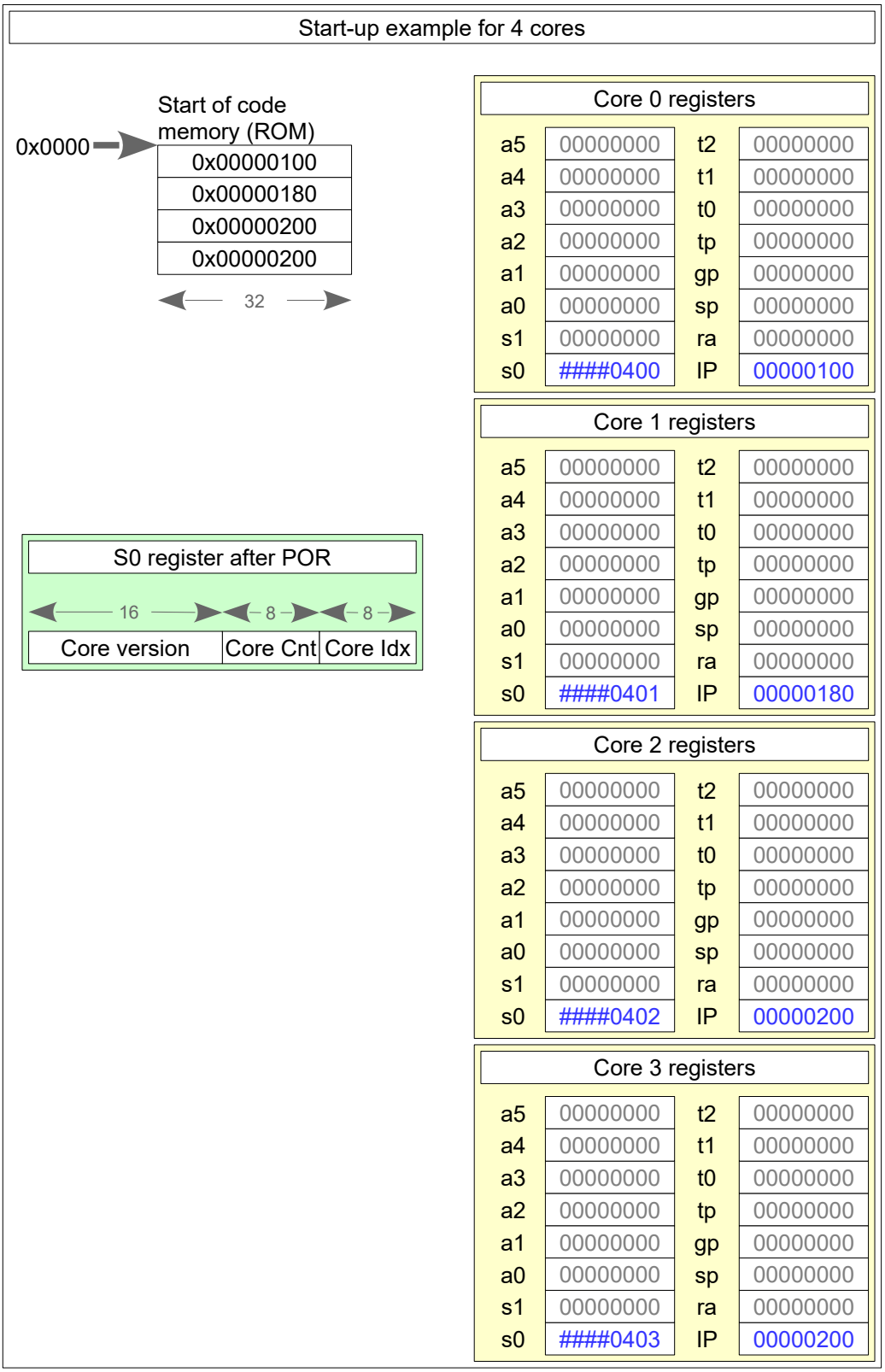
1 CPU startup

After power-on reset processor reads a table at the beginning of code memory. Each element of the table must contain the IP address of each processor core.



In the example above the system has 4 processing cores. Core 0 starts execution from address 0x00000100, core 1 starts execution from address 0x00000180, cores 2 and 3 start execution from address 0x00000200 eventually executing the same code. 32-bit register **s0** will contain information about

core/processor version, total number of cores and an individual core index. All other registers will contain zeroes.

So the SW has 2 possible ways to start-up in multi-core system:
- Define a dedicated address for each core.
- Start all cores at the same address and analyse the **s0** register at the beginning of execution.

If the SW designer is satisfied with this method, there is no need to go further: each core can execute it's own task (thread or hart).

But this way can be very inefficient: if the thread waits for an event or some data it can release the core and thus give processing time for another thread. Also, the SW system may contain more threads than cores.

2 Multithreading

The SW may consider the thread as an endless loop. For example, we can imagine a class where one of the member functions is a thread.

```
struct TUartComm
 {
  private:
    uint8_t FRecvBuf[16];
    uint32_t FWaitTime;
    bool FTerminated;
    uint16_t FRollingCounter;
  protected:
    void Process ( void );
  public:
    TUartComm ( void );
    ~TUartComm ( void );
 };

 extern volatile uint16_t HCpuRelease;

 void TUartComm::Process ( void )
 {
  while (FTerminated==false)
   {
    HCpuRelease=0;
    FRollingCounter++;
   }
 }
```

In the example above, "TUartComm::Process" is a thread which is implemented as an endless loop. Variable "FTerminated" will only be set "true" when the class instance is not needed and is to be destroyed.

Inside the loop the SW writes "0" to a variable called "HCpuRelease" in order to release the CPU and give the execution power to other threads. This is a special address and an IO port. The operation of writing zero to this port triggers HW to suspend the execution of this thread and to switch to another one. The execution of this thread will resume at line "FRollongCounter++" when all other threads release the CPU.

All CPU register, local variables will appear unchanged after resuming, thus the process of thread suspend is completely transparent for the SW.

This is one of the way of how the mechanism can be implemented by the SW. This way is the simplest one and for many SW will be just enough. But the threads can be driven by the interrupts as well: see dedicated chapter.

3 Interrupts

For the SW there is no need to declare any interrupt service routines, save/load registers or bother with stack allocations. This is because interrupts are served in threads.

In the previous chapter the simplest way of processing was presented. Let's imagine that the SW needs to receive 10 bytes of data by UART and at the same time check if there is a time out.

In the following example "HUartCtrl" is an IO port where bit [0] indicates if a new byte was received. Other bits may indicate an error. As an example imagine that bit [1] indicates a time-out.

"HUartData" is an IO port where SW can pick a newly received byte.

```
struct TUartComm
 {
  private:
    uint8_t FRecvBuf[16];
    uint32_t FWaitTime;
    bool FTerminated;
    uint16_t FRollingCounter;
  protected:
    void Process ( void );
  public:
    TUartComm ( void );
    ~TUartComm ( void );
 };

 extern volatile uint16_t HUartCtrl;
 extern volatile uint8_t HUartData;

 void TUartComm::Process ( void )
 {
  uint8_t BByteIdx;
  uint16_t BFlags;

  ...

  for (BByteIdx=0; BByteIdx<10; BByteIdx++)
   {
    BFlags=HUartCtrl;
    if ((BFlags & 0x01)!=0) FRecvBuf[BByteIdx]=HUartData;
    else break;
   }
  if (BByteIdx!=10) {} // Communication error

  ...
 }
```

Code line "`BFlags=HUartCtrl;`" will stop the execution of "`TUartComm::Process`" until at least one bit in "HUartCtrl" is set. The execution will be suspended and processing power will be given to other threads. The execution will only resume when at least 1 bit in "HUartCtrl" is set to "1". For the example above it is either the reception of a new byte or a time-out. For more complex hardware one can imagine several bits to be set at the same time.

The operation of reading "HUartCtrl" port will trigger HW to suspend the execution of this thread and execute other threads in the meanwhile. The execution will be resumed as soon as there is a necessity: either a reception of a new byte or a time out.

This is how the interrupts are handled. For the SW it looks like there are no interrupts at all and the execution is flat.

The SW can emulate the legacy way of interrupt handling like following. It may stay in an endless loop and call event handlers like in the following example.

```
struct TUartComm
 {
  private:
    uint8_t FRecvBuf[16];
    uint32_t FWaitTime;
    bool FTerminated;
    uint16_t FRollingCounter;

    void OnRecvByte ( uint8_t AData );
    void OnCommError ( uint8_t AFlags );
  protected:
    void Process ( void );
  public:
    TUartComm ( void );
    ~TUartComm ( void );
 };

 extern volatile uint16_t HUartCtrl;
 extern volatile uint8_t HUartData;

 void TUartComm::Process ( void )
 {
  uint8_t BByteIdx;
  uint16_t BFlags;

  ...

  while (FTerminated==false)
   {
    BFlags=HUartCtrl;
    if ((BFlags & 0x01)!=0) OnRecvByte(HUartData);
    if ((BFlags & 0x02)!=0) OnCommError(BFlags);
   }

  ...
 }
```

4 Memory protection unit and data relocation

The SW code becomes smaller and more optimal if it occupies the beginning of the memory. This is because the addressing is shorter. But as SW gets more and more complex and the code grows it becomes difficult to locate everything at the beginning. The solution would be to create a virtual address space when independent pieces of code are compiled to occupy the beginning of memory but are mapped to a physical address space.

Since each thread is more or less independent unit, the threads can benefit from that.

When the CPU picks a next thread for execution and loads an associated register context, it can automatically load shadow registers of memory protection unit. These registers keep memory location, size and access permission for the thread. The thread is not allowed to access any memory outside of the permitted area unless the access to certain [shared] memory areas is granted.

For each memory area the relocation address can be specified. Thread operates with virtual addresses (in order to make the code more optimal) but actual address is computed for a physical address space.

Imagine there are 2 threads in the SW: one is to handle UART communication and another one is to handle ADC.

The code below is given for illustration purpose only.

```
struct TUartComm
 {
  private:
    uint8_t FRecvBuf[16];
    uint32_t FWaitTime;
    bool FTerminated;
    uint16_t FRollingCounter;
  protected:
    void Process ( void );
  public:
    TUartComm ( void );
    ~TUartComm ( void );
 };

extern volatile uint16_t HUartCtrl;
extern volatile uint8_t HUartData;

void TUartComm::Process ( void )
{
  uint16_t BFlags;

  ...

 BFlags=HUartCtrl;
 if ((BFlags & 0x01)!=0) FRecvBuf[0]=HUartData;
  ...
}
```

```
struct TAdcSar
 {
  private:
   uint32_t FAdcData[16];
   uint32_t FWaitTime;
   bool FTerminated;
  protected:
   void Process ( void );
  public:
   TadcSar ( void );
   ~TadcSar ( void );
 };

extern volatile uint16_t HAdcCtrl;
extern volatile uint32_t HAdcData;

void TadcSar::Process ( void )
{
  uint16_t BFlags;

  ...

  BFlags = HAdcCtrl;
  if ((BFlags & 0x01)!=0) FAdcData[0]= HAdcData;
  ...
}
```

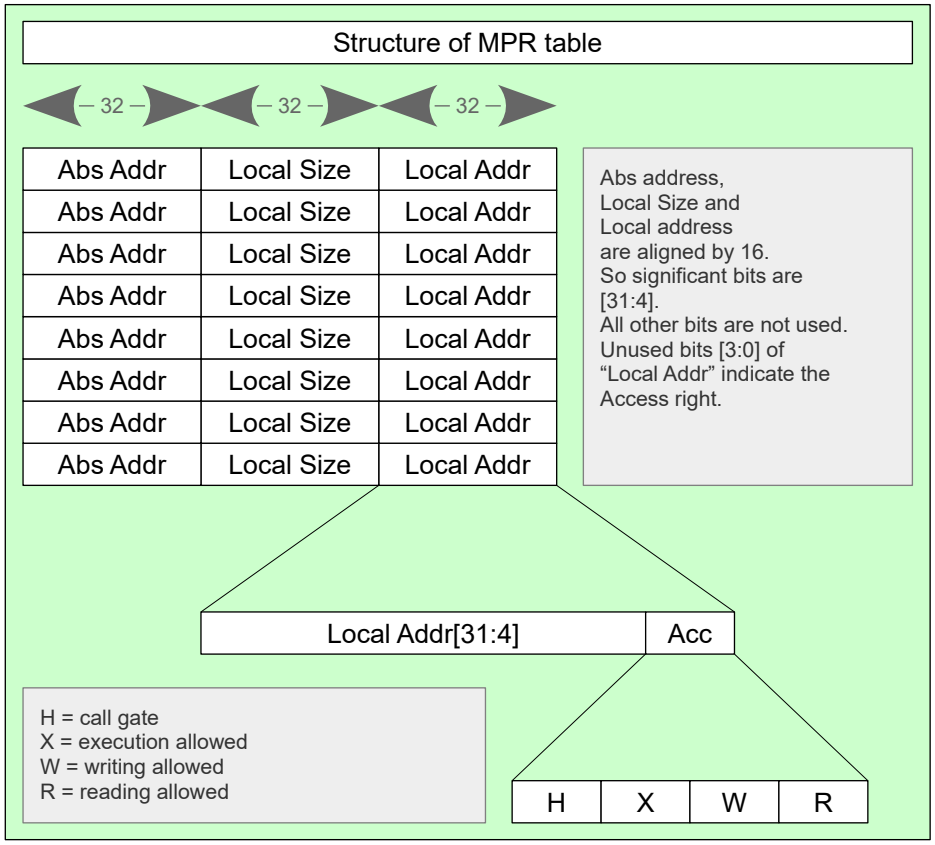
So there are 2 structs and 2 threads: "`TUartComm::Process`" and "`TAdcSar::Process`". We create 2 independent projects: one for UART and one for ADC. Each of the projects we compile individually, as optimal as we can. For each of them we indicate all addresses (especially RAM) to be located at the beginning of memory. These addresses may be the same. For example, RAM segment of UART module may occupy addresses from 0x100 to 0x1FF while RAM segment for ADC may occupy addresses from 0x100 to 0x2FF.

Then, we create 2 independent HEX files. These HEX files are loaded as is, without any changes. But the physical address space may be altered. Thus, UART RAM may be located at 0x1000 till 0x10FF and ADC at 0x1100 to 0x13FF.

The table has 8 records which should be enough to address ROM, RAM, Flash, Eeprom, IO spaces. The remaining 3 records may describe shared memory locations or be a gateway to access some system functions.

This table is quite big: up to 96 bytes, but it is not necessary to locate it in RAM. The table is static and it is not supposed to be changed during execution. It can be located in ROM. The gain in code size due to optimisation may largely exceed

By default, the table is not used by the Thread and is not loaded. Thus the memory access control is not activated and memory relocation is not used: virtual address = physical address.



See code example for how this table needs to be initialized.

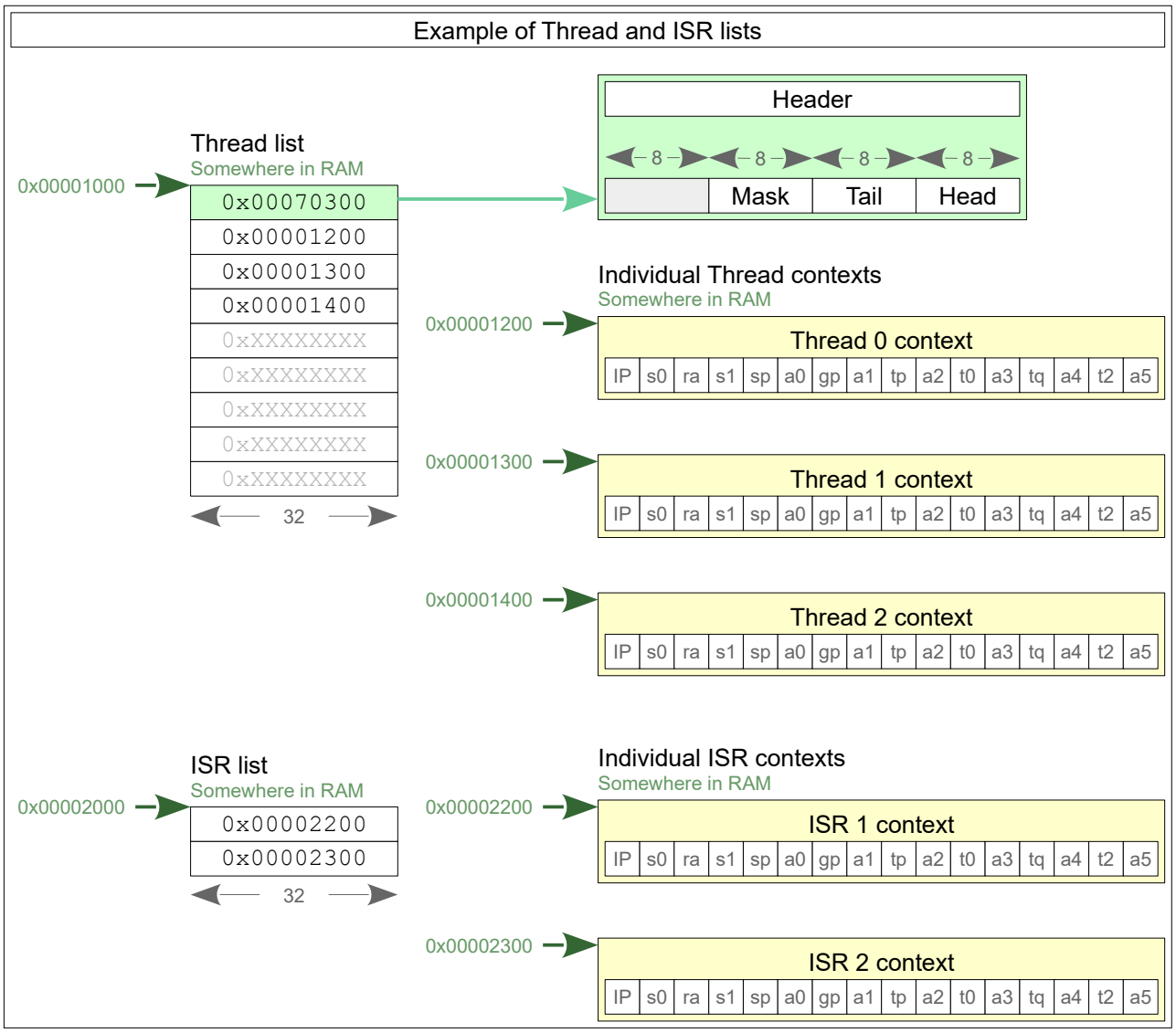
5 Thread and Interrupt table configuration

The HW performs a register context switch. The entire current register context is saved to memory and a new register context is loaded. This operation is much faster than saving/loading registers one-by-one: if the width of memory is 256-bit and the CPU register is 32-bit, then 4 registers a saved/loaded in one cycle. The operation bypasses cache in order not to provoke unnecessary page invalidation.

Though the multithreading is fully automated and is transparent for the SW, certain preparations steps need to be done.

SW needs to prepare a few tables and initialize them with addresses of threads and interrupt handlers. This chapter explains in details the structure of these tables.

The following explanation and the detailed example are provided for clarification purpose. You may want to skip reading this chapter because it contains too many details but may return to it later.



In the example above there are 3 threads and 2 ISRs. All elements can be located anywhere in RAM. Address is highlighted in green.

Thread table has a header, ISR table has no header. In Thread table the “mask” field indicates how many elements can be located in total. 0x07 = 8 elements, 0x0F = 16 elements and so on. It also contains the positions of Head and Tail. Software can add elements to the table and remove them. There cannot be more than 8 elements in total for the given example.

When Hardware performs a context switch, it stores the current thread at the Tail of the table and picks a new thread from the head of the table.

The SW can choose whether to perform a context switch or terminate the thread. In the second case the current context will not be saved and the core will only pick the next thread.

There is a special command “end thread” (In Mirabelle ISA there is a special opcode, but can be implemented as a bit in SFR).

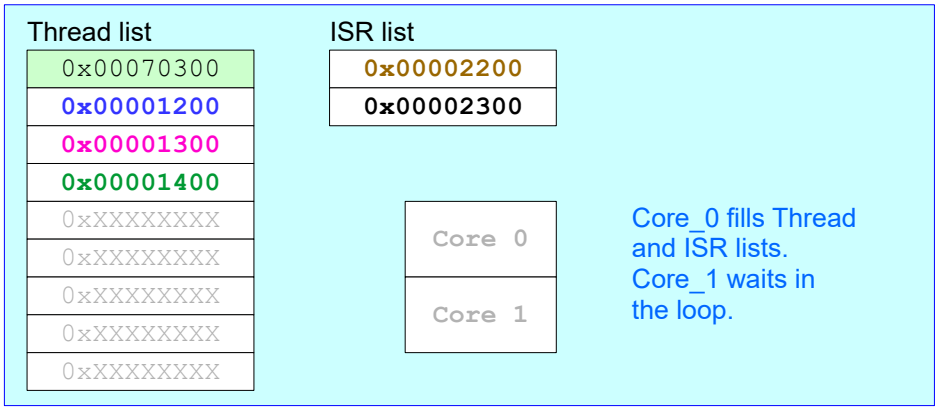
A generic conditional or unconditional JMP command can be used as “switch thread”.

Example

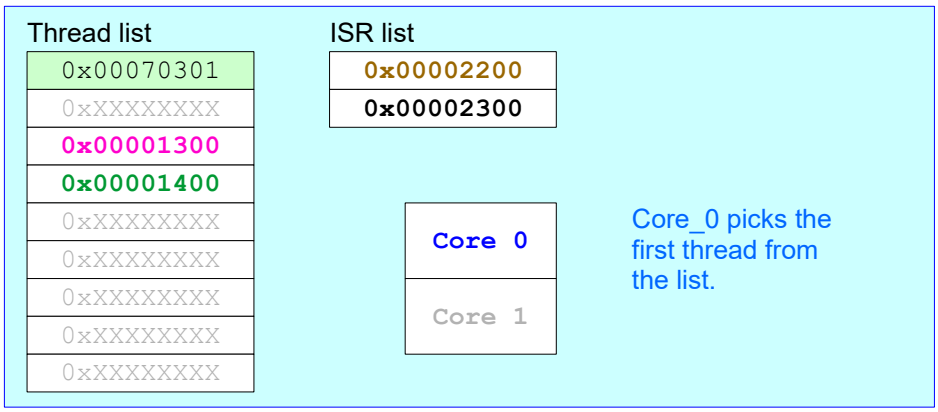
Imagine the HW containing 2 processing cores. For illustration purpose the thread context addresses will be highlighted by colours.

Let's see by example the processing step-by-step.

After POR core 0 will create the Thread and ISR tables, and core 1 will wait in the loop.

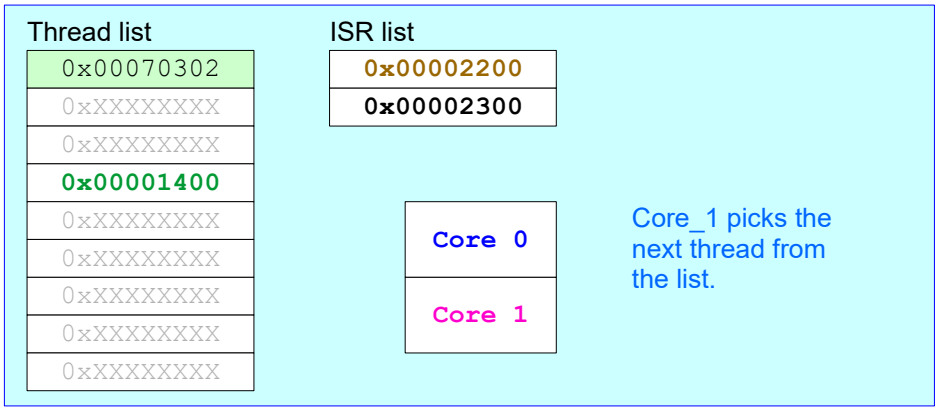


When Core_0 finishes with tables, it executes the “end thread” command and picks “blue” thread from the list.



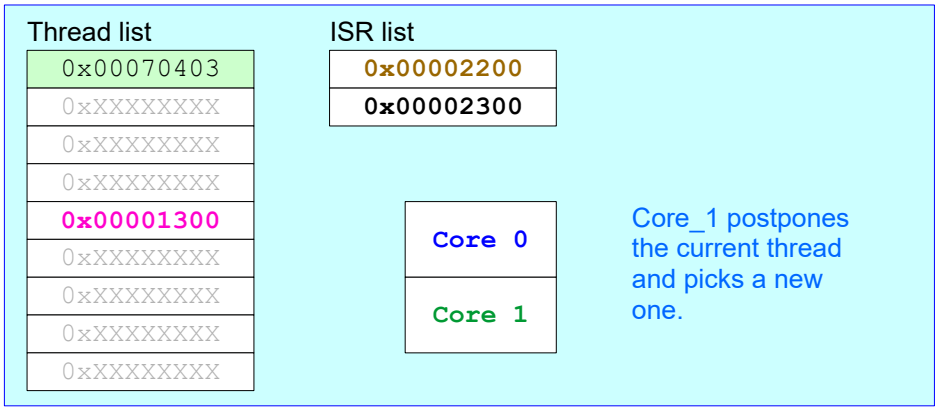
The Thread Head field (highlighted by green) in the Thread table header has changed from 00 to 01 (i.e. the value has changed from 00070300 to 00070301).

Now Core_1 exits the waiting loop and executes the “end thread” command and picks “pink” thread from the list.



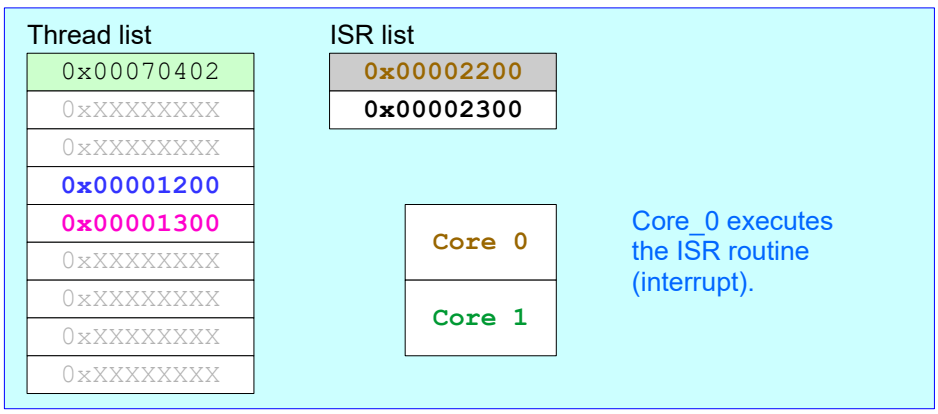
Head position has changed again, now from 01 to 02 (see value which is now 00070302).

While Core_0 is still busy with it's thread, Core_1 waits data from UART and data is not yet there. So, Core_1 can postpone the current thread and execute the next one. It performs the “context switch”.



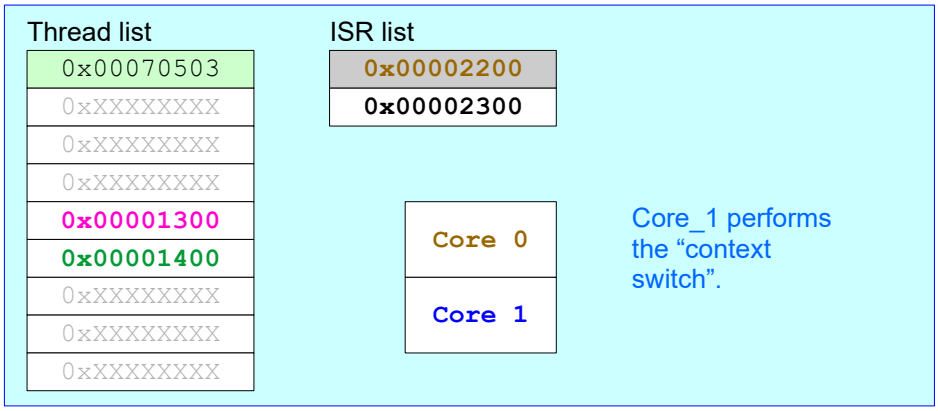
Now both Head and Tail position has changed (new value is 00070403).

Interrupt 0 arrives. Let's assume that interrupts are disabled by Core_1 and enabled by Core_0, so ISR will be executed by Core_0.



The thread interrupted by the IRQ is inserted to the Head (and not Tail) position. The Head index is decremented (new value is 00070402).

Let's imagine that while Core_0 is still executing the ISR, the “green” thread executed by Core_1 can be postponed. Core_1 will perform the “context switch”.

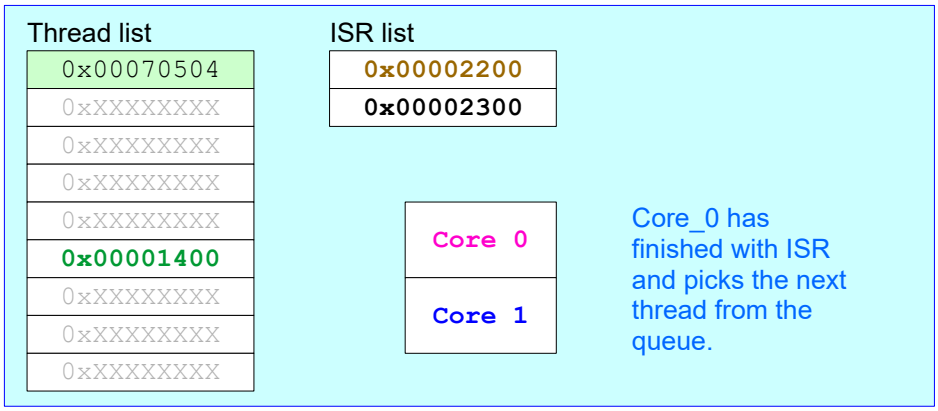


Core_1 picks the “blue” thread from the Head, interrupted some time ago by the IRQ.

This “blue” thread was executed by Core_0 before it was interrupted. It turns out that now it is Core_1 which continues.

Pay attention that when thread intentionally has decided to switch then it is written to the Tail position of the queue and thus has the longest time to wait a free core to pick it. But in this example the thread was interrupted by the incoming event, so it was not the thread itself who decided to switch. And in this case it is written to the Head position and has the highest priority and the shortest time to wait for the free core to be picked.

Now imagine that Core_0 has finished with ISR.



Note that the order of threads in the list can easily change.