in order to look like a delete function, a space is transmitted to clear the displayed character. This in turn requires another backspace to move the cursor back. A test is also performed to ensure that a backspace can not move back into the command prompt.



```
                    COMPARE UART_data, character_CR  ;test for end of string
                    RETURN Z
                    COMPARE UART_data, character_BS  ;test for back space
                    JUMP Z, BS_edit
                    ADD s1, 01                       ;increment memory pointer
                    COMPARE s1, s2                   ;test for pointer exceeding 16 characters
                    JUMP NZ, receive_full_test       ;next character
                    CALL send_backspace              ;hold end of string position on display
          BS_edit:  SUB s1, 01                       ;memory pointer back one
                    COMPARE s1, string_start         ;test for under flow
                    JUMP C, string_start_again
                    CALL send_space                  ;clear character at current position
                    CALL send_backspace              ;position cursor
                    JUMP receive_full_test           ;next character
string_start_again: CALL send_greater_than           ;restore '>' at prompt
                    JUMP receive_string              ;begin again
```

Software - String Output

This routine simply transmits an ASCII string held in scratch pad memory to the UART transmitter. The characters are transmitted one at a time using the ‘send_to_UART’ routine described previously.

The start of the string is defined by a constant and the end of the string is identified by the carriage return character (which is also transmitted). The register ‘s1’ is used as a memory pointer in this routine.

```
transmit_string: LOAD s1, string_start                ;locate start of string
   next_char_tx: FETCH UART_data, (s1)                ;read character from memory
                 CALL send_to_UART                    ;transmit character
                 COMPARE UART_data, character_CR      ;test for last character
                 RETURN Z
                 ADD s1, 01                           ;move to next character
                 JUMP next_char_tx
```

The execution speed of this subroutine will depending on the length of the string and the available space in the UART transmitter FIFO. The ‘send_to_UART’ routine waits for space to be available for each character.

Software - Clock

Although the main application focus of this reference design provides a real time clock, the code for this is actually of less interest to analyze and would be less likely to be used in another application. For this reason, only small sections of code are described to make some useful observations.

<u>Transparent Processing</u>

The 'update_time' routine is a significant section of code which is called from various points in the program. The routine requires several 'working registers'. Since it would become difficult to manage the use of these registers when dealing with other parts of the program, the routine is made effectively 'transparent' by preserving the contents of all the used registers in scratch pad memory during the routine.

The store and fetch instructions clearly take some code space and require time to execute, but this technique will make applications very portable and safe to use. This would be particularly useful for interrupt service routines requiring significant processing.

```
  update_time: STORE s0, time_preserve0
               STORE s1, time_preserve1
               STORE s2, time_preserve2
               STORE s3, time_preserve3
               STORE s4, time_preserve4
               STORE s5, time_preserve5
```

S0 to s5 free to be used

```
finish_update: FETCH s0, time_preserve0
               FETCH s1, time_preserve1
               FETCH s2, time_preserve2
               FETCH s3, time_preserve3
               FETCH s4, time_preserve4
               FETCH s5, time_preserve5
               RETURN
```

<u>16-bit operations</u>

The 'µs' and 'ms' values of real time are handled as 16-bit quantities. The ability to include the CARRY flag in arithmetic operations makes it possible to perform the 16-bit calculations that are required. In each case the LS-Bytes are processed first (without carry) and the carry incorporated into the processing of the MS-Bytes.

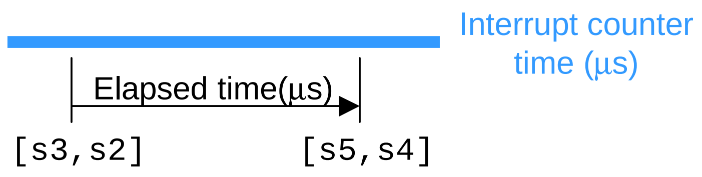
```
SUB s4, s2
SUBCY s5, s3
FETCH s2, us_time_lsb
FETCH s3, us_time_msb
ADD s2, s4
ADDCY s3, s5
```

Calculate number of 'µs' elapsed since last update. This could be up to 65,536µs.

[s5,s4] = [s5,s4] - [s3,s2]

Add elapsed 'µs' to the real time total of 'µs'.

[s3,s2] = [s3,s2] + [s5,s4]



Software - Clock

The seconds, minutes, and hours operation is just a classic example of assembler programming. This section of code begins with the value ‘00’ or ‘01’ in the ‘s0’ register indicating the requirement to add one second of real time.

```
             FETCH s1, real_time_seconds
             ADD s1, s0
             COMPARE s1, seconds_in_a_minute
             JUMP Z, inc_minutes
             STORE s1, real_time_seconds
             JUMP time_update_complete
inc_minutes: LOAD s1, 00
             STORE s1, real_time_seconds
             FETCH s1, real_time_minutes
             ADD s1, 01
             COMPARE s1, minutes_in_an_hour
             JUMP Z, inc_hours
             STORE s1, real_time_minutes
             JUMP time_update_complete
  inc_hours: LOAD s1, 00
             STORE s1, real_time_minutes
             FETCH s1, real_time_hours
             ADD s1, 01
             COMPARE s1, hours_in_a_day
             JUMP Z, reset_hours
             STORE s1, real_time_hours
             JUMP time_update_complete
reset_hours: LOAD s1, 00
             STORE s1, real_time_hours
```

Value Constants make the code easier to read and understand and are used here in the COMPARE instructions which actually perform a subtraction, discard the result, but still set the flags. Equal values will therefore cause the ZERO flag to be set.

Scratch pad memory and location constants.
The memory provided the main variable storage for the real time clock application and the registers are only used on a temporary basis. The use of location constants makes the meaning of each location obvious leading to very readable code which is portable.

Line labels are so easy to use and again help readability of the code. It is most unusual to use absolute address values in any JUMP or CALL instruction. Whilst it is useful to use descriptive labels, try to keep them brief to help formatting.

Note that the actual code includes comments. These are always worth adding to your code to increase the understandability of the code especially when you return to it several weeks or months later. However, it is advisable to keep comments brief otherwise log files can become difficult to print.

Future Plans and Exercises

Future Plans

Since the main purpose of this package was as a reference design, there is no commitment to further enhance this as a design. However, if user feedback indicates that this is a useful 'IP core' in itself, there may be further development. At this time, it is left to the reader to modify this reference design to provide any required functionality. Your feedback will be very useful in improving the reference design for both educational and IP requirements.

Exercises

It is suggested that the addition of the following features would provide suitable self training opportunities.

Modify the existing design to…..

1) Operate at a baud rate of 9600 - Hardware change only.
2) Provide an interrupt at 10µs intervals - Hardware and software changes (ISR).
3) Make the alarm condition transmit a message to the terminal - Software change only.

Start/Stop Timer

Similar to setting a home VCR or central heating controller, provide the ability to program the clock with a start time and an end time which results in an output pin being turned 'on' during the specified times.
Hardware - Add an additional output port.
Software - interpret new command string and implement another time value in scratch pad memory.

Day/Date support

Add support for day name and full date. My be you even know the way to compute the day from the date!
Software - New command strings, messages and values stored in scratch pad memory.

KCPSM3 Resource Charts

Scratch Pad Memory

00		10		20		30	
01		11		21		31	
02		12		22		32	
03		13		23		33	
04		14		24		34	
05		15		25		35	
06		16		26		36	
07		17		27		37	
08		18		28		38	
09		19		29		39	
0A		1A		2A		3A	
0B		1B		2B		3B	
0C		1C		2C		3C	
0D		1D		2D		3D	
0E		1E		2E		3E	
0F		1F		2F		3F	

Register Names

s0	
s1	
s2	
s3	
s4	
s5	
s6	
s7	
s8	
s9	
sA	
sB	
sC	
sD	
sE	
sF	

Constants Values
