MultiTasking Manual

E-LAB AVRco

Pascal Multi-Tasking for Single Chips

Version for

AVR

© Copyright 1996-2019 by E-LAB Computers

Blaise Pascal Mathematician 1623-1662

Table of Contents

Multi Tasking with E-LAB AVRco 3

Basics 3

Processes 3

SysTick 3

Tasks 4

Priorities 4

Additional Conditions for the Priorities 5

Start_Processes 7

Various System Functions of the Multi Tasking System 7

Stack and Frame 7

Locked Variable 8

DeviceLock 9

Pipes 9

Semaphore 9

Call Sequence 10

Multi Tasking with E-LAB AVRco

by Gunter Baab, revised by "Merlin"

Basics

MultiTasking and MultiTasking Operating Systems / -Compilers are extremely complex topics that are always controversial. Even for extremely powerful CPUs that are usually used with network servers there exists no "best solution" for such a design.
Even more contentiously debated is MultiTasking with Micro Controllers and their limited resources .
As a consequence with AVRco MultiTasking is "only" an option.

In order to understand the AVRco MultiTasking and to use it in an optimal way, it is essential to understand the implementation at the assembler level - at least at a basic level.
With imported MultiTasking (by ***Import*** *Tasks* ,***Import*** *Processes*) the AVRco includes the *scheduler* in the assembler code. The scheduler ensures that the independent defined modules (main, processes and tasks) are (virtually) executed in parallel.
This is achieved by running each module a certain (maximum) time until the next module gets the CPU. Thus the scheduler becomes the main program whose job is to start and stop one module after another. Because the main program is now the scheduler, the original Pascal main must be "degraded" to a normal module (more precisely to a process).

Important note:
Processes and Tasks are the modules of the Multitasking System.
They are exclusively administered by the System.
Even if they (may) have the structure of a subroutine you can not call them like you do with a standard subroutine.

Processes

Although a process has the formal structure of a subroutine, it works in a similar way to a main program with an infinite loop defined by **Loop – Main Prog - EndLoop.** The infinite loops of the processes are not explicitly programmed in the AVRco Pascal. Nevertheless, the compiler creates an assembler code sequence whose last command is a jump to the beginning of the sequence.
When a process is started by the scheduler for the first time, it is (roughly speaking - see below) equivalent to a simple assembler jump to the beginning of the process. From then on, this process would work its endless loop for ever, and the scheduler would never regain control.

This is where the *SysTick* enters the equation.

SysTick

The SysTick is a periodic timer interrupt that interrupts the active module (for example, the process just discussed) on a regular basis (recommendation: every 10 msec).
For this an 8-bit timer (timer 0 or timer 2) of the controller will be used exclusively.
The SysTick is the interrupt service routine of this interrupt. Here are, depending on the imported drivers (SysTimer, ADC, ...), various "system chores" performed, which are not of interest at this point.
Thereafter, the control is passed to the scheduler.
The scheduler now decides which module will continue working next. This may be the process just interrupted or a completely different module. This depends among others on the *Priorities* (see below).

As mentioned above, every process runs (with interrupts) endlessly "in a circle". However, this means that every process, when it regains control after an interruption, it must find its former state again, e.g. its registers and its stack (see below: *Stack, Frame*).
This, in turn, requires **each process** to have its own storage area in RAM for this runtime environment. Before the scheduler gives back control to a process, it must restore its runtime environment or -at the first call- initialize it. On the other hand, if a process is interrupted, the scheduler must save its actual runtime environment
(unfortunately, this is time-consuming since the ATMEL controllers only have one set of registers. Controllers with multiple register sets greatly simplify multitasking, since then at least the correct register set can be switched on quickly and easily).

A process thus runs until the computing time allocated to it is used up (see below: *Priorities*). However, if a process has nothing to do right now, it would be a waste of resources. Therefore, there are some techniques to prevent this. For example, a process can abort itself prematurely with the *Schedule* command (and give control back to the scheduler) or even suspend itself with *Suspend (self)* (until it is reactivated with a *Resume* command).

It is in general unpredictable how often a process runs. That would only be possible. if there were only processes, they would always be active and would all use the full computing time assigned to them. Sometimes the work to be done is very short (taking much less than a Systick for instance) and can be called at regular intervals, To put these into Processes would lead to excessive overheads, so *Tasks* have bee implemented. Tasks can also be used if an event needs to occur at regular intervals,

Tasks

a task is called by the scheduler at regular intervals. For example every 50ms (see: *Priorities*). Internally Tasks, are simple subroutines that end with a "Return from Subroutine" and thus return control to the scheduler (which started the task = called this subroutine).

A "well behaving" task runs from start to end and, unlike a process, never continues its work later because it is never interrupted. As a result, **there is only one storage area for the runtime environment** of all tasks, which can be an enormous resource savings.
Another result is that each task may only run **shorter than one single systick**.

If a task runs too long, it is aborted by the systick and the scheduler starts it again from the beginning on the next scheduled call. Such a task will never finish its work.

Priorities

every module (every task, every process (including the main process) has a priority. The priority is an integer number and is set when defining the task / process. The command *SetPriority* can be used to dynamically change the priority at runtime.

Examples:

```
Process Proc1 (32, 32 : iData; 5, suspended);   // prio= 5, no automatic start
Task Task1 (iData, 8, suspended);               // prio= 8, no automatic start
```

> note:
> the priority of a module and its state (*suspended / resumed*) are system variables that control the scheduler. These can be changed at any time with the corresponding functions (*SetPriority, Suspend, Resume*), regardless of whether the module or the scheduler is currently running.

Without explicitly specifying a priority, tasks get the priority 5, processes the priority 3 and the main process also the priority 5. The unit of the priorities is SysTicks (see below).

There are important differences in the definitions of the priorities in tasks and processes:

- if a task has priority n, it is called every n SysTicks. The lower the priority of a task, the more frequently it is called.

- if a process has the priority n, it runs a maximum of n SysTicks before the scheduler starts the next process. The lower the priority of a process, the less CPU time it gets allocated.

After the start of the multitasking system (see *Start_Processes*), the main process first gets control until it is interrupted by the first SysTick. In the SysTick the system chores are always done first and then the scheduler is started.

The scheduler starts the **First Task in the First SysTick** (first task the compiler found = first task defined in the source). As soon as this task has finished its work, the currently authorized process runs, until it is interrupted by the 2.SysTick.

The scheduler then starts **the Second Task in the Second .SysTick** ... then the currently authorised process, until it is interrupted by the third SysTick and so on.

When all tasks have been started once, their priority determines when they will be called again and generally tasks will not longer be started with each SysTick (see below: "*Additional Conditions*").but only when they are due.

The rest of the time (when no task is running) is used by the processes. If the main process has e.g. the priority = 3, then after 3 SysTicks the next process the compiler has found is authorised to run and gets the control for the number of SysTicks specified in its priority.

> Note:
> this also makes clear that the priority of a process defines its ***maximum*** computing time.
> The time for the "system chores" in the SysTick and, if applicable, the runtime for a task will reduce the effective process runtime. However, these times are typically unimportant compared to the duration of a process.

Important: **a maximum of one single task can run per SysTick**

Additional Conditions for the Priorities

if several tasks are active, there are important constraints on their priorities, otherwise a cyclic call is not possible:

1: the priorities of all tasks must have a common denominator

Why? Let's look at a (wrong!) Example: Task1 / Priority = 3 and Task2 / Priority = 4.
Now it should run

Task1 at tick number 1, 4, 7, **10**, 13, 16, 19, **22**, ... and
Task2 at tick number 2, 6, **10**, 14, 18, **22**, ...
At the 10. and 22. (etc.) SysTick actually two tasks would have to run, which is not possible.

However, if you give both tasks a (correct) priority = 3
Task1 runs at tick number 1, 4, 7, 10, 13, 16, 19, 22, ... and
Task2 runs at tick number 2, 5, 8, 11, 14, 17, 20, 23, ...
This is OK.

But why not both tasks with priority=2?
With this setting it should run

Task1 at tick number 1, 3, 5, 7, 9, ... and
Task2 at tick number 2, 4, 6, 8, 10, ...

Even then, no two tasks would ever have to run.

But if you now imagine that both tasks use their maximum allowed computing time (= 1 SysTick!) almost completely, it becomes clear that there is practically no time left for the existing processes (at least the main process is always active).
In the second example above with priority=3, the processes have the SysTicks 3, 6, 9, ... exclusively for their needs.
From this follows the 2nd condition:

2 .: the lowest priority of all tasks must be greater than the number of tasks (at least by 1)

As you can see in the second example, for all processes, only every third SysTick is guaranteed to be reserved as run time (plus the time the tasks do not consume).
This can, in particular if several processes are active, still result in too little computation time being available to them.
To be safe the
smallest allowed priority should be greater than "number of tasks + number of processes".
So, for example with 2 tasks, 2 processes + main process, the smallest allowed priority should be = 6.

> Note:
> If several tasks had to run in one SysTick (caused by a wrong priority assignment), the order of the task definition (in the source) determines which task is delayed. By a "particularly unfortunate choice" of the priorities, one can even "achieve" that a completely "bottom defined" task is never called because each time a "further defined" task receives control.
>
> e.g.
> defined is
> Task1/Priority=2 and
> Task2/Priority=4.
> Later you define
> (maybe after a certain time a quick modification without analysing the given situation)
> Task3/Priority=4.
> Now should run
> *Task1 at Tick number 1, 3, 5, 7, 9, 11, 13, 15 ...,*
> *Task2 at Tick number 2, 6, 10, 14, 18, ... and*
> Task3 at Tick number 3, 7, 11, 15, ...
> As you can see Taks3 is completely blocked by Task1 and will never run.

I hope it is now clear that it is not possible to make a simple and universally valid prescription for the optimal allocation of priorities. There may be applications where it is possible to deviate from the above rules. But then you have to respect the relationships and consequences:

e.g.:
- are all Tasks guaranteed to use extremely short runtimes
- are the Processes doing only minor short computations and need not much time
- maybe you can guarantee that only a part of the modules is active at the same time

etc.

If necessary, you can even dynamically adjust the priorities during the runtime to the respective operating status of the application, thus achieving optimal performance at all times. However, the maintenance and expansion of such complex programs quickly tends to turn into a nightmare.

Start_Processes

As mentioned, the scheduler is included by the definition of tasks and / or processes and called regularly by the SysTick.
The SysTick is a timer interrupt. For this to work, the interrupt system must usually be started with *EnableInts*. However, if MultiTasking is implemented, *Enable_Ints* must be **replaced** by *Start_Processes*.

Various System Functions of the Multi Tasking System

the AVRco includes a variety of functions to control the behaviour of the MultiTasking system. These are described in the compiler manual in the chapter "Multi-Task Functions" and should be sufficiently explained there with the above background knowledge.

..\E-LAB\DOCs\DocuCompiler.pdf - chapter "Multi-Task Functions"

At this point, therefore, only a few chosen topics will be discussed whose meaning and possible uses are less obvious.

Stack and Frame

the stack is used in AVRco the same way as in a non-multi-tasking system:
for storing the return addresses with subprogram calls / interrupts and for transferring variables (or their address) to subroutines.
The frame is used to store local variables and, if subroutines are called, also to temporarily store transfer parameters.

As explained above, all tasks have a common stack and frame area defined in the program header.

e.g.:

```
Define
  ...
  TaskStack     =     $0020, iData;
  TaskFrame     =     $0010; //TaskFrame uses same data page as TaskStack !
```

The task with the greatest needs defines the necessary sizes of the stack and frame.

Since processes are interrupted and later have to find their old environment again, **each** process has its **own** stack and frame area.
For the main process, these areas are also defined in the program header.

e.g.:

```
Define
  ...
  StackSize     =     $0064, iData;
  FrameSize     =     $0064, iData;
```

For all other processes, their respective stack and frame ranges are defined individually during the definition:

```
Process ProcessName (StackSize, FrameSize : word; DataPage);
```

Again, the same data page is used for stack and frame.

The necessary sizes of the stack and frame areas are (independent of the MultiTasking) thus determined solely by the structure of the respective program modules:

- are subroutines used and how deep are those nested
- what is the need for transfer parameters?
- what is the need for local variables?

A general recommendation is not possible.

However, if stack / frame is too small, the data of other processes will be overwritten resulting in a system crash. Unfortunately, a constant check of the free stack/frame is with a uC system not possible for performance reasons

The simplest way out of this dilemma is to test the program intensively in the simulator and simulate as many possible program constellations as possible. In the window "*Processes*" you can then look at the maximum consumption of stack / frame with the button "*State*".

Furthermore, there are some system routines that determine the current consumption at runtime. These are described in the chapter "*Multi-Task Functions*" in the section "*Stack and Frame Consumption*" and should be sufficient to solve such problems in difficult cases.

Locked Variable

a problem with MultiTasking is a competing access to data

Example:
one process is currently writing a global variable (which requires some assembler instructions for more complex types such as *integer* or even *float*) and is interrupted by another process while writing. If the interrupting process then reads this variable, it is not fully updated yet and trash is read.

Solutions:
either the variable is defined *"locked"*, which encapsulates every access in
"**DisableInts**" – (Access) – "**EnableInts**".

```
var xyz : integer, locked;
```

Disadvantage:
this variable must not be accessed continuously (polling) because then almost always the interrupts are locked and the MultiTasking can not work properly anymore.

Or the process prevents itself from being interrupted. This is with

```
Lock (self);
xyz := ...
Unlock (self);
```

easily possible.
Disadvantage:
the cyclic call of tasks is possibly disturbed.

DeviceLock

another problem is a concurrent access to the hardware.
e.g. process A writes to the first line of a display and process B writes to the second line.
While process A is writing it is interrupted, process B runs and the write cursor is positioned in the second line. Later process A regains control and ends the interrupted write - but the cursor will be somewhere in line 2 ==> muddle.
In such cases, a DeviceLock (boolean variable) can be used. Both processes have to take care of the lock state (write only if the display is not currently locked by the other process) and apply the Device Lock (set / <then write> / delete).

Pipes

Pipes allow easy data exchange between individual MultiTasking modules.
e.g. Module A receives / generates data. Module B processes these data.
For this Module A could allocate a memory area, write the data there and Module B could read these data. However, this is not very efficient because module B has to be called regularly - just to check the existence of new data ("polling").
A more sophisticated approach: the scheduler starts Module B only when new data are present.
For this purpose one can define a *Pipe*, a memory type (FIFO), which the scheduler is aware of.
In module B, the command *WaitPipe (<pipe name>)* is sufficient and the scheduler does the job of restarting module B only when data from module A has been stored in the pipe.
You do not have to worry about the "fill level" of the pipe: Module A can write until everything is done (assuming the pipe is big enough) and module B reads until the pipe is reported to be "empty".

Semaphore

Semaphores are used to synchronize competing resources. Although these are usually used in large systems (with multi-kernel CPUs, multi-path data channels, etc.), they also have potential applications with microcontrollers.
Semaphores are stored internally as bytes and have a value (0..255). In that regard, a DeviceLock is a special semaphore that can only accept two values (False / True).
Similar to *WaitPipe* there is a *WaitSema (<Sema-Name>)*. When the Semaphore is > 0, the scheduler activates the waiting module and decrements the Semaphore.

An application could be, for example, the use of a WLAN chip with several sockets.
It does not matter which socket a process uses so a semaphore could be used to control the release of sockets. Any Process that needs to transmit data issues a *WaitSema* and is delayed until a socket becomes free.

Call Sequence

The following diagram shows the switch of tasks and processes in the MultiTasking system

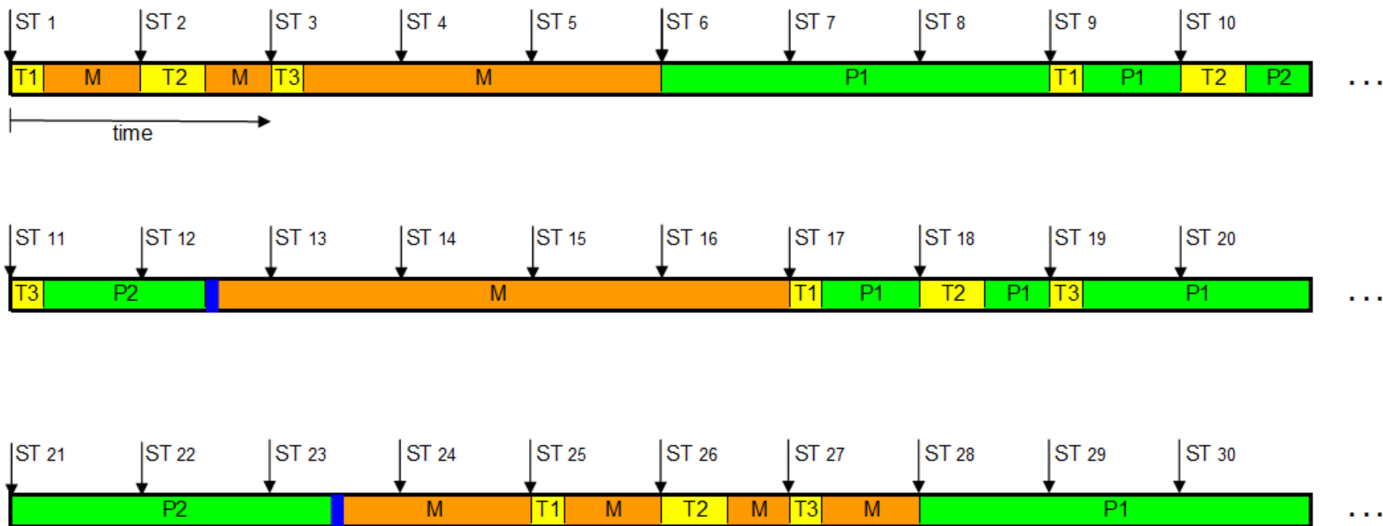
<u>Call Sequence of</u>

<u>Processes and Tasks</u>

<u>in E-LAB AVRco</u>

by Gunter Baab

ST x	SysTick No. x
M:	Main Process, Priority 5 (Default Priority)
T1, T2, T3:	Tasks, all Priority 8
P1,P2:	Processes
	P1: Priority 4
	P2: Priority 3, terminated with *Schedule*
▮ :	Command *"Schedule"* (P2)



Notes

©1996-2019 ***E-LAB Computers***
Grombacherstr. 27
D74906 Bad Rappenau

Tel. 07268/9124-0
Fax. 07268/9124-24

Internet: www.e-lab.de
e-mail: info@e-lab.de