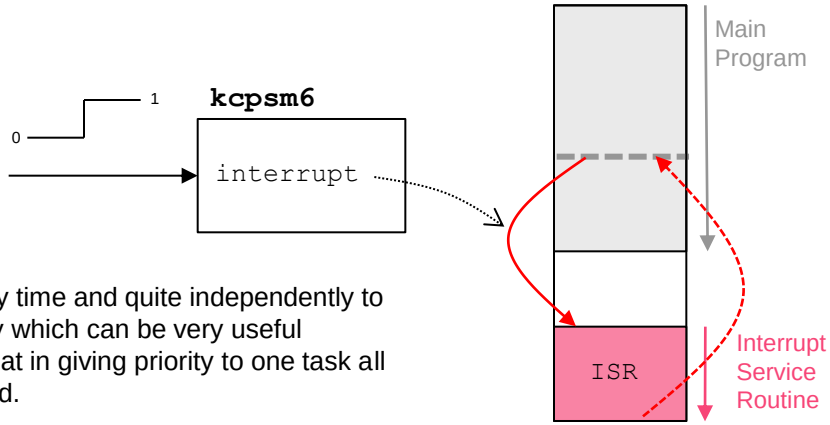
# Interrupts

Interrupts can be extremely useful so KCPSM6 provides an ‘interrupt’ input pin, an ‘interrupt_ack’ output pin, an optional ‘interrrupt_vector’ generic and three interrupt related instructions. However, it would be fair to say that interrupts are quite an advanced technique and require understanding, thought and preparation to be used wisely and successfully. This subject is made more interesting because each KCPSM6 is fully embedded into your FPGA design meaning that you have the option to define hardware dedicated to servicing tasks in a way that simply isn’t available when using a standard microcontroller device. In fact, many PicoBlaze users have discovered that because each PicoBlaze is so small and efficient, it is often beneficial to use multiple instances within the same design in order that each is dedicated to a particular task and therefore avoiding the requirements for interrupts altogether. So it is well worth considering what an interrupt actually does and when it provides greatest benefit in a KCPSM6 design.

What does an interrupt do?

To state the obvious, an interrupt is used to interrupt the normal program execution sequence of KCPSM6. This means that when the ‘interrupt’ input is driven High (‘1’), it will force KCPSM6 to abandon the code that it is executing, save its current operational state and divert its attention to executing a special section of program code known as an Interrupt Service Routine (ISR). Once the interrupt has been serviced, KCPSM6 returns to the program at the point from which it was interrupted and restores the operational states so that it can resume execution of the program as if nothing had happened.




Hence the interrupt mechanism provides a way for KCPSM6 to react to an event at any time and quite independently to the main tasks being performed. In other words an interrupt is given the highest priority which can be very useful particularly when reacting to a critical situation. However, it must also be recognised that in giving priority to one task all other tasks can be interrupted and hence their execution rates can be erratic or delayed.

When do interrupts make sense?

The key observation is that an interrupt has the highest priority. So clearly the most obvious application for an interrupt is to react quickly to system critical or emergency situations. These may be such rare events that they may never happen in normal operation such as the detection of a fire and the need to activate the water sprinklers. More common situations are less of an emergency but important to system integrity with a good example being the requirement to react to a FIFO buffer becoming full so that data is read from it before it actually overflows and data is lost.

Another application involves a regular or semi-regular steam of interrupts to KCPSM6 which become a fundamental part of the way in which the program normally operates. For example a hardware counter could easily generate an interrupt every milli-second which KCPSM6 uses as the reference for an accurate real time clock. The main program possibly enabling that clock to be set, displayed and for controlling the times at which appliances must be turned on and off. Alternatively KCPSM6 may use each interrupt as the trigger to perform a sequence of tasks but do virtually nothing else whilst waiting.

# Interrupts

When are interrupts NOT suitable?

To answer this question their are two important observations. First is that whilst KCPSM6 is servicing an interrupt, it is not making any progress executing the main program (i.e. the main program has been interrupted!). Secondly, KCPSM6 can only service one interrupt at a time which means that if another interrupt occurs whilst KCPSM6 is busy executing the ISR then that new interrupt will either be missed or will have to wait neither of which is ideal. In general terms, an interrupt scheme in not suitable if the rate at which interrupts occur is too fast for them to be serviced and for the main program to make adequate progress. Clearly the definition of 'too fast' depends on how demanding both the main program and the ISR are but the one absolute constant is that every KCPSM6 instruction always takes 2 clock cycles to execute. So at least you can easily determine the code execution rate for a given clock frequency and compare that with the demands of your program and your expected interrupt rate.

For example, consider the use of interrupts generated at 1ms intervals for use as a time reference for a real time clock. With a KCPSM6 operating at a clock frequency of 66MHz it will execute 33,000,000 instructions per second and therefore it will be able to execute 33,000 instructions between each interrupt. This is clearly a large number and most unlikely to impede the ability to make good progress through any program whilst always being ready to service the next interrupt. But suppose the interrupts are generated at 1µs intervals with the aim of achieving finer timing resolution. Now KCPSM6 would only be able to execute 33 instructions between each interrupt (i.e. Less instructions that you can print out on one side of a piece of paper!). Unless the ISR is very brief it will not complete in time. Even if the ISR was only 12 instructions it would mean that over a third of the computing power was consumed servicing the simple ISR and that means that the main program would execute proportionally slower with an associated 'hesitancy' caused by the continuous interruptions. This may still be acceptable for the application but it is certainly on the verge of being unsuitable and will make it very difficult to expand the features implemented by the program code.

What are the alternatives?

When interrupts make sense then it a very useful feature of KCPSM6 to exploit. However, when they are not suitable the benefit of using a Xilinx FPGA is that there are very good alternatives. The biggest mistake people often make it to battle with interrupt based solutions when they are not suitable. It is much better to exploit alternative solutions to make the overall design much easier to implement.

Increased use of hardware – Quite simply circuits are implement which perform what would have been achieved by the software based ISR such that interrupts are avoided or their rate greatly reduced. For example a hardware based counter/timer block can be very simple to implement in hardware and then KCPSM6 can read time values from it when it needs to. The complexity of a real time clock could still be implemented in software but the timing resolution is best handled by the naturally fast hardware. Interrupts could then be used occasionally when a hardware comparator matches a time value set by KCPSM6.

Divide and conquer! – If a KCPSM6 processor is 100% dedicated to a task then really it is always performing an ISR. This makes sense if the ISR is relatively complex to consider implementing in hardware. With KCPSM6 being so small (26 Slices) dedicating a different processor to each demanding task can often be the easiest and best solution. Indeed, PicoBlaze is often used to service interrupts for a larger processor such as MicroBlaze.

# ‘interrupt_vector’ and ‘ADDRESS’ Directive



KCPSM6_User_Guide_30Sept14

See pages 83-85 for interrupt related instructions

When KCPSM6 responds to an interrupt it executes the equivalent of a CALL instruction as well as the interrupt specific tasks such as preserving the states of the flags. The interrupt vector is the address that KCPSM6 effectively calls and it has the default value of 3FF hex. However, this can be set to any value within the range of the program memory available in your design using the ‘interrupt_vector’ generic in your HDL design description.

```
processor: kcpsm6
  generic map (                 hwbuild => X"00",
                       interrupt_vector => X"3FF",
                scratch_pad_memory_size => 64)
  port map(                     address => address,
                            instruction => instruction,
Etc...
```

Component declaration (part of) showing the default values of the three generics.

Component Instantiation (part of) showing that the interrupt vector has been set to ‘F80’ hex.

```
processor: kcpsm6
  generic map (                 hwbuild => X"41",
                       interrupt_vector => X"F80",
                scratch_pad_memory_size => 256)
  port map(      address => address,
              instruction => instruction,
Etc...
```

PSM file...

```
        ADDRESS F80
        ;
ISR :   ADD sF, 1'd
        RETURNI ENABLE
```

**ADDRESS Directive**

Use the ADDRESS directive in your PSM code to force the ISR to be assembled starting at the same address as the interrupt vector.

What is a good address for ‘interrupt_vector’?

3FF is the last location in a 1K program memory and is consistent with KCPSM, KCPSM-II and KCPSM3. So for direct compatibility with legacy PicoBlaze programs this is the best address to start with and hence the reason why it is the default. Of course you could modify the program and vector.

Generally the most convenient address is somewhere *close to the end* of the program memory available but leaving enough space for the ISR. This means that the ISR can begin servicing the interrupt immediately. It is also convenient from a programming perspective because the ADDRESS directive must be used to align the start of the ISR code with interrupt vector and having this as the last section of your PSM program allows your main program the flexibility to expand up to it. As your code becomes stable you can always fine tune your matching ‘interrupt_vector’ and ADDRESS directive for best memory fit.

What are bad values? If you try to put your ISR somewhere in the middle of your program then you will probably find that you are always having to adjust the ADDRESS directive and ‘interrupt_vector’ which is just an inconvenient waste of time as well as error prone. The absolute worst address would be zero! Under no circumstances would you want your ISR to execute on power up or following a reset (RETURNI should only be used following an interrupt).

# Hardware arrangements for KCPSM6 Interrupt

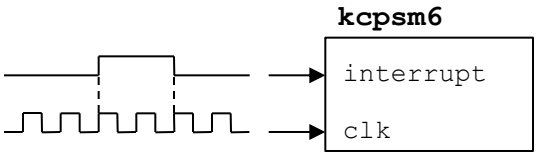


See pages 83-85 for interrupt related instructions

The KCPSM6 processor has two pins dedicated to interrupts; an 'interrupt' input and an 'interrupt_ack' output. To initiate an interrupt the 'interrupt' input must be driven High and the fundamental interrupt response time is just 3 or 4 clock cycles. As shown on the next page (Interrupt Waveforms) the interrupt input is sampled once every two clock cycles consistent with the instruction execution rate. For this reason it is vital that the interrupt input is High at the right time to be observed by KCPSM6 and the easiest way to achieve that us to drive the interrupt input High for longer than one clock cycle. There are two fundamental schemes that can be used which can really be describes as being 'open-loop' and 'closed loop'.

'Open-Loop' interrupt pulse

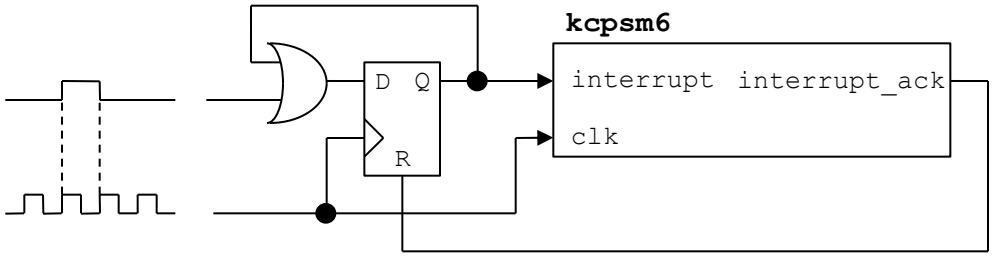
The simplest way of initiating an interrupt is to generate an active High pulse that has a duration of 2 clock cycles. The pulse can be longer but should have returned Low before the ISR completes otherwise KCPSM6 will immediate think there is another interrupt to service (remember that each instruction executes in 2 clock cycles so some ISR's may not take very many clock cycles). Once KCPSM6 observes the High level on its interrupt input it will abandon the next instruction and immediately move to the ISR.




The simplicity of the 'open-loop' method is obvious but it must also be recognised that any open loop system has its limitations. In this case there is the potential for KCPSM6 to miss an interrupt request and therefore fail to service it. This could happen if the KCPSM6 program has deliberately disabled interrupts or is already servicing a previous ISR. KCPSM6 will also ignore the interrupt input whilst held in sleep mode. Therefore this technique should only be used if you can predict that KCPSM6 will always be ready to respond to an interrupt request or if it is acceptable for interrupts to be missed.

'Closed-Loop' interrupt (recommended)

In this scheme your design drives the interrupt signal High to request an interrupt and then keeps driving it High until KCPSM6 generates an 'interrupt_ack' pulse confirming that it has seen it. This ensures that the interrupt will always be observed by KCPSM6 when it is able to. If interrupts have been temporarily disabled deliberately, or whilst servicing a previous interrupt, then the response will be delayed but the event can not be missed. Likewise, if KCPSM6 is held in sleep mode when the interrupt is requested it will remain active until KCPSM6 is allowed to wake up and observe it.




Hint – Some systems can require a more comprehensive closed-loop arrangement in which KCPSM6 would be expected to indicate when the ISR has completed rather than just started (which is what 'interrupt_ack' signifies). This can be achieved using an output port with associated 'OUTPUT' or 'OUTPUTK' instructions at the end of your ISR. Alternatively you could detect when instruction[17:12] = "101001" corresponding with the 'RETURNI' instruction being fetched from the program memory.

# Interrupt Waveforms





See pages 83-85 for interrupt related instructions

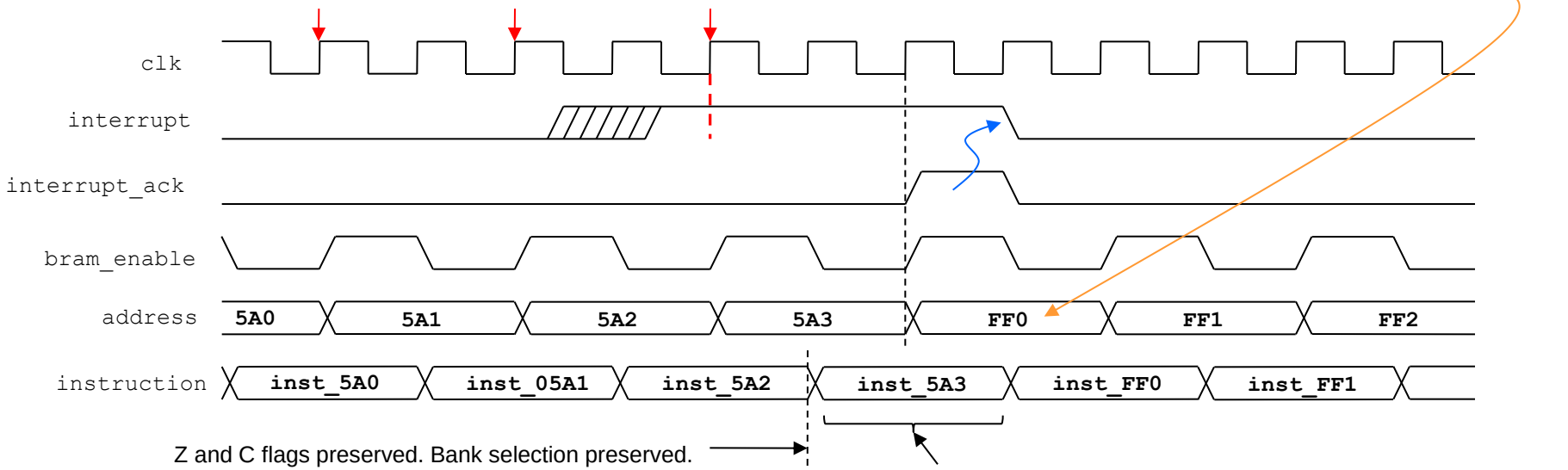
An interrupt is performed when the 'interrupt' input is driven High, interrupts have been enabled by the program and KCPSM6 is not in sleep mode or otherwise busy servicing a previous interrupt. When KCPSM6 detects an interrupt it forces the next instruction to be abandoned, preserves the current states of the 'Z' and 'C' flags, notes the current bank selection ('A' or 'B') and then forces the program counter to the interrupt vector (default value is 3FF hex which is the last location of a 1K program memory but can be set to any value using the 'interrupt_vector' generic).

The waveforms shown below illustrate a normal response to an interrupt when interrupts have been enabled within the program and KCPSM6 is ready to respond. In the hardware design the interrupt vector was set to FF0 hex and a 'closed-loop' interrupt scheme used (implemented by the VHDL shown on the right) to ensure that the interrupt pulse can not be missed.

```
interrupt_control: process(clk)
  begin
    if clk'event and clk = '1' then
      if interrupt_ack = '1' then
        interrupt <= '0';
       else
        if kcpsm6_interrupt = '1' then
          interrupt <= '1';
         else
          interrupt <= interrupt;
        end if;
      end if;
    end if;
end process interrupt_control;
```

```
interrupt_vector => X"FF0",
```

The 'interrupt' input is sampled on the rising clock edges that the address

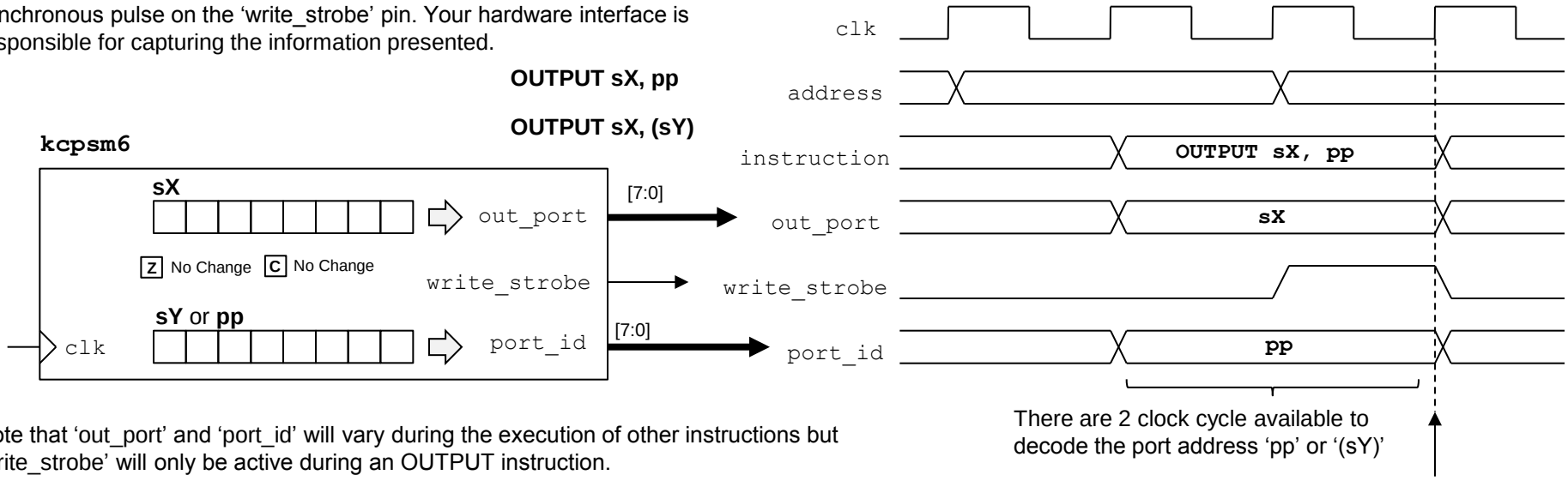



Z and C flags preserved. Bank selection preserved. All will be restored by the RETURNI instruction.

The last instruction read from program memory before the interrupt takes place is abandoned. This will be the first instruction executed following a RETURNI after the interrupt has been serviced.

# OUTPUT sX, pp
# OUTPUT sX, (sY)

An 'OUTPUT' instruction is used to transfer information from a register 'sX' to a general purpose output port specified by an 8-bit constant value 'pp' or the contents of another register '(sY)'. KCPSM6 presents the contents of the register 'sX' on 'out_port' and the port address defined by 'pp' or '(sY)' is presented on 'port_id'. Both pieces of information are qualified by an active High ('1') synchronous pulse on the 'write_strobe' pin. Your hardware interface is responsible for capturing the information presented.




Note that 'out_port' and 'port_id' will vary during the execution of other instructions but 'write_strobe' will only be active during an OUTPUT instruction.

Hint – In most cases a fixed port address 'pp' is used so CONSTANT directives provide an ideal why track your port assignments and make your code easier to write, understand and maintain.

Examples

```
CONSTANT LED_port, 05
;
LOAD s3, 3A
OUTPUT s3, LED_port
```

If you want to keep your designs small and fast then assign port addresses that facilitate smaller logic functions.

In this example a set of 8 LEDs are mapped to port 05 hex and only 3-bits of 'port_id' together with 'write_strobe' are decoded.

VHDL

```
if clk'event and clk = '1' then
  if write_strobe = '1' then
    if port_id(2 downto 0) = "101" then
      led <= out_port;
    end if;
  end if;
end if;
```

```
OUTPUT s6, (s2)
OUTPUT s4, 40
OUTPUT sB, 64'd
```

Decimal values can be used to specify port addresses but hex or binary values are normally easier to work with when defining the hardware.

# Constant-Optimised Output Ports

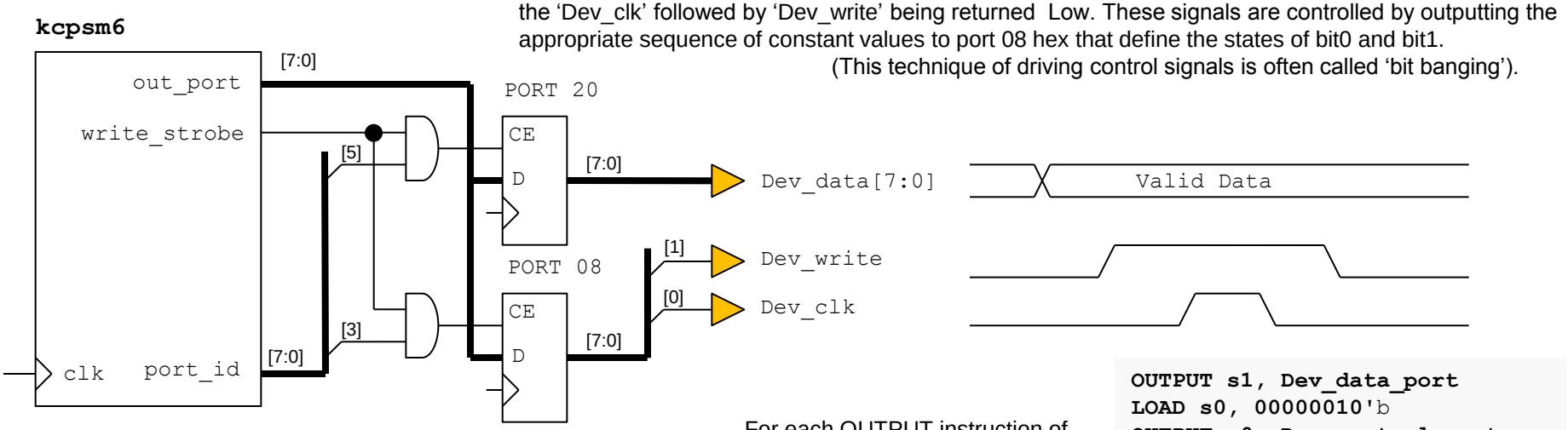


In order to understand the motive for the constant-optimised ports and to know when it is better to use them, it is necessary to appreciate the situations in which the general output ports can adversely effect the size of your program code and/or result in lower performance. The 'OUTPUT sX, pp' and 'OUTPUT sX, (sY)' instructions associated with the general purpose output ports both require that the value to be written to the port to be held in a register 'sX'. This is ideal when the value is a variable in your system but when you want to send a constant value, or more likely, a series of constant values to a port the act of loading 'sX' each time increases code size and reduces performance. In many applications this overhead can be tolerated and you should feel no pressure to adapt your design and code to use the constant-optimised ports unless you really want to. However, using constant-optimised ports appropriately can make code easier to write and avoid the code size and performance overhead associated with general purpose output ports when necessary.

**Using General Purpose Output ports……**

In this example KCPSM6 is required to write 8-bit data to an external device. The data is naturally variable and is presented to the device interface by outputting to port 20 hex. Then KCPSM6 is required to generate the correct sequence of control signals; 'Dev_write' is set High before a pulse is generated on the 'Dev_clk' followed by 'Dev_write' being returned Low. These signals are controlled by outputting the appropriate sequence of constant values to port 08 hex that define the states of bit0 and bit1.

(This technique of driving control signals is often called 'bit banging').



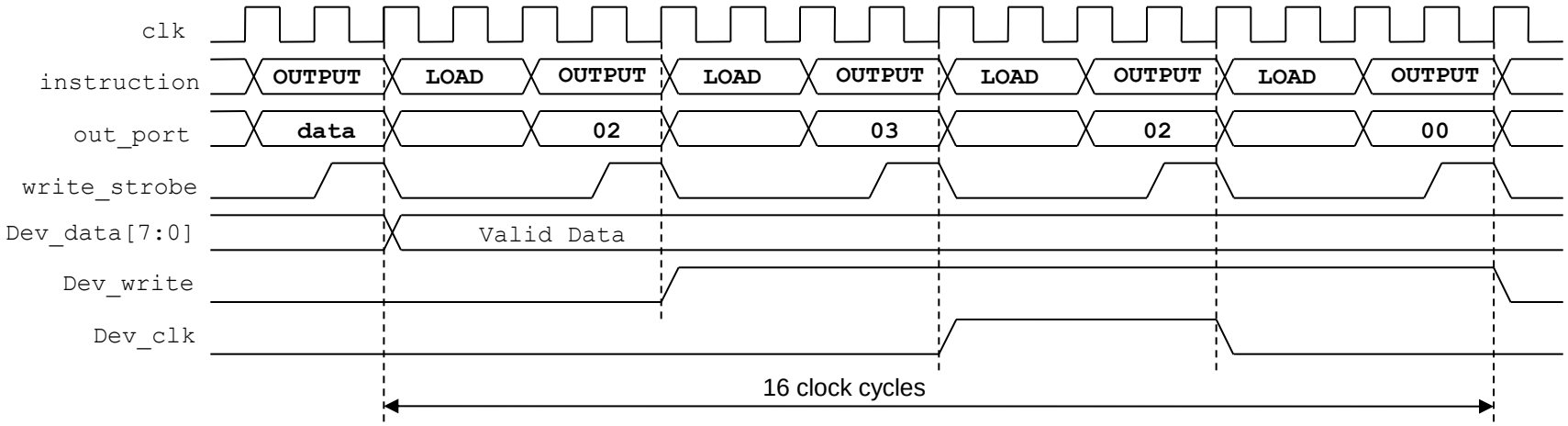

```
CONSTANT Dev_data_port, 20
CONSTANT Dev_control_port, 08
```

For each OUTPUT instruction of the control sequence waveform there is a corresponding LOAD instruction that prepares 's0' with the required constant value.

```
OUTPUT s1, Dev_data_port
LOAD s0, 00000010'b
OUTPUT s0, Dev_control_port
LOAD s0, 00000011'b
OUTPUT s0, Dev_control_port
LOAD s0, 00000010'b
OUTPUT s0, Dev_control_port
LOAD s0, 00
OUTPUT s0, Dev_control_port
```

# Constant-Optimised Output Ports

The timing diagram for the code using the general purpose output ports shows that it takes 16 system clock cycles to generate the control sequence because every instruction takes 2 clock cycles and every OUTPUT instruction requires a corresponding LOAD instruction to initialise 'sX' ('s0' was used in the example). It can also be seen that this results in 4 clock cycles between each transition of the control sequence.




There are a number of applications where it is beneficial that KPCSM6 slows down the generation of waveforms. For example, the communication rate with an SPI Flash memory device may be 33MHz maximum. So if your system clock was 200MHz you would be looking to divide that by at least a factor or 6 and KCPSM6 could help to achieve that naturally. However, if you require higher 'bit banging' performance without just increasing the system clock frequency then clearly there is a limit when using the general purpose output ports.

```
OUTPUT s1, Dev_data_port
LOAD s2, 00000010'b
LOAD s3, 00000011'b
LOAD s0, 00000000'b
OUTPUT s2, Dev_control_port
OUTPUT s3, Dev_control_port
OUTPUT s2, Dev_control_port
OUTPUT s0, Dev_control_port
```

One potential workaround that has been used in KCPSM3 based designs in the past, and is still applicable to KCPSM6 designs, is to reorder your code. As shown on the left, the constant values have been pre-loaded into a set of registers so that the waveform can be generated with a burst of sequential OUTPUT instructions. Whilst this does result in the highest possible 'bit banging' transition rate of the signals during the actual generation of the sequence it also requires more registers to be used and the same amount of time is required to execute the code overall.

Hint – To generate single clock cycle pulses you can use the single clock cycle 'write_strobe' qualified by the 'port_id' rather than set and reset a data bit of a full output port.

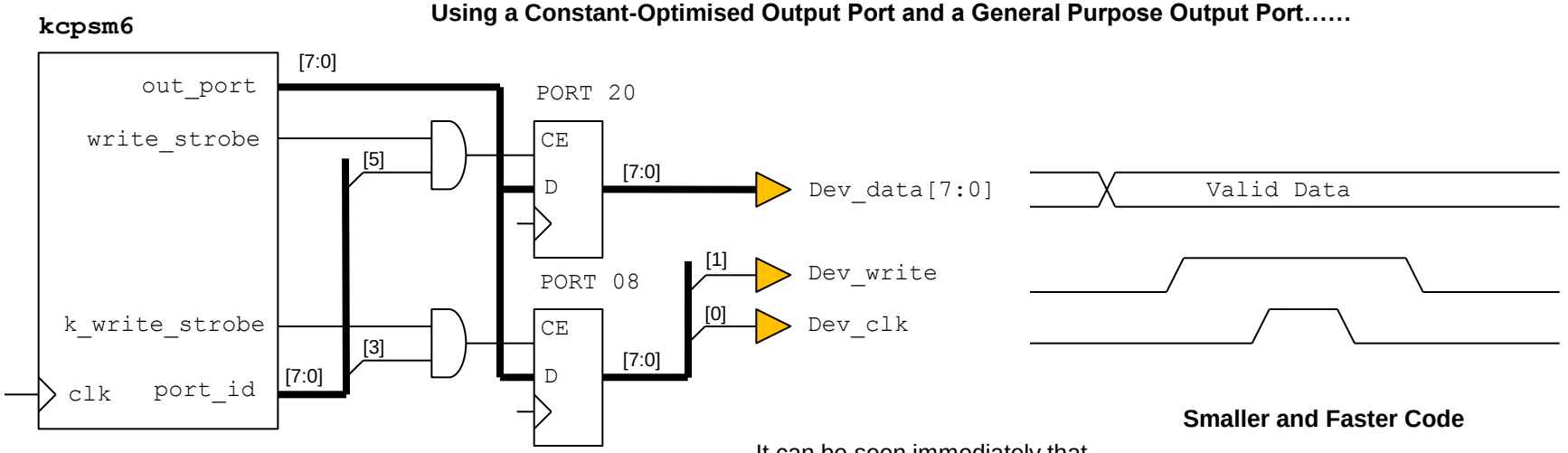
# Constant-Optimised Output Ports



KCPSM6 provides up to 16 constant-optimised output ports. From a hardware perspective these are used in an identical way to the general purpose output ports except that 'k_write_strobe' is used to qualify the port address which is presented on port_id[3:0]. Hence only port addresses '0' to 'F' (0'd to 15'd) can be used and port_id[7:4] should be ignored. Good optimum designs will allocate output port addresses to minimise the decoding of 'port_id' so this should not pose any challenges.

Returning to the same example of writing data to an external device we can see that port 08 hex has now been allocated to a constant-optimised output port by using the 'k_write_strobe' whilst port 20 hex is still associated with 'write_strobe' because the data is naturally variable. So there is very little difference in the hardware as long as you remember that only port_id[3:0] are defined during an OUTPUTK instruction. Note also that you could now have two different output ports with the same address; on for variable data and the other for constant values (see page 79).

**Using a Constant-Optimised Output Port and a General Purpose Output Port……**



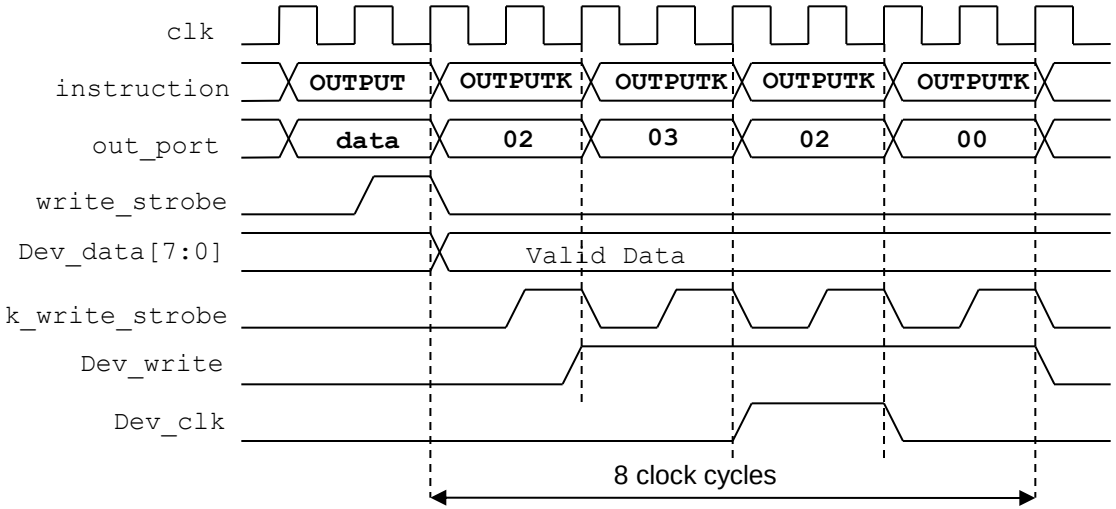

```
CONSTANT Dev_data_port, 20
CONSTANT Dev_control_port, 08
```

It can be seen immediately that all the LOAD instructions have been eliminated saving code space and reducing the execution time. This also means that register 's0' used previously to define the sequence of values is now free for another purpose.

**Smaller and Faster Code**

```
OUTPUT s1, Dev_data_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000011'b, Dev_control_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000000'b, Dev_control_port
```

# Constant-Optimised Output Ports
# OUTPUTK kk, p




**Smaller and Faster Code**

```
OUTPUT s1, Dev_data_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000011'b, Dev_control_port
OUTPUTK 00000010'b, Dev_control_port
OUTPUTK 00000000'b, Dev_control_port
```

Hint – Using a TABLE directive would also make this code easier to write.

This timing diagram clearly shows the performance advantage when using a constant-optimised output port for a 'bit banging' application. The example control sequence is now completed in 8 rather than 16 clock cycles. More significantly, the standard transition rate is every instruction or 2 system clock cycles. All without the need to use any registers.

**OUTPUTK kk, p**

The OUTPUTK instruction has two operands. The first operand is the 8-bit constant value 'kk' that will be presented on 'out-port' and therefore must be in the range '00' to 'FF' hex. The second operand must specify the port address that will be presented on port_id[3:0] and therefore must be in the range '0' to 'F' hex. This instruction has no effect on the contents of any registers used or the state of the flags.

Examples

```
CONSTANT token, 61
CONSTANT control_port, 0A

OUTPUTK 61, A
OUTPUTK 97'd, 10'd
OUTPUTK "a", A
OUTPUTK token, control_port
```

These examples show how the KCPSM6 assembler enables the constant and port to be defined and specified in multiple ways. All four 'OUTPUTK' instructions shown are actually the same!

The constant value 'kk' can be specified immediately using hex, decimal or an ASCII character. Alternatively the name allocated to a constant by a CONSTANT directive can be used.

The port address 'p' can also be specified immediately using hex or decimal but remember that this can only be in the range '0' to 'F' (0'd to 15'd). Likewise, the name of constant defined by a CONSTANT directive can be used providing that the value assigned to it also falls within the required range.

# ENABLE INTERRUPT
# DISABLE INTERRUPT


See also pages 40-44

These instructions are used to control when interrupts are allowed to happen. Following device configuration or the application of a reset to the KCPSM6 macro the program starts executing from address zero and interrupts are disabled. Quite simply, this means that a High level on the 'interrupt' input will be ignored. The 'ENABLE INTERRUPT' instruction is used to enable interrupts by setting the interrupt enable flag (IE = 1). Hence this instruction needs to be included at a suitable point in your code to activate the 'interrupt' input such that KCPSM6 will react to an interrupt request. 'ENABLE INTERRUPT' has no other effects.

**INTERRUPT ENABLE** | IE ← '1' | Z No Change | C No Change

Important – You should *never* execute an 'ENABLE INTERRUPT' within your ISR (i.e. anywhere between the interrupt vector and the RETURNI instruction). Only one interrupt can be serviced at a time and if you re-enable interrupts before the end of the ISR then there is every risk that another interrupt may occur.

The 'DISABLE INTERRUPT' instruction is used to disable interrupts interrupts by clearing the interrupt enable flag (IE = 0). This would typically be used to temporarily prevent an interrupt from interfering with the execution of a critical section of code. 'DISABLE INTERRUPT' has no other effects.

**DISABLE INTERRUPT** | IE ← '0' | Z No Change | C No Change

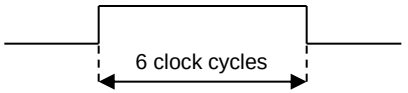
Hint – It is considered good coding practice if these instructions are only executed when they actually modify the state of the interrupt enable flag. Whilst it does not cause a problem to execute the instruction in a way that confirms the state (e.g. using 'ENABLE INTERRUPT' when IE is already '1') such a coding style makes it less clear at what points you in your code interrupts are enabled and disabled and this can lead to confusion when debugging in the long term.

Examples

```
          TEST s6, 02
          JUMP NZ, no_pulse
          DISABLE INTERRUPT
          OUTPUTK 01, trigger_port
          LOAD s0, s0
          LOAD s0, s0
          OUTPUTK 00, trigger_port
          ENABLE INTERRUPT
no_pulse: LOAD s3,  JUMP Z,
```

This section of code is taken from a program at a point when interrupts are enabled and therefore subject to interruption at any time that the interrupt input is driven High.

The state of Bit1 of register 's6' is tested, and if it is High, a pulse is generated on Bit0 of 'trigger_port'. A pair of 'LOAD s0, s0' instructions are used to stretch the pulse to be exactly 6 clock cycles in duration (3 instructions).



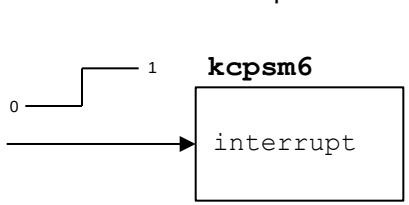

If an interrupt were to occur whilst generating the pulse then its duration could be considerably increased would have been unacceptable in this example. So to ensure that the pulse would always be 6 clock cycles long, interrupts are temporarily disabled only when the time critical code is executed.

Another example would be to temporarily disable interrupts whist the main program reads information from scratch pad memory that was put there by a previous ISR. This would ensure that the information read is a complete set and not a mixture of the information resulting from 2 separate interrupts.

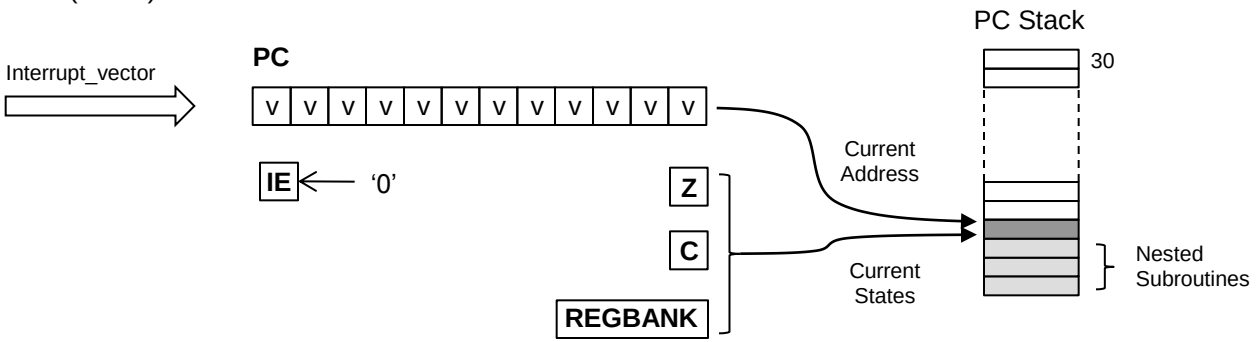
# RETURNI ENABLE
# RETURNI DISABLE



When an interrupt occurs the program counter is loaded with the interrupt vector and the current address (corresponding with the location of the instruction that is abandoned) is pushed onto the stack. In addition, the states of the carry flag (C), the zero flag (Z) and the register bank selection are also pushed onto the stack and further interrupts automatically disabled (IE = 0).




Effects of hardware interrupt when interrupts are enabled (IE = 1)



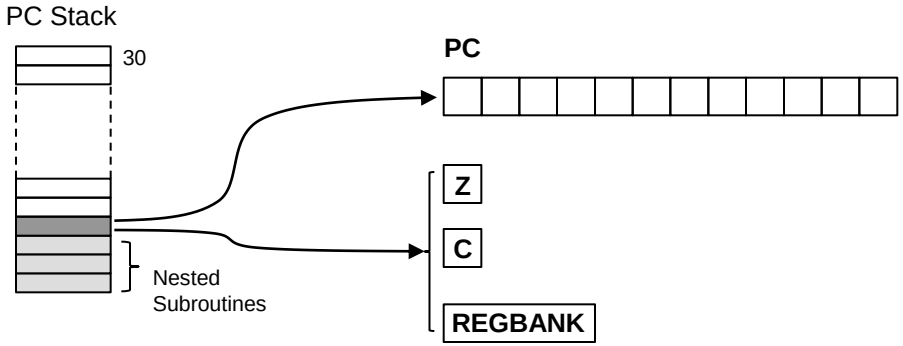

The 'RETURNI' instruction is similar to the unconditional 'RETURN' instruction *but it must only be used to terminate an interrupt service routine* (ISR). When the 'RETURNI' is executed, the last address held on the PC Stack is popped off and loaded directly into the program counter so that the program resumes execution starting with the instruction that was abandoned when the interrupt occurred. In addition, the RETURNI restores the values of the carry flag (C), the zero flag (Z) and the register bank selection so that they are exactly the same as when the interrupt occurred. Either the 'ENABLE' or 'DISABLED operand must be used to specify if interrupts are to be enabled or disabled on return from the ISR.

**RETURNI ENABLE**

IE ← '1'

**RETURNI DISABLE**

IE ← '0'




Continued on next page….

# RETURNI ENABLE
# RETURNI DISABLE



Important 1 – Always terminate an ISR with a 'RETURNI' and always terminate a normal subroutine with 'RETURN'. The execution of the inappropriate instruction will result in incorrect operation. Obviously that would be bad enough, but combined with the whole concept of interrupts occurring at any point in the execution of the main code the symptoms of the incorrect operation failure can be subtle and make it extremely difficult to identify the cause.

Important 2 – Just as each 'RETURN' must be executed to correspond with the 'CALL' that invoked a normal subroutine, a 'RETURNI' must only be executed to correspond with the interrupt that invoked the ISR. Your ISR can exploit KCPSM6's ability to implement nested subroutines just as they can be used in any part of your program but it is vital that each level is invoked and completed in order. The maximum number of levels is 30 and it should be remembered that an interrupt requires one of these levels. If an interrupt does result in a stack overflow then KCPSM6 will automatically generate an internal reset. Likewise if RETURNI is used in a way that results in a stack underflow then KCPSM6 will also reset itself automatically.

Examples

```
ISR: ADD sE, 1'd
     ADDCY sF, 0'd
     RETURNI ENABLE
```

This simple ISR increments the 16-bit value contained in the register pair [sF, sE]. This may relate to a scheme in which interrupts occur at regular intervals to provide the basis for a real time clock or timer (i.e. the value held in [SF,sE] is then used by the main program when required). The 'RETURNI ENABLE' instruction terminates the ISR and enables interrupts ready for the next time.

```
ISR: INPUT sF, int_data0
     STORE sF, 2A
     INPUT sF, int_data1
     STORE sF, 2B
     LOAD sE, 2A
     RETURNI DISABLE
```

This ISR reads two bytes of information from input ports and stores them in scratch pad memory. It is reasonable to assume that this information relates in some way to the reason for the interrupt and therefore probably represents some important information that had to be captured at that particular time. It can also be imagined that the main program needs to process this special information in some way with the value '2A' loaded into register 'sE' signifying that information has been captured and stored starting at location 2A hex. It can be imagined that the main program must be given time to process the captured information so the 'RETURNI DISABLE' instruction terminates the ISR but prevents a further interrupts overwriting the important information before it has been used. The main program would use an 'ENABLE INTERRUPT' once it had.

```
ISR: LOAD sA 00
     CALL motor_drive
     RETURNI ENABLE
```

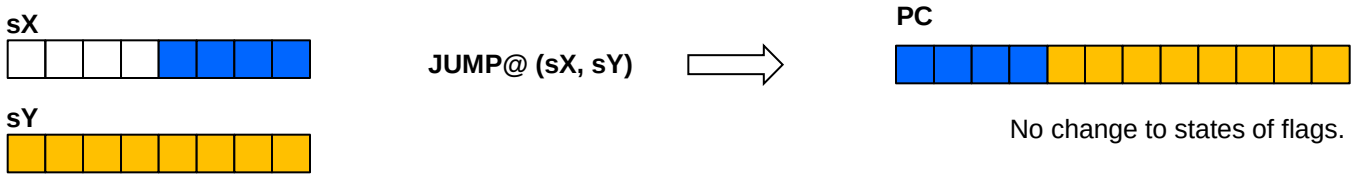
Providing all the normal rules of nested subroutines are followed then an ISR can also make use of subroutines.

```
motor_drive: OUTPUT sA, PWM_value
             OUTPUTK 01, update_strobe
             OUTPUTK 00, update_strobe
             RETURN
```

Hint – Be very careful to make sure that no code executed as part of your ISR procedure contains an 'ENABLE INTERRUPT' instruction.

# JUMP@ (sX, sY)

The ‘JUMP@ (sX, sY)’ is an unconditional JUMP which forces the program counter (PC) to the address defined by the contents of the ‘sX’ and ‘sY’ registers.




The 12-bit address is formed of the lower 4-bits of the ‘sX’ register and all 8-bits of the ‘sY’ register. The upper 4-bits of ‘sX’ are ignored and the contents of both registers are unaffected by the operation. There is no restriction on which registers can be used but it would be common coding practice to assign an adjacent pair such as ‘sB’ and ‘sA’.

Since the destination address is defined by the contents of the registers this is powerful instruction but also has the potential to be dangerous! You are entirely responsible for writing a program in which the computed address presented by the pair of registers corresponds with a valid location within your physical program space. The KCPSM6 assembler can do nothing to prevent you computing an inappropriate address but it does provide a mechanism to enable you to determine the addresses associated with line labels as shown in the following example.

Example This example assumes that a user selects an option from a menu by providing a numerical ASCII character in the range “1” to “4” (this range could easily be extended). The program reads this character, converts it to a value in the range 0 to 3 and then jumps to the appropriate routine of ‘choice’.

```
INPUT s0, selection_port
COMPARE s0, "1"
JUMP Z, choice1
COMPARE s0, "2"
JUMP Z, choice2
COMPARE s0, "3"
JUMP Z, choice3
COMPARE s0, "4"
JUMP Z, choice4
```

```
       LOAD sB, menu'upper
       LOAD sA, menu'lower
       INPUT s0, selection_port
       SUB s0, "1"
       ADD sA, s0
       ADDCY sB, 00
       JUMP@ (sB, sA)
menu:  JUMP choice1
       JUMP choice2
       JUMP choice3
       JUMP choice4
```

Without the ‘JUMP@’ instruction the menu would be implemented by a sequential series of compare and jumps (as shown on the right) which does not scale very well but is suitable when there is a small number of choices. Using the ‘JUMP@’ can help when there are lots of choices and also means that the execution time is the same regardless of the selection being made.

The KCPSM6 assembler provides ‘upper and ‘lower attributes that can be used with labels to define the 8-bit constants to be loaded into the registers. These abstracts of the LOG file show how the upper and lower parts of the address are resolved into ‘kk’ values.

```
7B4  01B07    LOAD sB, 07[menu'upper]
7B5  01ABB    LOAD sA, BB[menu'lower]
```

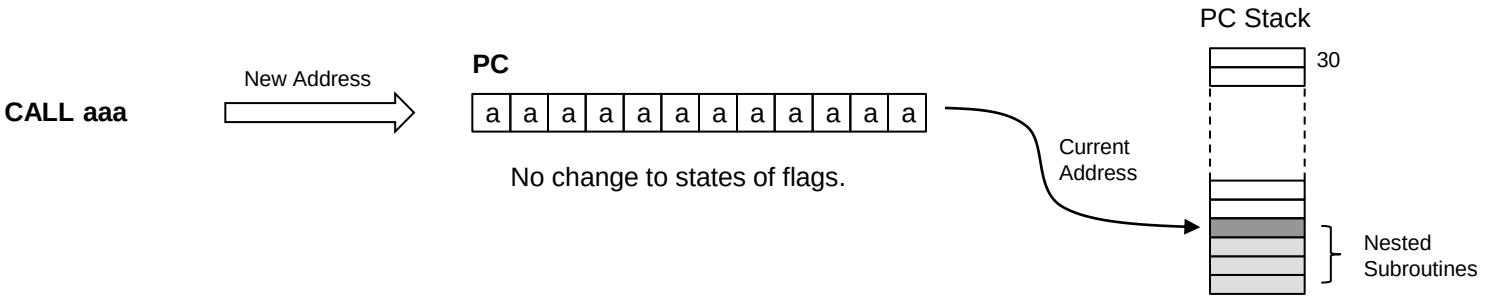
```
7BB  22862  menu: JUMP 862[choice1]
```

Hint - The ‘upper and ‘lower attributes can also be Used to derive ‘kk’ values for use in other instructions Such as ‘ADD sX, kk’ or ‘COMPARE sX, kk’.

# CALL aaa



'CALL aaa' is an unconditional CALL to a subroutine which pushes the current contents of the program counter (PC) onto the stack and loads the PC with the address defined by the value 'aaa'. A subroutine should end with a 'RETURN' instruction which will pop the last pushed address off of the stack, increment it and load it back into the program counter such that the program then executes the instruction following the initial CALL. Please also see the description of 'JUMP aaa' regarding the valid range of 'aaa' values and how the assembler is typically used to resolve their values for you.




Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that for each CALL made to a subroutine you have a *corresponding* RETURN. You must also ensure that execution of your program does not exceed 30 'nested' subroutines but this limit is rarely challenged by typical programs. Remember that an interrupt is a special case equivalent to a call and will use one level. If the stack does overflow then KCPSM6 will automatically reset.

Example Within some code a CALL is made to a subroutine called 'inc_count' which contains a 12-instruction procedure that increments a 32-bit number stored in 4 bytes of scratch pad memory. The corresponding RETURN at the end of the subroutine allows the program to continue.

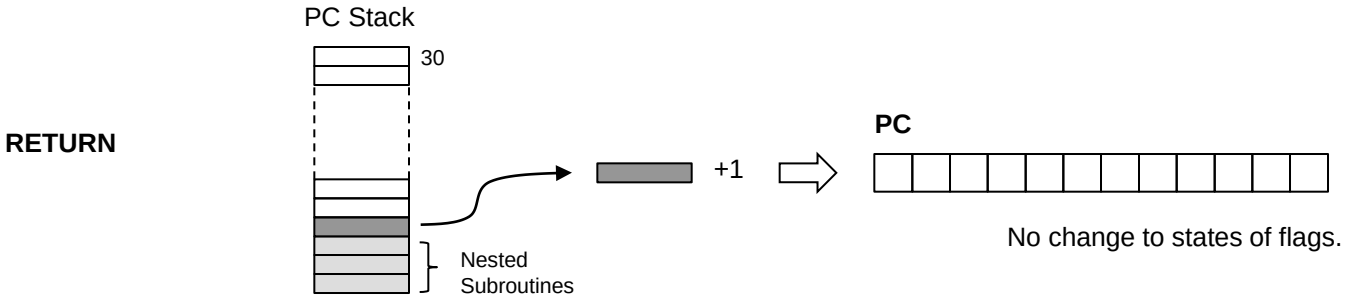
```
AND s0, 01
OUTPUT s0, status
CALL inc_count32
LOAD s0, 38
JUMP main_loop
```

```
inc_count32: FETCH s0, count0
             FETCH s1, count1
             FETCH s2, count2
             FETCH s3, count3
             ADD s0, 1'd
             ADDCY s1, 00
             ADDCY s2, 00
             ADDCY s3, 00
             STORE s0, count0
             STORE s1, count1
             STORE s2, count2
             STORE s3, count3
             RETURN
```

Hint – A subroutine can be located anywhere in a program relative to the CALL instructions that invoke it but it is vital that the subroutine is only executed as the result of a CALL otherwise their will be no address in the PC stack to correspond with the subsequent RETURN instruction.

# RETURN

The ‘RETURN’ instruction is used to unconditionally complete a subroutine. The last address pushed on to the PC Stack by the previous call to the subroutine is popped off the stack, incremented and loaded into the program counter. This automatic process ensures that the return is made to the address following the CALL instruction that initiated the subroutine.




Whist the PC Stack is completely dedicated and automatic you are entirely responsible for making sure that each RETURN is only executed to complete a subroutine that was invoked by the *corresponding* call instruction. If your code should incorrectly execute a RETURN that results in stack underflow then KCPSM6 will automatically reset. Remember that an interrupt is a special case equivalent to a call and requires a corresponding RETURNI instruction.

Example

```
        LOAD s9, 00
        LOAD s8, 00
        LOAD s1, 30'd
        CALL test_stack
        OUTPUT s9, 02
        OUTPUT s8, 01

test_stack: ADD s8, s1
        ADDCY s9, 00
        SUB s1, 01
        CALL NZ, test_stack
        RETURN
```

This example illustrates the general arrangement in which one part of the program calls a subroutine. In most cases line labels are used to make the code easier to write and maintain and the assembler resolves the actual addresses.

The subroutine labelled ‘test_stack’ is called from the main program. When this subroutine completes the RETURN forces the program counter to the address corresponding with the instruction immediately following the CALL which in this case is an OUTPUT instruction.

Whilst this example does show the general arrangement it actually describes a rather special case when we look at the code in detail. In the main program [s9,s8] has been cleared and then ‘s1’ has been loaded with 30 decimal. The ‘test_stack’ subrouine adds the value of ‘s1’ to [s9,s8] and then decrements the value in ‘s1’. But each time ‘s1’ is not zero it actually calls ‘test_stack’ again. Hence this subroutine is called 30 times and eventually [s9,s8] will be the sum of all values from 1 to 30 which is 465 (01D1 hex). When ‘s1’ does reach zero, KCPSM6 will execute the RETURN instruction 30 times until it eventually returns to the main program. Hence there is no restriction on how subroutines are arranged providing you do not exceed 30 levels and every CALL has a corresponding RETURN.