FAST MOVING TECHNOLOGY

STÄUBLI

CS9 Real Time Val3/Fieldbus configuration

Technical documentation

White paper

TSS.0000.014.04-A - 06/07/2021
CS9 Real Time Val3/Fieldbus configuration

A "readme.pdf" document may be delivered on the robot's DVD. It contains the documentation addenda and errata.

Table of Contents

History

Revision	Modification	Date (yyyy-mm-dd)	By
A	Initial release	2021-07-06	A.TAGLIABUE
B			
C			

Version

That document has been tested with:

- SRC : s8.10.4
- Safety : 1.003 / SafePMT 3.0.0.28
- SYCON.net : The one of (SRS 2019.7.2)

Keyword

Val3, Real Time, Fieldbus, Synchronization.

1 Preliminary

DANGER

Instructions drawing the reader's attention to the risks of accidents that could lead to serious bodily harm if the steps shown are not complied with. In general, this type of indication describes the potential danger, its possible effects and the necessary steps to reduce the danger.
It is essential to comply with the instructions to ensure personal safety..

SAFETY

Instructions drawing the reader's attention that its responsibility is engaged if the steps shown are not complied with.
It is essential to comply with the instructions to maintain the robot safety level.

Caution

Instructions directing the reader's attention to the risks of material damage or failure if the steps shown are not complied with. It is essential to comply with these instructions to ensure equipment reliability and performance levels.

ELECTRICAL risk

Instructions drawing the reader's attention to the risks of electrical shock.
It is essential to comply with the instructions to ensure personal safety..

Information

Supplies further information, or underlines a point or an important procedure. This information must be memorized to make it easier to apply and ensure correct sequencing of the operations described.

2 Introduction

2.1 What is the aim of this document?

For standard applications, usually the asynchronous task management is enough. It is good to promote the asynchronous management because it is better from the CPU usage point of view and it is enough for 90% of industrial robotics applications. It is possible because the motion of the robot is not managed by a standard Val3 task: the Starc board controls the position loop of the robot in a synchronous way even if the Val3 runs only asynchronous tasks.

For this reason it is necessary to use Val3 synchronous tasks only when it is needed, for example for motion control of an external device or a reverse engineering application: Basically when is mandatory to send / receive data at a defined and absolutely constant frequency.

This document explains how to create a real time synchronization between a Val3 task and a Fieldbus.

3 System task management

3.1 What are the basic system tasks on CS9?

- Synchro system task (System)

This task handles the robot trajectory and refresh of system IOs boards, which are: CpuIO, DsiIO, FastIO, PowerSupplyIO, Rsi9IO, StarcIO.

- IO Refresh task (System)

This task handles the refresh of user IOs boards, which are, J206 (EtherCAT master), J207/J208 (Real Time Ethernet), Fieldbus (Hilscher CIFX50E-xx PCIe boards).

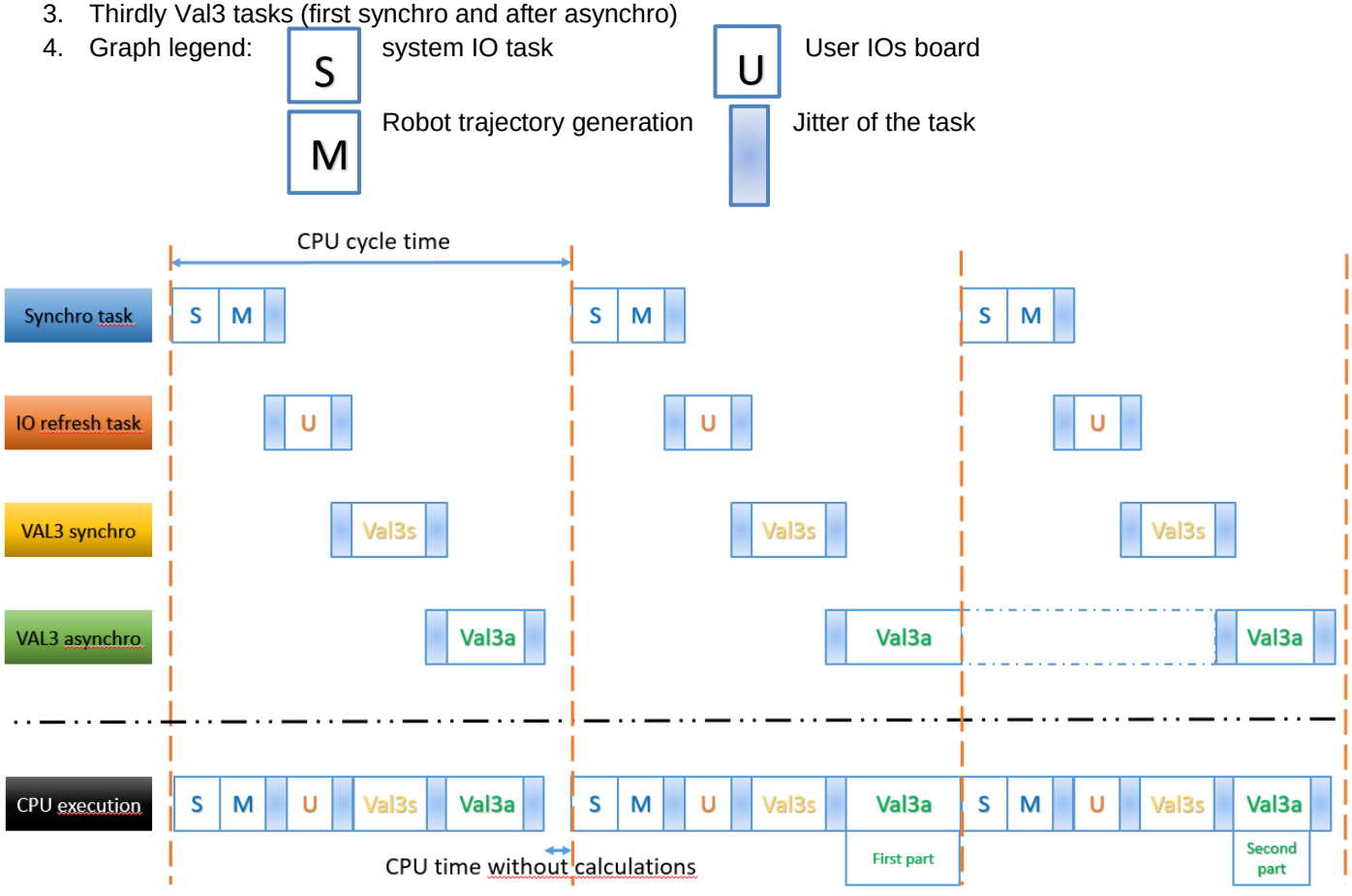
- Val3 task (User)

This section indeed contains many tasks, all of them can be written by user. Each of these tasks can be Asynchronous or Synchronous. In theory there is no limit to the number of different tasks, in reality the limit is the computation power of the CPU. Pay attention that a synchronous Val3 task consumes a lot of CPU resources. See chapter 4 of this document for more detailed information.

3.2 What happen in a single cycle of the CPU of a CS9?

In each scan the CPU executes:

1. Firstly the synchro system task
2. Secondly the IO refresh task
3. Thirdly Val3 tasks (first synchro and after asynchro)
4. Graph legend:

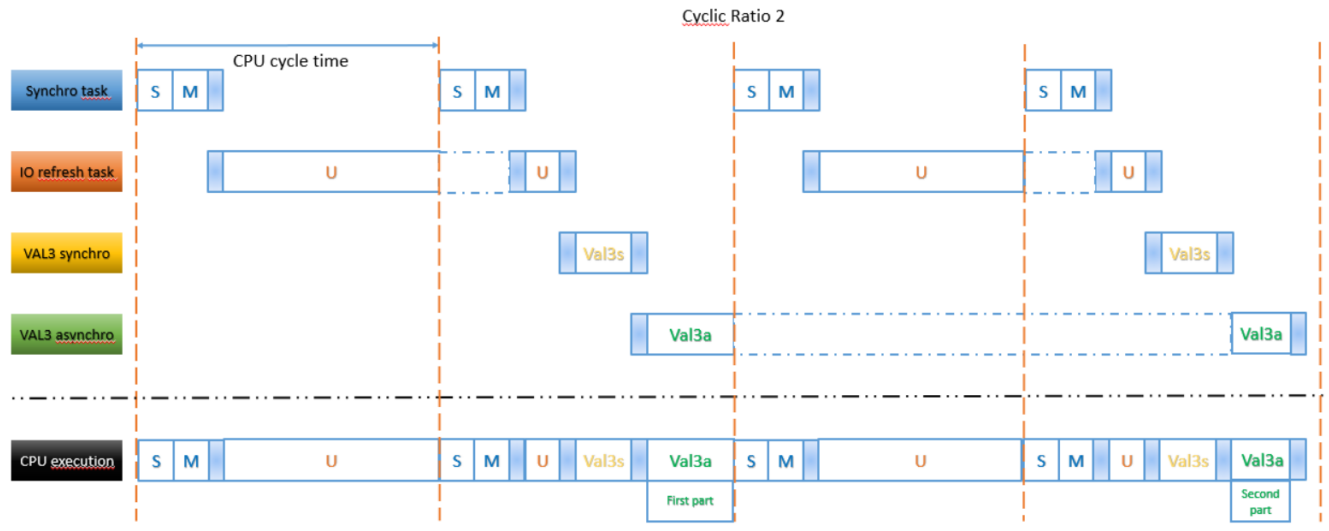


3.3 Cyclic ratio management

When the number of IOs is very high, the CPU could not be able to refresh all IOs in one scan. If it is, the IO refresh task stops with an error. For this reason, it is possible to configure the IO refresh cycle ratio: the variable means how many scan of the CPU the IO refresh task can take to update all IOs. The standard value is one, if for example cyclicRatio is set to 2, it means that the IO refresh task can take up to 2 scan of the CPU to refresh all IOs.

To change the cyclicRatio, in usr/configs/cell.cfx

```
<UserIORefresh>
      <Float name = "cyclicRatio" value = "2" />
</UserIORefresh>
```

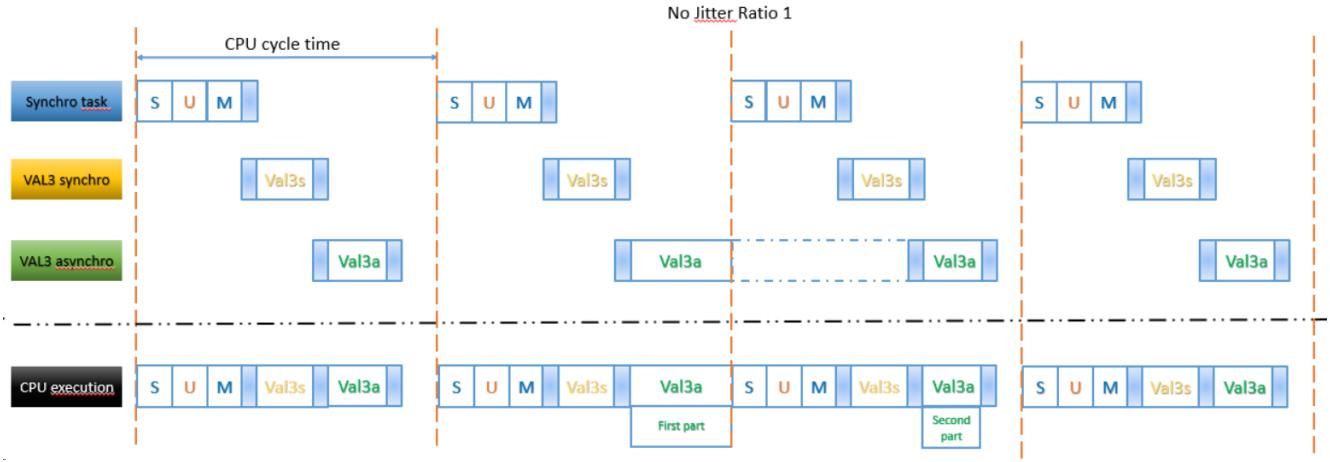


Obviously it is recommended to set this value as little as possible. In other words to increase this value only if it is needed. A bigger value of cyclicratio will lead to a bigger discrepancy between the Val3 program and the IO refresh.

3.4 Jitter Management

In no jitter mode, the IO refresh task does not exist and the user boards are refreshed during the Synchronous system task. Basically in each scan the CPU executes:

1. Firstly the synchro system task, which contains IOs
2. Secondly Val3 tasks (first synchro and after asynchro)

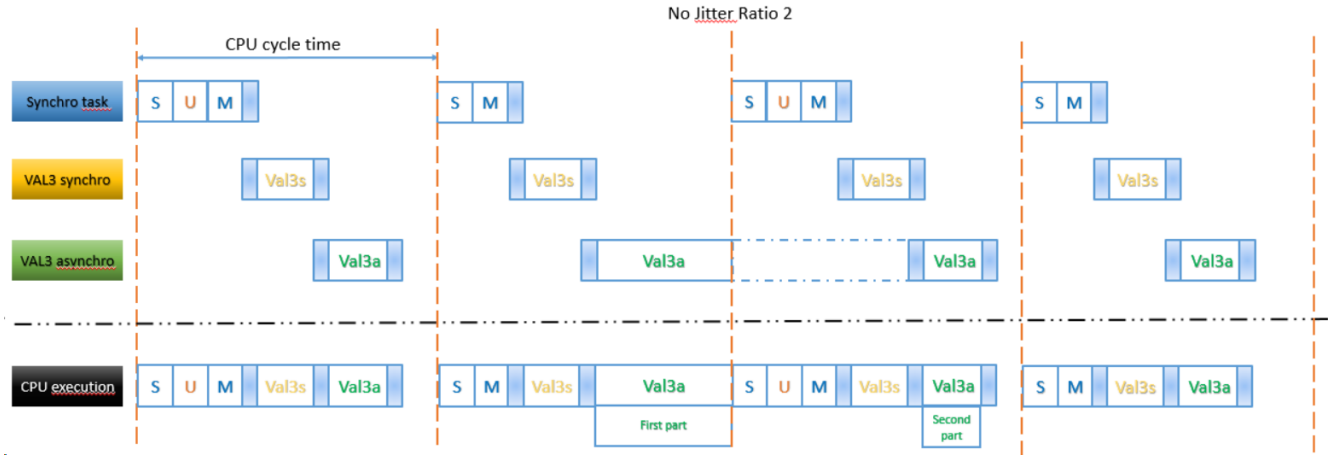


If the user board refresh time becomes too long an overrun will occur on the synchro system task: the robot is stopped and a reboot is needed. In this way we can force the system to be sure that the IOs are correctly updated each cycle. If not, the robot stops.

To change the Jitter mode, in usr/configs/cell.cfx

```
<UserIORefresh>
	<string name = "mode" value = "noJitter" />
</UserIORefresh>
```

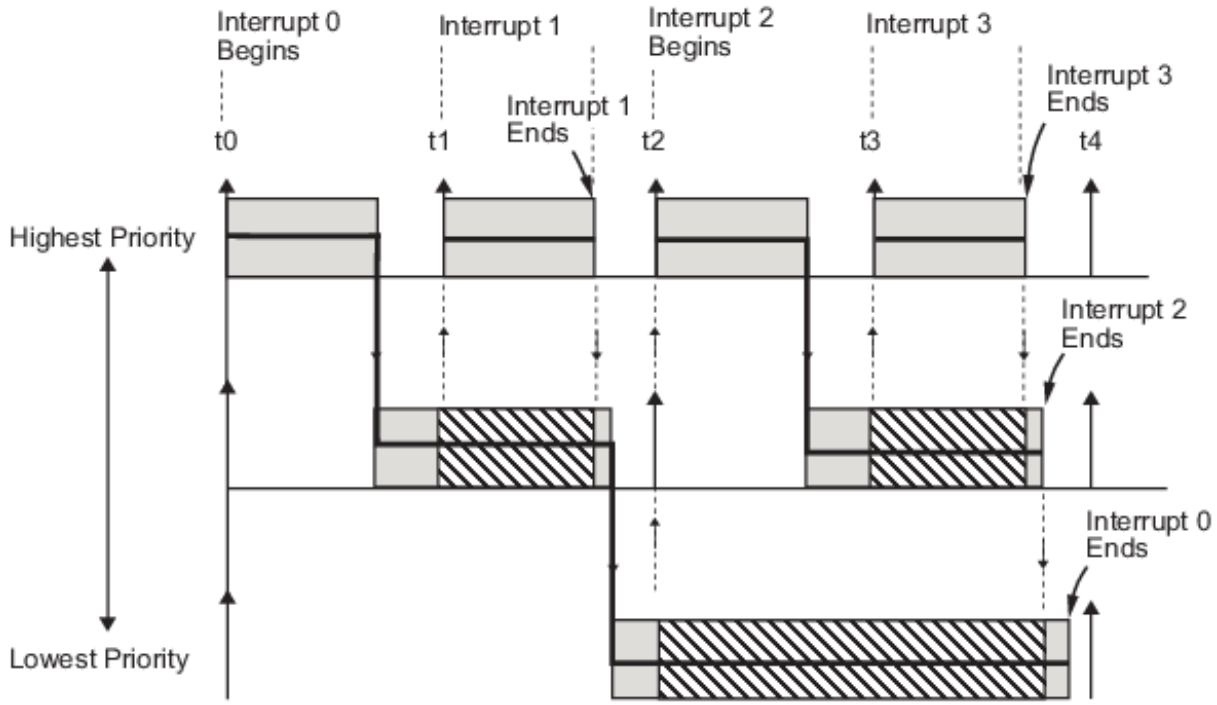
In no Jitter mode, the cyclic ratio management remains valid, the only difference is that the value two means that the IOs will be refreshed in the Synchro task every two scan. In this condition there will be a synchro task with IO refresh, the following one without, the one after with and so on…



4 Synchronous and Asynchronous Val3 task

4.1 How asynchronous Val3 task management works

When several tasks of an application are running, they appear to run concurrently and independently. This is true if the whole application is observed over a sufficiently long period of time (about a second), but not true if its specific behavior is examined over a short period of time.
In fact, as the system has only one processor, it can only execute one task at a time. Simultaneous execution is simulated by very fast sequencing of the tasks that execute a few instructions in turn before the system moves on to the next task. In the following picture is explained in which way is possible to execute many Val3 tasks at the same time with different priority:



Basically if the cycle time of the CPU is set to 4ms, every cycle of the CPU

VAL 3 asynchronous task sequencing obeys the following rules:
1. The tasks are sequenced in the order in which they were created
2. During each sequence, the system attempts to execute a number of VAL 3 instruction lines corresponding to the priority of the task.
3. When an instruction line cannot be terminated (runtime error, waiting for a signal, task stopped, etc.) the system moves on to the next VAL 3 task.
4. When all VAL 3 tasks have been completed, the system keeps some free time for lower priority system tasks (such as network communication, user screen refresh, file access), before a new cycle is started.
The maximum delay between two sequential cycles is equal to the duration of the last sequencing cycle; but, most of the time, this delay is null because the system does not need it.
The VAL 3 instructions that can cause a task to be sequenced immediately are as follows:

- **watch()** (condition wait timeout)
- **delay()** (timeout)
- **wait()** (condition waiting time)
- **waitEndMove()** (arm stop waiting time)
- **open()** and **close()** (arm stop waiting time followed by timeout)
- **taskResume()** (waits until the task is ready for restart)
- **taskKill()** (waits for the task to be actually killed)
- **disablePower()** (waits for power to be actually cut off)
- The instructions accessing the contents of the disk (**libLoad, libSave, libDelete, libList, setProfile**)
- The sio reading/writing instructions (operator =, **sioGet(), sioSet()**)
- **setMutex()** (waits for the Boolean mutex to be false)

4.2 How synchronous Val3 task management works

It is sometimes necessary to schedule tasks at regular periods of time, for data acquisition or device control: such tasks are called **synchronous tasks**.
They are executed in the sequencing cycle by interrupting the current asynchronous task between two VAL 3 lines. When the synchronous tasks have finished, the asynchronous task resumes.
The sequencing of the VAL 3 synchronous tasks obeys the following rules:

1. Each synchronous task is sequenced exactly once per period of time specified at the task creation (for instance, once every 4 ms).

2. At each sequence, the system executes up to 3000 VAL 3 instruction lines. It shifts to the next task when an instruction line cannot be completed immediately (runtime error, waiting for a signal, task stopped, ...).
In practice, a synchronous task is often explicitly ended by using the "delay(0)" instruction to force the sequencing of the next task.

3. The synchronous tasks with the same period are sequenced in the order in which they were created.

If the execution of a VAL 3 synchronous task takes longer than the specified period, the current cycle ends normally, but the next cycle is canceled. This overrun error is signaled to the VAL 3 application by setting the Boolean variable specified for this purpose at the task creation to "**true**". At the beginning of each cycle this Boolean variable thus shows whether the previous sequencing was carried out entirely or not.

5 Real time synchronization implementation in Val3

5.1 Principle

Basically the idea is to create a Val3 program that interacts with the external device each CPU cycle in a synchronous way. We need to be sure that the IOs are updated each scan of the CPU, not one IO update every two or three CPU cycle. To achieve this result, it is necessary to use a lot of notions explained in Chapter 3 and 4 of this document. It is needed:

- A synchronous task
- Cyclic ratio to 1 if not jitter, if not, could be more (see graph…)

5.2 Val3 Code

In this example it is explained how to manage the sending of the position of the robot on a fieldbus each cycle time of the CPU

It is needed to create a Synchronous task which reads the robot position and write it on the fieldbus variable:

```
taskCreateSync "HERE",nCycletimeSync,bOverrun,SaveWhereIam()
```

The task calculates the position of the robot and writes it on the fieldbus synchronously every cycle.

```
while true
  dOutRobPosReady=false
  // --- Calculate where the robot is now in flange world coordinates
  l_pHere=here(flange,world)
  // --- Sending position & timestamp to Profinet Slave
  aioSet(aOutTimeStamp2,(clock()*1000))
  aioSet(aOutPoint[0],l_pHere.trsf.x)
  aioSet(aOutPoint[1],l_pHere.trsf.y)
  aioSet(aOutPoint[2],l_pHere.trsf.z)
  aioSet(aOutPoint[3],l_pHere.trsf.rx)
  aioSet(aOutPoint[4],l_pHere.trsf.ry)
  aioSet(aOutPoint[5],l_pHere.trsf.rz)
  aioSet(aTimeStamp,(clock()*1000))
  dOutRobPosReady=true
  delay(0)
endWhile
```

And an Asynchronous task which sends the movements commands to the motion generator

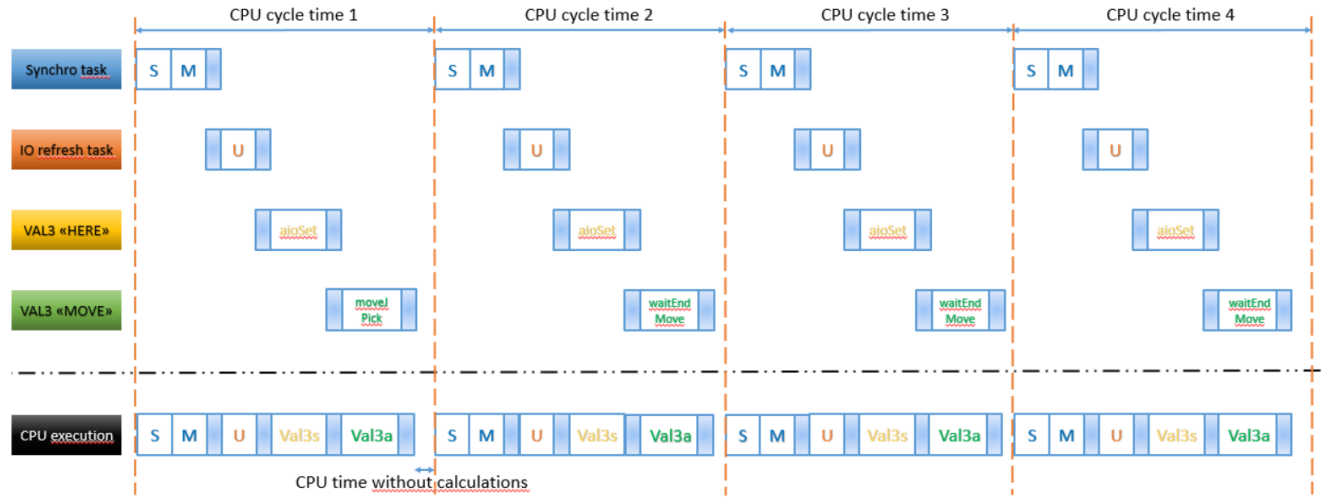
```
taskCreate "MOVE",100, RobotMotion()
```

```
while true
  // --- Move the robot to the desired position
  movej(jPick, tGripper, mNomSpeed)
  waitEndMove()
  movej(jPlace, tGripper, mNomSpeed)
  waitEndMove()
endWhile
```

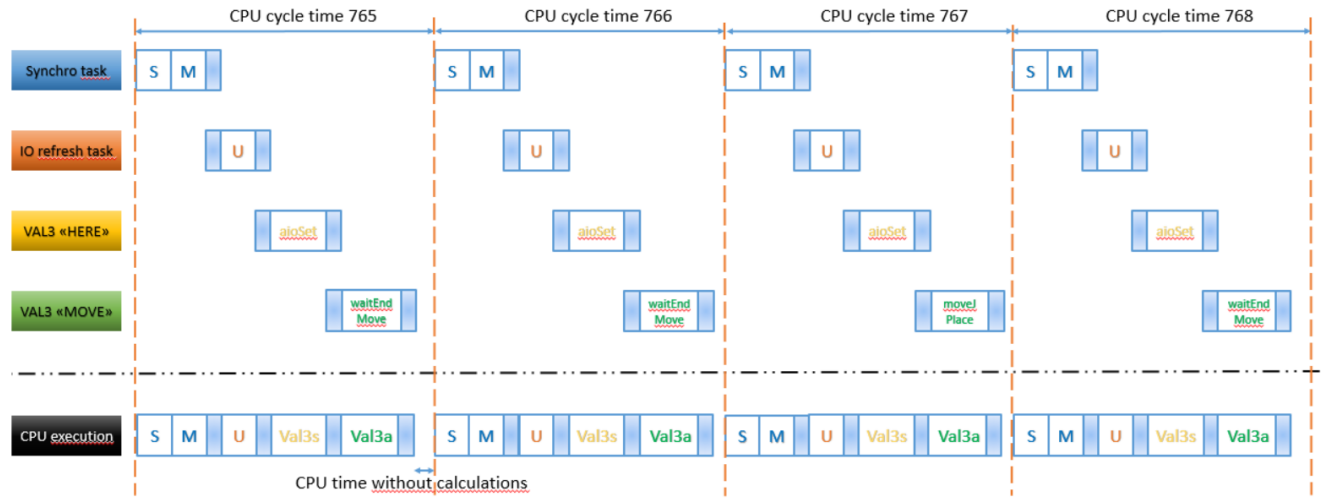
5.3 Execution

In the following graph you can see in a simplified version of what happen in the CPU when you launch the program:

```
begin
    taskCreateSync "HERE",nCycletimeSync,bOverrun,SaveWhereIam()
    taskCreate "MOVE",100, RobotMotion()
end
```

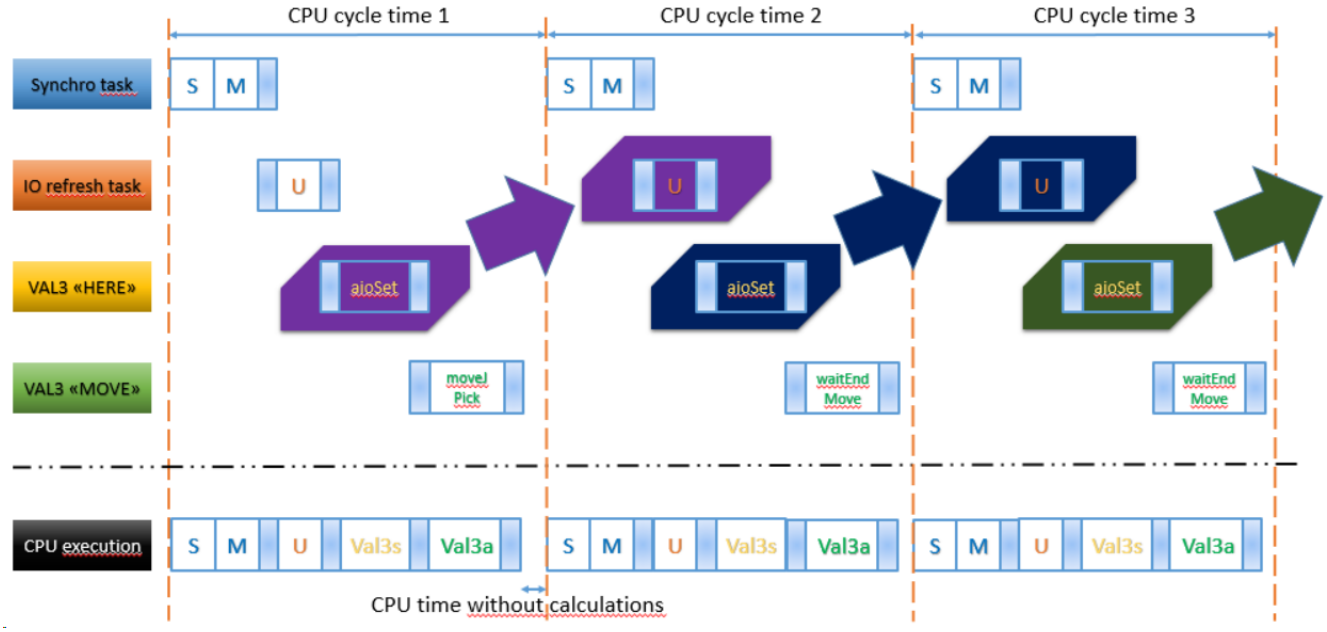


…after a while…



5.3.1 Io refresh

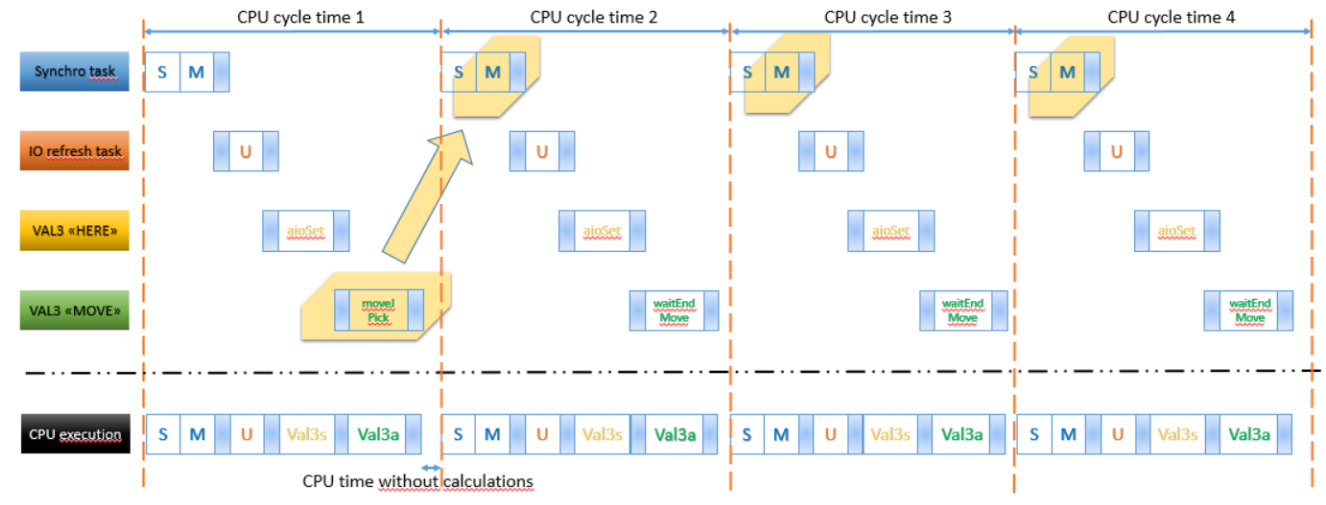
If we focus on the IO refresh, we see that the first aioset is after the first IO refresh task, it means that the coordinates collected in the cycle 1 by the synchro task are sent to the Fieldbus during the cycle 2:



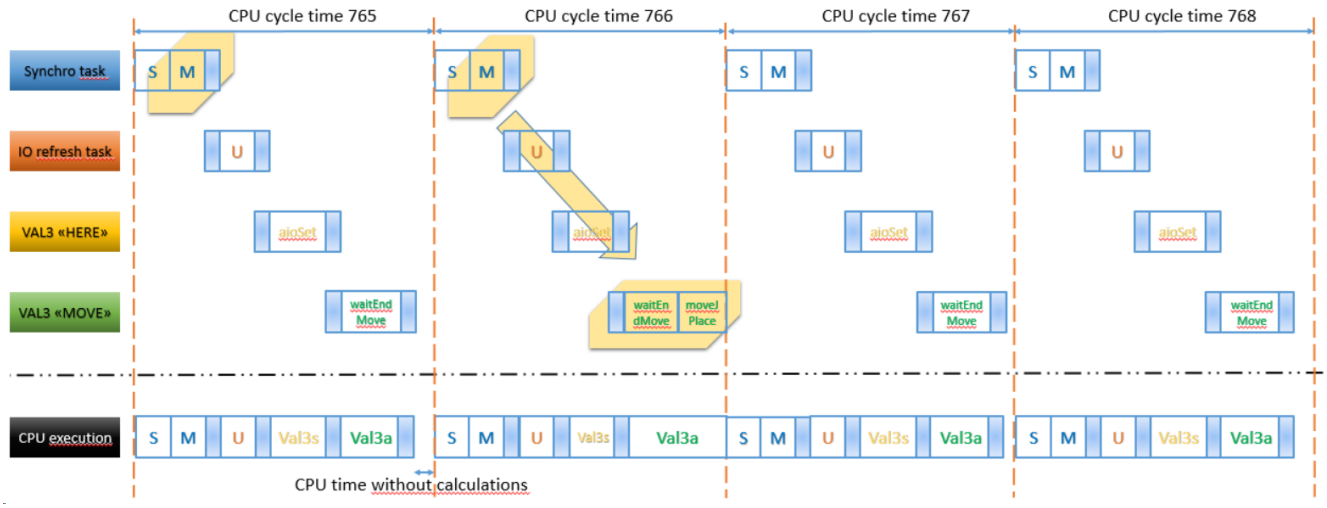
It means that if we are working with a cycletime of 4ms, the position sent on the fieldbus has a delay of 4ms compared to the position measured. Here is clear that if we use an asynchronous task to send the position to the fieldbus we are not sure of the frequency of the refresh rate of the IOs

5.3.2 Arm motion

During the first cycle, the asynchronous task “move”, which is executing the program contained in the routine RobotMotion(), executes the first moveJ. Starting from the cycle after, the asynchronous task stays in the waitEndMove() line waiting that the system Synchro task, during the motion part, handles the trajectory generation and execution till the arrival point.



When the arm will arrive to the final point, the waitEndMove will become true and the following line of the Val3 asynchro will be executed.



In our example, if we are working with a cycletime of 4ms, the Arm takes **3.056 seconds** to do the complete movement from jPick to jPlace.

Here is clear how much the arm movements and the CPU cycletime are frequency decoupled.