IPP-HURRAY! Research Group

Polytechnic Institute of Porto
School of Engineering (ISEP-IPP)

Fieldbus Networks: Real-Time from the Perspective of the Application Tasks

André MENDES
Luís Lino FERREIRA
Eduardo TOVAR

HURRAY-TR-0011
June 2000

relatório técnico

technical report

THIS WORK IS PARTIALLY SUPPORTED BY FCT UNDER PROJECT DEAR-COTS (Praxis/P/EEI/17/1998) AND BY FLAD UNDER PROJECT SISTER (Proj.471/97)

Fieldbus Networks: Real-Time from the Perspective of the Application Tasks

Luís Lino FERREIRA, Eduardo TOVAR

IPP-HURRAY! Research Group
Polytechnic Institute of Porto (ISEP-IPP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto
Portugal
Tel.: +351.22.8340502, Fax: +351.22.8321159
E-mail: {llf, emt}@dei.isep.ipp.pt
http://www.hurray.isep.ipp.pt

André MENDES

Polytechnic Institute of Porto (ISEP-IPP)
E-mail: andre@.ipp.pt

Abstract:

Fieldbus communication networks aim to interconnect sensors, actuators and controllers within distributed computer-controlled systems. Therefore, they constitute the foundation upon which real-time applications are to be implemented. A potential leap towards the use of fieldbus in such time-critical applications lies in the evaluation of its temporal behaviour. In the past few years several research works have been performed on a number of fieldbuses. However, these have mostly focused on the message passing mechanisms, without taking into account the communicating application tasks running in those distributed systems. The main contribution of this paper is to provide an approach for engineering real-time fieldbus systems where the schedulability analysis of the distributed system integrates both the characteristics of the application tasks and the characteristics of the message transactions performed by these tasks. In particular, we address the case of systems where the Process-Pascal multitasking language is used to develop P-NET based distributed applications.

Fieldbus Networks: Real-Time from the Perspective of the Application Tasks

André Mendes, Luís Ferreira, Eduardo Tovar

Polytechnic Institute of Porto, ISEP-IPP, IPP-HURRAY! Research Group
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto, Portugal
E-mail: andre@ipp.pt; {llf, emt}@dei.isep.ipp.pt
http://www.hurray.isep.ipp.pt

Abstract

Fieldbus communication networks aim to interconnect sensors, actuators and controllers within distributed computer-controlled systems. Therefore, they constitute the foundation upon which real-time applications are to be implemented. A potential leap towards the use of fieldbus in such time-critical applications lies in the evaluation of its temporal behaviour. In the past few years several research works have been performed on a number of fieldbuses. However, these have mostly focused on the message passing mechanisms, without taking into account the communicating application tasks running in those distributed systems. The main contribution of this paper is to provide an approach for engineering real-time fieldbus systems where the schedulability analysis of the distributed system integrates both the characteristics of the application tasks and the characteristics of the message transactions performed by these tasks. In particular, we address the case of systems where the Process-Pascal multitasking language is used to develop P-NET based distributed applications.

1. Introduction

This paper focuses on the field level of the automation hierarchy, where typically the process-relevant field devices are used by a computer system to automatically conduct the process.

Basically, the computer system should be able to receive, via the instrumentation interface, information about the status of the controlled object, compute new commands according to the references provided by the man-machine interface, and transmit new commands to the actuators, also via the instrumentation interface. To perform these operations, the computer system should be provided with a control application program. The connection between the control system and the sensors and actuator can be made by point-to-point links or by means of a (field level) network.

Typically, a field level network will be a broadcast network (like in most types of local area networks), where several network nodes share a common communication channel. Messages are transmitted from a source node to a destination node via the shared communication medium. A major problem occurs when at least two nodes attempt to send messages via the shared medium at about the same time. This problem is solved by a medium access control (MAC) protocol.

The control software can run on one controller (centralised control) or the control functions can be distributed by several control units each one performing a part of the control algorithm. This kind of architecture is usually called a distributed computer-controlled system (DCCS)[1].

A DCCS is implemented by a set of computational devices. Each computational device runs a number of tasks. These tasks may communicate their results by passing messages between computational devices across a field level communication network. In order to guarantee that the timing requirements of the DCCS are met, the communication delay between a sending task queuing a message, and the related receiving task being able to access that message, must be upper bounded. This total delay is termed end-to-end communication delay [2], and is composed of the generation delay (time taken by the sender's task to generate and queue the related message), queuing delay (time taken by the message to gain access to the field level communication network), transmission delay (time taken by the message to be transmitted on the field level communication network), delivery delay (time taken to process the message at the destination processor before finally delivering it to the destination task).

In terms of the response time analysis of tasks, distribution brings the need to include the end-to-end communication delays, as one of the components of the overall task's response time. The behaviour of the tasks will also determine the communication pattern between applications.

In this paper, we will focus on this holistic approach for engineering real-time DCCS. The important contribution is not only in the consideration of a specific fieldbus network, the P-NET [3], but also in reasoning the real-time analysis from

the point of view application tasks. In this paper we address the case of a commercial software tool for developing distributed applications for P-NET networks: the Process-Pascal language [4].
The remainder of this paper is organised as follows. In Section 2, we analyse the task models available in Process-Pascal. Importantly, we reason on how the different types of tasks interact with each other when contending. In Section 3 we review previous relevant work in determining the worst-case response of a task in a single processor system. This analysis will be the basis for the response time analysis of Process-Pascal tasks, which is proposed in Section 4. In Section 5 we introduce the P-NET network aspects by analysing the worst-case response time of communicating Process-Pascal tasks. Importantly, we show the impact of the specific Process-Pascal task models in the evaluation of the end-to-end communication delay of P-NET messages and propose a real-time guaranteed approach for developing distributed applications with Process-Pascal where distribution is provided by P-NET networks. Finally, in Section 6 we draw some conclusions.

2. Task's Model in Process-Pascal

The Process-Pascal language is similar to standard Pascal but it includes some extensions to allow multitasking and to enable interoperation with industrial microprocessor-based controllers. One of those extensions targets the use of P-NET networks to support the access to variables in remote network nodes.
The multitasking capabilities of Process-Pascal allow the division of a program into a set of tasks, each one performing a distinct function. These tasks are scheduled by the operating system running on the network node (typically a controller). The philosophy employed in Process-Pascal tries to give the user some control over the scheduling process, by allowing enabling/disabling pre-emption or even to control in which points of the programs the scheduler should run.
In general, Process-Pascal tasks should contain its code within an endless loop, like it can be seen in the following pseudo-code example:

```
Task GeneralControl
Begin
   (* initialisation code *)
   Loop
      (* code of the task *)
      ChangeTask;
   End;
End;
```

If the code is not comprised within an endless loop, when the processor reaches the last end statement the task will go into the suspended state, and will not run again unless it is explicitly activated by another task.
In the previous pseudo-code example, the *CHANGETASK* call triggers the operating system scheduler, leading to a switch from the running task to another.
Three different types of tasks can be defined in Process-Pascal: `CYCLIC`; `TIMEDINTERRUPT` and `SOFTWIREINTERRUPT` tasks. `CYCLIC` tasks have the lowest relative priority (among the three different types) and `SOFTWIREINTERRUPT` tasks have the highest relative priority.
`CYCLIC` tasks are executed in sequence. These tasks are placed on the `CYCLIC` task chain, and executed by an order determined by the order of their definition within the program's source code. Importantly, this kind of task can be pre-empted by any other type of tasks except if it calls the *DISABLE*(*TimedInterrupt*), *DISABLE*(*SoftwireInterrupt*) or *DISABLE*(*Interrupt*) system calls, to disable pre-emption imposed by `TIMEDINTERRUPT` tasks, by `SOFTWIREINTERRUPT` tasks or by both these two types of tasks, respectively. By default, interruptions are enabled in `CYCLIC` tasks. However, interrupts can be explicitly disabled inside a `CYCLIC` task (to disallow pre-emption in a section of the task) and then explicitly enabled to allow pre-emption again, by the use of *ENABLE*(*Type_of_task*) system calls.
When a `CYCLIC` task is pre-empted by a higher priority task (either a `TIMEDINTERRUPT` or a `SOFTWIREINTERRUPT` task), the higher priority task will run until it ends and the `CYCLIC` task will then resume execution from the point of interruption. In the example of Fig. 1, and throughout the rest of the paper, we consider that the time needed to switch from task to task can be neglected.
`TIMEDINTERRUPT` tasks are released at well-defined time instants. At the end of its execution, to switch from a `TIMEDINTERRUPT` task to another type of task, the *CHANGETASK* system call must also be used. A `TIMEDINTERRUPT` task can pre-empt any `CYCLIC` task. Note however that it can not be pre-empted by a `SOFTWIREINTERRUPT` task (the reverse is also valid). In Fig. 1 we illustrate these characteristics with an example set of three `CYCLIC` tasks (allowing pre-emption) and one `TIMEDINTERRUPT` task (obviously periodic).
`SOFTWIREINTERRUPT` tasks are released only when, for instance, there is an access to an internal Process-Pascal global variable (an event). Note that global variables in Process-Pascal can be internal (stored in the local network-node -

controller) or external (stored in another device interconnected by, for instance, a P-NET network). Examples of events that trigger `SOFTWIREINTERRUPT` tasks are keyboard activation or when a remote node reads a local variable. In Fig. 1 we exemplify a scenario where a `SOFTWIREINTERRUPT` task pre-empts `CYCLIC` tasks, but does not pre-empts the `TIMEDINTERRUPT` tasks.

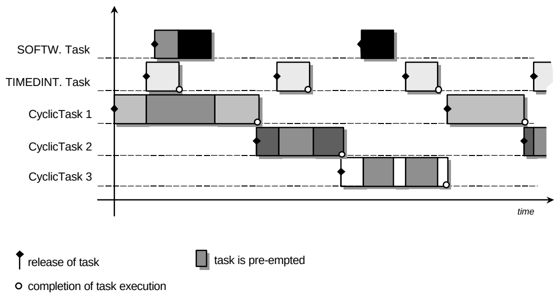

Figure 1 – Interaction between Process-Pascal tasks

To each `SOFTWIREINTERRUPT` task, it is required to define a variable with a specific softwire number (0-31). The task to be released in association to the interrupt will have the same softwire number. The softwire number will define the priority for the associated `SOFTWIREINTERRUPT` task: if two different events occur "simultaneously", the one with the higher softwire number will be processed first.

All tasks are grouped in a task chain system. `CYCLIC` tasks are placed on the cyclic chain list with one task pointing to the next task. `TIMEDINTERRUPT` tasks are placed on another chain list where the order is determined by the next time they will run. Finally, `SOFTWIREINTERRUPT` tasks are in a third chain ordered by its interrupt connections. These will run whenever an event occurs.

When *`CHANGETASK`* is called the next task to run can be the task with the higher priority on the `SOFTWIREINTERRUPT` task chain. If there are no tasks on the `SOFTWIREINTERRUPT` task chain but there are tasks on the `TIMEDINTERRUPT` chain, ready to run, the first will run. If there are no tasks on the `SOFTWIREINTERRUPT` and `TIMEDINTERRUPT` task chains, the next task to run will be the next task on the `CYCLIC` task chain.

3. Response Time Analysis of Tasks in Single Processor Systems

In a single-processor real-time system, one must ensure that all tasks will be schedulable. Basically this means that the response time of any task in the system; that is, the time interval measured from the instant a task is made runnable (is released) to the instant it completes its execution, will not be higher than the acceptable for that task. The maximum response time allowed for a task is usually called the task's relative deadline.

In this section we briefly survey previous relevant work in deriving pre-run-time schedulability analysis for guaranteeing the schedulability of a task set. It is assumed that the task set is composed of independent tasks for which relative deadlines (denoted D) are smaller or equal to the task's periodicity (denoted T). It is also assumed that tasks are scheduled according to the deadline monotonic (DM) [5] priority assignment policy.

3.1. In the Pre-emptive Context

In [6] the authors proved that the worst-case response time R_i of a task i is found when all tasks are synchronously released (critical instant) at their maximum rate. R_i is defined as:

$$R_i = I_i + C_i \qquad (1)$$

In equation (1), C_i corresponds to the worst-case execution time (WCET) of task i. I_i is the maximum interference that task i can experience from higher-priority tasks in any interval $[t, t + R_i)$. The maximum interference (I_i) occurs, when all higher-priority tasks are released synchronously with task i (the critical instant). Without loss of generality, it can be assumed that all processes are released at time instant 0.

The response time can be given by equation 2 where the first term is the interference

$$R_i = \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \right) + C_i \qquad (2)$$

where $hp(i)$ denotes the set of higher-priority tasks (than task i).

Equation (2) embodies a mutual dependence, since R_i appears in both sides of the equation. In fact all the analysis underlay this mutual dependence, since in order to evaluate R_i, I_i must be found, and vice-versa. The easiest way to solve such equation is to form a recurrence relationship [7].

$$W_i^{m+1} = \sum_{j \in hp(i)} \left(\left\lceil \frac{W_i^m}{T_j} \right\rceil \times C_j \right) + C_i \tag{3}$$

The recursion ends when $W_i^{m+1} = W_i^m = R_i$, and can be solved by successive iterations starting from $W_i^0 = C_i$. Indeed, it is easy to show that W_i^m is non-decreasing. Consequently, the series either converges or exceeds D_i (in the case of DM). If the series exceeds D_i, the task t_i is not schedulable.

3.2. In the Non Pre-emptive Context

In [7] the authors updated the analysis of Joseph and Pandya to include blocking factors introduced by periods of non pre-emption, due to the non-independence of the tasks. The worst-case response time is then updated to:

$$R_i = B_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \right) + C_i \tag{4}$$

which may also be solved using a similar recurrence relationship. B_i is the maximum blocking (higher-priority tasks are blocked by lower-priority ones due to non pre-emption) a task *i* can suffer, and is defined as follows:

$$\begin{cases} B_i = 0, & \text{if } P_i = \min\limits_{j=1,..,N} \{P_j\} \\ B_i = \max\limits_{j \in lp(i)} \{C_j\}, & \text{if } P_i \neq \min\limits_{j=1,..,N} \{P_j\} \end{cases} \tag{5}$$

where *lp(i)* denotes the set of lower-priority tasks (than task *i*).
Some care must be taken using equation (4) for the evaluation of the worst-case response time of non pre-emptable independent tasks. In the case of pre-emptable tasks, with equation (2) we are finding the processor's level-*i* busy period preceding the completion of task *i*; that is, the time during which task *i* and all other tasks with a priority level higher than the priority level of task *i* still have processing remaining. For the case of non pre-emptive tasks, there is a slight difference, since for the evaluation of the processor's level-*i* busy period we cannot include task *i* itself; that is, we must seek the time instant preceding the execution start time of task *i*.
Therefore, equation (1) can be used to evaluate the task's response time of a task set in a non pre-emptable context and independent tasks, where the interference must be now re-defined as follows:

$$I_i = B_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{I_i}{T_j} \right\rceil \times C_j \right) \tag{6}$$

4. Response Time Analysis for Process-Pascal Tasks

The response time analysis outlined in the previous section will now be adapted in order to encompass the characteristics of Process-Pascal tasks.
Process-Pascal tasks can be characterised by their type, their worst-case execution time (C_i) and their period (T_i). For a `SOFTWIREINTERRUPT` task, T_i represents the minimum interval between two consecutive releases of the task. For a `TIMEDINTERRUPT` task, T_i is equal to its period, which is defined on the declaration of the function. Finally, for the case of `CYCLIC` tasks, its period will be defined has the minimum time between two consecutive executions. This time will be sum of the best-case execution times for the tasks in the cyclic task chain.
To calculate the maximum response time of a task, we must know the component parts of that response time. Note again that both `SOFTWIREINTERRUPT` and `TIMEDINTERRUPT` tasks are not pre-emptable. If two tasks of these types are launched at the same time, one will have to wait for the other to finish..
In the following sections we will denote *SI* as the set of `SOFTWIREINTERRUPT` tasks, *TI* as the set of `TIMEDINTERRUPT` tasks and *CC* as the set of `CYCLIC` tasks.

4.1. SOFTWIREINTERRUPT Tasks

Consider that a SOFTWIREINTERRUPT task 1 with the highest priority (e.g. 31) is runnable. This task will execute immediately unless there is another SOFTWIREINTERRUPT or TIMEDINTERRUPT task running. As these types of tasks cannot be pre-empted, the new task will wait for the first task to finish its execution, and then starts its execution. The response time of the highest priority task is then the worst-case waiting time to start executing added to its own worst-case execution time.

Considering equation (1), the worst-case response time for that task will happen when the SOFTWIREINTERRUPT task is released just after the release of another SOFTWIREINTERRUPT or TIMEDINTERRUPT task. Thus, $R_1 = I_1 + C_1$, where I_1 is as follows: $I_1 = \max(C_i)$, with $i \in (lpsi(1) \cup TI)$, with $lpsi(1)$ being the set of lower priority SOFTWIREINTERRUPT tasks.

Assume now the SOFTWIREINTERRUPT task with the second highest priority (also released at the critical instant). Firstly, that task will have to wait for the completion of a blocking task. Then there is the interference caused by the highest priority task, and then for any other tasks with the same priority that may already be in the task chain before it starts its execution. This interference may only occur before the task starts its execution, since a SOFTWIREINTERRUPT task cannot be pre-empted. So, the response time of the second highest priority task is $R_2 = I_2 + C_2$, where the interference is given by $I_2 = B_2 + \lceil I_2/T_1 \rceil \times C_1 + \sum_{k \in epsi(2),\ k \neq 2,\ k \neq bi} C_k$. The term within the ceiling function gives the number of times the highest priority task will be executed before the second task is allowed to run. b_i is the task that has caused the initial blocking. B_2 represents the maximum completion time for any lower priority SOFTWIREINTERRUPT task and TIMEDINTERRUPT tasks. e*psi*(2) represents the set of tasks with the same priority as task 2. Finally, B2 is defined as follows: $B_2 = \max(C_i)$, with $i \in (lpsi(2) \cup TI)$.

This reasoning can be generalised for any-priority SOFTWIREINTERRUPT task. The worst-case response time for this type of tasks is given by equation (1), where the interference is defined as follows:

$$I_i = B_i + \sum_{j \in hpsi(i)} \left\lceil \frac{I_i}{T_j} \right\rceil \times C_j + \sum_{k \in epsi(i), k \neq i, k \neq bi} C_k \qquad (7)$$

where B_i is defined as: $B_i = max(C_m), m \in (lpsi(i) \cup TI)$.

In our model we are considering that SOFTWIREINTERRUPT tasks are usually sporadic. In this case T_i has a different meaning than the one for periodic tasks. When referring to sporadic tasks, T_i represents the minimum time between two consecutive executions of the task *i*. This makes our model somewhat pessimistic.

4.2. TIMEDINTERRUPT Tasks

All TIMEDINTERRUPT tasks have the same priority, so the first task to be runnable will be the first task to run, if they are ready to run at the same time. The tasks are stacked on a FIFO queue called timed interrupt task chain. When a TIMEDINTERRUPT task is released and there are only lower priority (CYCLIC) tasks running, the task will run immediately (no waiting period). Therefore, and assuming again equation (1) for the response time analysis, the interference will be $I_1 = 0$. If we consider that there is already a task in the timed interrupt task chain, the waiting time will be $I_2 = C_1$.

Therefore, for generalising, and including the interference resulting from SOFTWIREINTERRUPT tasks (which have all higher priority than TIMEDINTERRUPT tasks), the interference is given by:

$$I_i = \sum_{y \in SI} \left\lceil \frac{I_i}{T_y} \right\rceil \times C_y + \sum_{k \in TI, k \neq i} C_k \qquad (8)$$

4.3. CYCLIC Tasks

This type of tasks can be pre-empted by TIMEDINTERRUPT or SOFTWIREINTERRUPT tasks, which means that they may suffer interference during their whole response time (refer to Section 4 for clarification). This characteristic will have to be included in our models. For this case, it is not possible to divide the response time in waiting time and running time as made in Sections 4.1 and 4.2.

Taking this into consideration, the worst-case response time for a CYCLIC task is given by:

$$R_i = C_i + \sum_{y \in SI} \left\lceil \frac{R_i}{T_y} \right\rceil \times C_y + \sum_{k \in TI} \left\lceil \frac{R_i}{T_k} \right\rceil \times C_k + \sum_{z \in CC, z \neq i} C_i \tag{9}$$

As mentioned in Section 3.1, equations for evaluating worst-case response times are typically mutually dependent equations. This is also the case of equations (7), (8) and (9). Forming a recurrence relationship solves these equations.
For the case of `CYCLIC` tasks (equation (9)), the recurrent relationship will be:

$$R_i^{n+1} = C_i + \sum_{y \in SI} \left\lceil \frac{R_i^n}{T_y} \right\rceil \times C_y + \sum_{k \in TI} \left\lceil \frac{R_i^n}{T_k} \right\rceil \times C_k + \sum_{z \in CC, z \neq i} C_z \tag{10}$$

The first value of the iteration is $R_i^0 = C_i$. It can be proved that set of values R_i^0, R_i^1, R_i^2, …, R_i^n, … is monotonically non-decreasing. When $R_i^{n+1} = R_i^n$, the solution to the equation has been found.
Equations (7) and (8), for the `SOFTWIREINTERRUPT` and `TIMEDINTERRUPT` tasks, respectively, can be solved similarly. The difference is that in these cases the recurrence aims at determining simply the value of the interference (the time instant at which the task under analysis starts execution), whilst in the case of `CYCLIC` tasks we are seeking the time instant when the task completes its execution (it can be pre-empted at any point by `SOFTWIREINTERRUPT` or `TIMEDINTERRUPT` tasks).

5. Response Time for Communicating Tasks

When engineering a real-time system, it is necessary to evaluate the worst-case response time of the complete set of tasks associated to it. When the system is a distributed one, a component of the tasks' response time will be time need by a communicating task to process remote accesses.
For analysing the behaviour of Process-Pascal tasks, we consider two different cases. First, we consider that `CYCLIC` tasks may explicitly disable pre-emption. In a second case we consider that `CYCLIC` tasks allow pre-emption. This is the most important since, as will be later seen, the Process-Pascal task models may impact the analysis for the evaluation of messages' worst-case response time.

5.1. Models for Process-Pascal Communicating Tasks

In Process-Pascal, external variables may represent variables related to other P-NET network nodes. These variables have to be defined with the special keyword *`AT NET`* and with the address of the module, as can be seen on the following extract of code.

```
VAR
DigModule: PD3221 AT NET: (1, 45);
light->
       DigModule.Digital_IO_1.FlagReg[6];
```

This pseudo-code includes the definition of a variable called *`DigModule`* as the entire interface of a P-NET module, which can be accessed by P-NET port 1 (a P-NET gateway can be a multi-port node) and is resident on a P-NET network node of the type PD3221 (slave node) with the network address 45. Then the variable *light* is defined to access a specific bit, bit 7, of a register in the PD3221.
When a task is being executed and it wants to access a variable in a remote node, it simply does: *`a = light`* (to read) or *`light = 1`* (to write), where *`a`* can be a local variable. In effect, this equality operation is not so simple because it involves communications through the P-NET network (sending a request and receiving the related response).
During the communication time, a `CYCLIC` task cannot be pre-empted by other `CYCLIC` tasks, but it can be pre-empted by `SOFTWIREINTERRUPT` or `TIMEDINTERRUPT` tasks.

5.1.1. Case 1: Interrupts are Disabled

In Fig. 2, we show the impact of disabling pre-emption in a communicating `CYCLIC` task. When the `SOFTWIREINTERRUPT` task is released, it can not immediately run because the `CYCLIC` task disabled interrupts. The `CYCLIC` task may disable interrupts in order to perform critical operations, such as being involved in communications. While waiting for the response (from the slave) to the request, the `CYCLIC` task will be blocked. When the response is received the task continues to perform its critical operations and finally enables the interrupts. At this point the `SOFTWIREINTERRUPT` task can run, until completion.

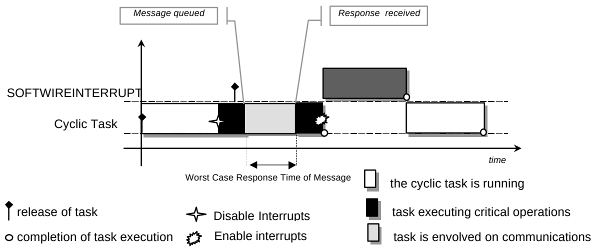

Figure 2 – A cyclic task with the interrupts disabled

5.1.2. Case 2: Interrupts are Enabled

In the case that the `CYCLIC` task does not disable interrupts, this will allow other tasks to pre-empt it even if the `CYCLIC` task is involved in communications. This case permits a better utilisation of the processor, as the `SOFTWIREINTERRUPT` (or the `TIMEDINTERRUPT`) task will be executing while the `CYCLIC` task is blocked waiting for the response from the slave.

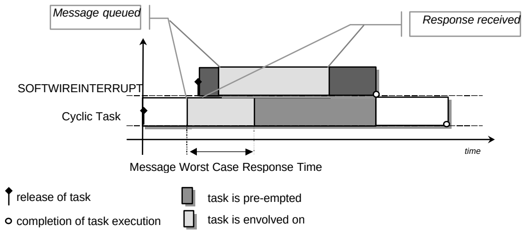

Figure 3 – A cyclic task with the interrupts enabled

The example is illustrated in Fig. 3, where a `CYCLIC` task initiated a communication transaction, queued a message and is waiting for the response. During this period a `SOFTWIREINTERRUPT` task is released and starts execution. Note that this `SOFTWIREINTERRUPT` task will also perform a communication transaction, but as there is already a communication going on, this task will have to wait for the end of the communication transaction by the `CYCLIC` task (refer to Section 5.2. for better understanding this aspect).

5.2. Response Time Evaluation of P-NET Messages

The name P-NET is a derivation of “Process Network”. P-NET was designed as a communications link between distributed process control sensors, actuators and small programmable controllers.

P-NET is a multi-master standard. Therefore, all communication is based on a principle, where a master sends a request and the addressed slave immediately returns a response. For multi-master support, P-NET uses a Virtual Token Passing (VTP) scheme.

The P-NET standard also stands that each master is only allowed to perform one message cycle (a message request from the master followed by the immediate related response from the slave) per token visit. This is an important notion for the remainder of this section.

Assume that C_M is the maximum transmission duration of all message cycles in a P-NET network. This duration includes both the longest request and response transmission times, and also the worst-case slave’s turnaround time.

Therefore, if a master uses the token to perform a message cycle, we can define a token holding time as: $H = r + C_M + t$. In this expression, the symbol t (= 40 bit periods) corresponds to the time to pass the token after a message cycle has been performed. The symbol r (≤ 7 bit periods) denotes the worst-case master’s reaction time. If a station does not use the token to perform a message cycle, the bus will be idle during s (= 10 bit periods). These aspects and the following basic message response time analysis thoroughly explained in [8].

We consider a network with n masters, with addresses ranging from 1 to n. Each master accesses the network according to the VTP scheme. Hence, first master 1, then master 2, 3, … until master 1, and then again 2, 3, … Slaves will have network addresses higher than n. We also assume the following message stream model:

$$S_i^k = (C_i^k, T_i^k, D_i^k) \qquad (11)$$

S_i^k defines a message stream *i* in master *k* (*k* = 1, .., *n*). A message stream is a temporal sequence of message cycles concerning, for instance, the remote reading of a specific process variable. C_i^k is the longest message cycle duration of stream S_i^k. T_i^k is the periodicity of stream S_i^k requests. Finally, D_i^k is the relative deadline of the message cycle, that is, the maximum admissible time span between the instant when the message request is placed in the outgoing queue and the complete reception of the related response at the master's incoming queue. We consider that messages generated in the distributed system can be periodic or sporadic. For the case of sporadic message requests, its period corresponds to the minimum time between any two consecutive requests for that stream. ns^k is the number of message streams associated with a master *k*.

In this model, the relative deadline of a message can be equal or shorter than its period ($D_i^k \leq T_i^k$). Thus, if in the outgoing queue there are two message requests from the same message stream, this means that a deadline for the first of the requests was missed. It also results that the maximum number of pending requests in the outgoing queue will be, in the worst-case, ns^k.

We denote the worst-case response time of a message stream *i* in a master *k* as R_i^k. This time is measured starting at the instant when the request is placed in the outgoing queue, until the instant when the response is completely received at the incoming queue. Basically, this time span is made up of the two following components: the time spent by the request in the outgoing queue, until gaining access to the bus (queuing delay) and the time needed to process the message cycle, that is, to send the request and receive the related response (transmission delay).

Thus,

$$R_i^k = Q_i^k + C_i^k \qquad (12)$$

where Q_i^k is the worst-case queuing delay of a message stream *i* in a master *k*.

In order to have simpler and more understandable analysis, we will use the maximum token holding time ($H = r + C_M + t$) for all message cycle transactions, instead of considering the actual length for each particular message cycle. Thus, in equation (13), C_i^k is replaced by C_M.

A basic analysis for the worst-case response time can be performed if the worst-case token rotation time is assumed for all token cycles (in [9], the authors developed a more sophisticated analysis by considering the actual token rotation time).

As the token rotation time is the time span between two consecutive visits of the token to a particular station, the worst-case token rotation time, denoted as *V*, is: $V = n \times H$, which gives the worst-case time interval between consecutive token visits to any master *k* (*k* = 1, .., *n*).

In P-NET, the outgoing queue is implemented as a first-come-first-served (FCFS) queue. Therefore, a message request can be in any position within the ns^k pending requests. ns^k is also the maximum number of requests which, at any time, are pending in the master *k* outgoing queue. This results from the adopted message stream model, which considers $D_i^k \leq T_i^k$. Hence, the maximum number of token visits to process a message request in a master *k*, is ns^k. The worst-case queuing delay occurs if ns^k requests are placed in the outgoing queue just after a message cycle was completed.

Based on these assumptions, in [9] the authors prove that the worst-case response time for a P-NET request is given by:

$$R^k = ns^k \times V = ns^k \times n \times H = ns^k \times n \times (r + C_M + t) \qquad (13)$$

5.3. Holistic Analysis

For the evaluation of the worst-case response time (WRCT) of the tasks it is important to note that the worst-case execution time (*C*) of the tasks includes a portion concerning the communication response time (*R*).

In the remainder of this section we show, for the two referred cases (allowing and not allowing pre-emption of `CYCLIC` tasks), that there is an important influence of the Process-Pascal task model in the evaluation of messages' response time (equation (13)).

Additionally we update response time analysis of `SOFTWIREINTERRUPT` and `TIMEDINTERRUPT` tasks (Sections 4.1 and 4.2, respectively) to include periods of non pre-emption in `CYCLIC` tasks. The analysis is specifically updated taking into account the message passing mechanisms.

5.3.1. Case 1: Interrupts are Disabled

Equations (7) and (8) must be changed in order to include the blocking time due to the disabling of interrupts by `CYCLIC` tasks.

Therefore, in the case of `SOFTWIREINTERRUPT` tasks the calculation of the blocking time is: $B_i = max(C_i, B_{cc}), i \in (lpsi(i) \cup TI)$, where B_{CC} is the longest blocking time due to the non pre-emptive period of any `CYCLIC` task.

For TIMEDINTERRUPT tasks, we will have also to consider the impact of non pre-emptive periods in CYCLIC tasks (B_{CC}). As TIMEDINTERRUPT tasks are not pre-emptable, this term must be considered in the equation for the evaluation of the interference. Therefore, equation (8) is updated to:

$$I_i = B_{CC} + \sum_{y \in SI} \left\lceil \frac{I_i}{T_y} \right\rceil * C_y + \sum_{k \in TI, k \neq i} C_k \qquad (14)$$

An important result of not allowing pre-emption in any task (note that both TIMEDINTERRUPT and SOFTWIREINTERRUPT tasks are not pre-emptable) is that there can only be one message at a time waiting to be transmitted in a master *k* outgoing queue.
Therefore, and for this scenario, equation (14) will result in $R^k = V$. This result can be used to evaluate the maximum blocking time for a task due to communication delays. This time can be incorporated to obtain a parcel of the blocking time, B_{CC}, of a SOFTWIREINTERRUPT or TIMEDINTERRUPT task.

5.3.2. Case 2: Interrupts are Enabled

In this case, a CYCLIC task can be pre-empted. However, there is still an additional blocking time in both SOFTWIREINTERRUPT or TIMEDINTERRUPT tasks, due to the fact that if they have message cycles to perform, they may have to wait for the completion of a message cycle previously initiated by a CYCLIC task (refer to Fig. 3).

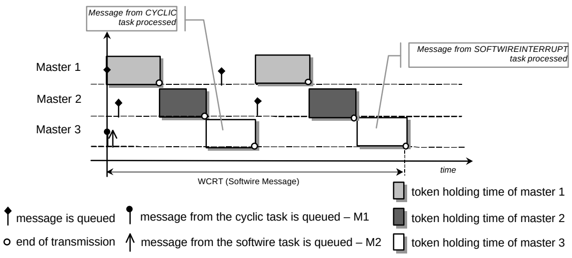

Figure 4 – Messages on the network

As SOFTWIREINTERRUPT and TIMEDINTERRUPT tasks cannot interrupt each other there will be at most two pending requests (one from the cyclic task and another from the higher priority task) on the output communication queue. Take the example of Fig. 4 and assume the case that in master 3 there are two tasks: a CYCLIC task and a SOFTWIREINTERRUPT task.
The worst case happens when master 3 as just finished transmitting a message and a CYCLIC task queues a message (M1). Then the CYCLIC task is pre-empted by a SOFTWIREINTERRUPT task, which queues another message (M2).
As P-NET uses a FCFS communication queue, the message from the SOFTWIREINTERRUPT task will have to wait for the transmission of the message that is already in the communication queue. As in P-NET a master is only able to process a message cycle per token visit, M2 will only be transmitted after 2 token visits to master 3.
Therefore, and for the evaluation of the tasks' response time, in the case of CYCLIC tasks, the messages' response time will be $R^k = V$, whilst for the other two types of tasks will be $R^k = 2 \times V$.

6. Conclusions

The problem of engineering real-time distributed applications is a complex one. A potential leap towards the use of fieldbus in such time-critical applications lies in the evaluation of its temporal behaviour.
In the past few years several research works have been performed on a number of fieldbuses. However, these have mostly focused on the message passing mechanisms, without taking to account the real implementations of those communication protocols, and emphatically without taking into account the application development tools for those distributed systems. The main contribution of this paper was to provide an application software perspective for engineering real-time with fieldbus networks. We address the case of P-NET fieldbus networks and the Process-Pascal tool to develop P-NET based distributed applications.
Importantly, we have developed worst-case response time analysis for the actual tasks that are executed in P-NET networks and integrated this analysis with the worst-case response time analysis of P-NET network messages.
In this way, we provide an important set of analysis for engineering real-time distributed applications with P-NET networks using the natural system developer's perspective: an application software perspective.

Also an important result was to show how the timing analysis performed merely at the message level can be influenced in its assumptions when application task models (communicating tasks) are brought into consideration.

7. References

[1] Prince, S. and Solomon, M. (1981). Communication Requirements for a Distributed Computer Control System. In *IEE Proceedings,* Vol. 128, No. 1, pp. 21-34.

[2] Tindell, K., Burns, A. and Wellings, A. (1995). Analysis of Hard Real-Time Communications. In *Real-Time Systems,* 9, pp. 147-171.

[3] Cenelec (1996). General Purpose Field Communication System. EN 50170, Vol. 1/3 (P-NET), Vol. 2/3 (PROFIBUS), Vol. 3/3 (FIP), Cenelec.

[4] Process-Data A/S (1999). Process-Pascal, Users Manual, Version 4. Process-Data A/S.

[5] Leung, J. and Whitehead, J. (1982). On the Complexity of fixed-priority Scheduling of Periodic Real-Time Tasks. In *Performance Evaluation*, Vol. 22, No. 4, pp. 237-250.

[6] Joseph, M. and Pandya, P. (1986). Finding Response Times in a Real-Time System. In *The Computer Journal*, Vol. 29, No. 5, pp. 390-395.

[7] Audsley, N., Burns, A., Richardson, M., Tindell, K and Wellings, A. (1993). Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. In *Software Engineering Journal*, Vol. 8, No. 5, pp. 285-292.

[8] Tovar, E., Vasques, F. and Burns, A. (1999). Supporting Real-Time Distributed Computer-Controlled Systems with Multi-hop P-NET Networks. In *Control Engineering Practice*, 7(8), pp. 1015-1025.

[9] Tovar, E., Vasques, F. and Burns, A. (1999). Communication Response Time in P-NET Networks: Worst-Case Analysis Considering the Actual Token Utilisation. Department of Computer Science, University of York, Technical Report YCS 312, to appear in *Journal of Real-Time Systems*.

Acknowledgements

This work was partially funded by FLAD, under project SISTER 471/97, by FCT, under project DEAR-COTS Praxis/C/EEI/14187/98 and by ISEP.