ASTEC

Advanced Software Technology

FIXED PRIORITY SCHEDULING ANALYSIS OF THE POWERTRAIN MANAGEMENT APPLICATION EXAMPLE USING THE SCHEDULITE TOOL

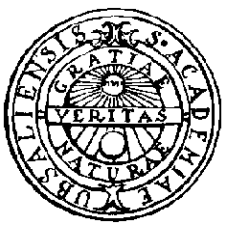
Jens Larsson

t91jla@docs.uu.se

Technical Report
ASTEC 97/03
DoCS 97/82

Department of Computer Systems
Uppsala University

February 20, 1997



DoCS

ABSTRACT

This report presents how the fixed priority scheduling analysis tool ScheduLite can be used to perform scheduling analysis of a distributed hard real-time system. The analysed system is a Powertrain Management application which is an example of a hard real-time system embedded in a car. Four nodes and a communication link is the hardware architecture used to implement functions such as anti-spin control, ABS and a clutch free gearbox. It is showed how processes and inter process relations specified in the system are represented in the tool. The result of the analysis is presented with tables including calculated time parameters. Further more, it is suggested how the ScheduLite tool could be extended with new features to even better support scheduling analysis of distributed real-time systems.

PREFACE

This report is done in connection to my Master of Science thesis work in engineering physics. The work was performed at the Department of Computer Systems, at Uppsala University, under supervision of Hans Hansson.

The project is performed together with Mecel AB within the ASTEC (Advanced Software TEChnology) centre of excellence supported by NUTEK (Swedish National Board for Technical and Industrial Development). Support has also been provided by Volvo Science Foundation.

This report was written in Microsoft Word 6.0 under Windows 3.11.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Hans Hansson for proof-reading this document. I would also like to thank Sven Larsson, Magnus Lindahl and Mikael Strömberg at Mecel in Göteborg for developing the Powertrain Management application example.

CONTENTS

1. INTRODUCTION 8

2. THE POWERTRAIN MANAGEMENT APPLICATION EXAMPLE 9

2.1 THE SYSTEM ARCHITECTURE 9
2.2 PROCESSES 9

3. THE SCHEDULITE TOOL 11

4. SCHEDULING ANALYSIS OF THE CASE STUDY USING SCHEDULITE 13

4.1 REPRESENTING INTER TASK RELATIONS 13
4.1.1 Local Asynchronous 13
4.1.2 Local Synchronous 13
4.1.3 Distributed Asynchronous 14
4.1.4 Distributed Synchronous 15
4.2 DESCRIPTION OF THE WORK 15

5. FURTHER DEVELOPMENT OF SCHEDULITE 19

5.1 PRIORITY ASSIGNMENT 19
5.2 SUPPORT OF INTER PROCESS RELATION REPRESENTATION 20

6. RESULTS OF THE SCHEDULING ANALYSIS 22

6.1 ANALYSIS RESULT TABLES 22
6.1.1 MMICU 22
6.1.2 ECU 24
6.1.3 GCU - Task Set and Semaphores 26
6.1.4 ABSCU- Task Set and Semaphores 26
6.1.5 CAN 26
6.2 PRECEDENCE CHAINS 27
6.2.1 Precedence Chains for Processes 27
6.2.2 Precedence Chains for Messages 29

7. REFERENCES 34

1. INTRODUCTION

Real-time computing systems are becoming more and more important in the technological infrastructure of industrialised nations. Modern telecommunication systems, factories, spacecraft, aircraft and airports cannot operate without them. Real-time computing systems are also introduced in new fields, such as the automotive industry, to optimise performance and costs. For these systems to behave correctly it is necessary for different applications in the system to share resources (e.g. CPUs and networks) in a predictable and cost efficient way. Therefore it would be good for the system designers to have a design model that includes methods for verification and testing of the systems. One part is to predict whether process deadlines will be met before the costly implementation phase of a project begins. For this reason different scheduling methods and theories have been developed, for example static cyclic scheduling, generalised rate monotonic scheduling and fixed priority scheduling.

For scheduling analysis of a fixed priority scheduled distributed hard real-time system a tool called ScheduLite is developed. This report describes how ScheduLite can be used to analyse a case study called the Powertrain Management application example.

Brief descriptions of the Powertrain Management application example and the tool ScheduLite are found in Section 2 and Section 3 respectively. Section 4 describes how the application example is represented in ScheduLite and how to run the analysis. The contents of the analysis result tables in ScheduLite are presented separately in Section 6. Section 5 presents some ideas how ScheduLite could be extended with features to further simplify scheduling analysis of distributed hard real-time systems such as the Powertrain Management application example.

2. THE POWERTRAIN MANAGEMENT APPLICATION EXAMPLE

Within ASTEC* the project ART† is aiming at finding a practical working design model of hard real-time systems. The design model should support verification at each level of the design. The project is a cooperation between DoCS - Department of Computer Systems at Uppsala University and Mecel AB, Göteborg. A case study is used with the purpose of developing and testing the design model. The case study is an application example called Powertrain Management, which is an example of a hard real-time system embedded in a car. With four nodes and one communication bus it is suggested how functions such as ABS-brakes, anti spin and an automatic clutch free gearbox could be implemented. The system specification is given in [1]. We will now briefly describe the Powertrain Management application example.

2.1 THE SYSTEM ARCHITECTURE

The Powertrain Management application example has four different nodes (or CPUs): the Engine Control Unit (ECU), the Gearbox Control Unit (GCU), the Anti Block System Control Unit (ABSCU) and the Man Machine Interface Control Unit (MMICU) (see Figure 2-1). All nodes are connected through a communication link, e.g. a CAN-bus [3, 4]. The physical location of each CPU should be close to it's sensors and actuators.

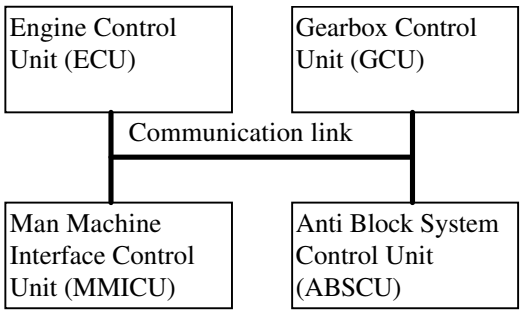


Figure 2-1 The system architecture.

2.2 PROCESSES

The functionality of the system is realised through different processes. In the terminology used in the application example a process can consist of one or more *software circuits*. Software circuit is a term orginating from the Basement concept [5]. A process implements a sub-function, for instance the Cruise Control process or the Throttle Control process. A process has a period time, i.e. all software circuits in the same process have the same period time. An end-to-end deadline can be defined for each process, but the deadlines of the processes are not derived in the current description of the application example [1].

There can be different types of relations between software circuits: asynchronous or synchronous as well as local or distributed. In [6] four different types of communicator

* ASTEC - Advanced Software TEChnology competence centre.
† ART - ASTEC Real-Time.

object are introduced. The communicators are used to make an abstraction of the following different relations: local asynchronous, distributed asynchronous, local synchronous and finally distributed synchronous.

The relations between the software circuits in a process are synchronous. The software circuits may have asynchronous relations with software circuits in other processes. A synchronous relation also means that there exists a precedence constraint, i.e. the software circuit producing a value has to finish before it can be consumed by another software circuit in the process.

In this application we have only local processes, i.e. we have local synchronous relations but no distributed synchronous relations. There exists both local and distributed asynchronous relations.

Some process has to be allocated at a certain node since it needs to access sensors or actuators located at that node. These processes are called static allocateable. Other processes do not have such a constraint and can be allocated at any node with enough computing capacity. Those processes are called dynamic allocateable.

3. THE SCHEDULITE TOOL

A description of the ScheduLite tool can be found in [2]. We will here give an overview of the abilities of the tool.

The application example described above uses the terms software circuit and process. The ScheduLite tool is based on Fixed Priority Scheduling [2, 7, 8, 9, 10, 11, 12]. In the terminology used together with Fixed Priority Scheduling a software circuit corresponds to a *task* (or *process*) and a process corresponds to a *precedence chain* or a *transaction*.

The ScheduLite tool is an application that runs under Microsoft Windows 3.11 and later versions. ScheduLite does scheduling analysis of distributed hard real-time systems where the fixed priority scheduling algorithm is used to schedule the tasks in the system. The tool has a graphical user interface that supports input of the hard real-time system and output of the analysis results. Graphs and tables are used to visualise the results.

When entering a hard-real time system the resources are first created, i.e. the CPUs and the communication link. For the communication link the analysis of an ideal CAN-bus is implemented in the current version of the tool. For each resource the tasks allocated at the resource are created. A message sent over the CAN-bus is also treated as a task in the tool. Each task has timing parameters that also has to be entered, e.g. period time T, worst-case execution time C and deadline D. For a CAN-message the message size s in bytes is given instead of C. The worst-case transmission time is then calculated by ScheduLite. Each CPU can have semaphores protecting shared objects located at the CPU. A shared object can for instance be a shared memory buffer. While a process is accessing a shared object it has to lock the semaphore protecting the object. To be able to perform the scheduling analysis ScheduLite needs information about how long time a process is locking a certain semaphore, so this information has to be entered.

For tasks sharing precedence constraints precedence chains have to be created in the tool. The tool then checks that the precedence constraints are met and no precedence contradictions are found in the precedence chains entered. A task sending a message, the message itself and the task receiving the message should always be included in a precedence chain since the tool uses this information to derive inherited jitter of the message and the task receiving the message.

For dynamic allocatable tasks it has to be defined at which CPUs the task is allowed to execute together with resource dependent information, i.e. worst-case execution time and locking time for semaphores.

When the system is entered the analysis can be run. The worst-case response time is then computed for each task in the system. The results are presented in a table where response times exceeding the deadlines are displayed with red coloured characters to alert that the system is not schedulable. The response times can also be presented in a bar graph. The utilisation of the different resources are calculated and presented in a graph together with a pie chart showing how much each task is loading the resource.

In this way ScheduLite helps the system designer to determine how an unschedulable system in the best way can be made schedulable.

4. SCHEDULING ANALYSIS OF THE CASE STUDY USING SCHEDULITE

This section describes how ScheduLite can be used to perform scheduling analysis of the Powertrain Management application example.

4.1 REPRESENTING INTER TASK RELATIONS

As we said earlier we have four types of inter task relations: asynchronous or synchronous as well as local or distributed. We will here describe how these can be represented in ScheduLite by using a simplified structure of the communicators described in [6].

4.1.1 LOCAL ASYNCHRONOUS

The local asynchronous inter task relation, or with the Powertrain Management terminology an asynchronous connection through a channel between two software circuits, can be realised with a shared memory buffer protected by a semaphore. The producing task (software circuit) writes data to the buffer while it is locking the semaphore. The consuming task or tasks locks the semaphore while the data is read. By using the semaphore the tasks are mutual excluded and the contents of the shared memory buffer can be kept consistent. It should be noted that certain data can be produced by only one task but consumed by one or more tasks.

In ScheduLite we just have to create a semaphore, preferable using the channel name (the name of the data or signal produced) as identifier. We do then need to know how long time each task accessing the shared buffer is locking the semaphore, i.e. we need to know the worst-case execution time of the task's *critical section*. A critical section is the part of the code implementing the task where a semaphore is locked. The locking time of each semaphore held by a task is entered into tool. This information is used by the tool to compute the blocking factors of the tasks. The blocking factor of a task is the maximum time the task can get blocked by a lower priority task when it tries to lock a semaphore.

4.1.2 LOCAL SYNCHRONOUS

When we have local synchronous relation between two tasks we have to create a precedence chain with the producing task as the first member of the precedence chain and the consuming task as the second member. The communication can also here be realised by using a shared memory buffer. In general we should have a semaphore to protect the buffer even here, but if there are only one producing task and one consuming task that share the buffer we do not need a semaphore since the tasks have a precedence constraint preventing them to access the buffer at the same time. If we anyway let the tasks lock a semaphore, the analysis will compute a blocking factor for the task with the higher priority of the two as if it could be blocked by the task lower priority task. The precedence constraint tells us that this cannot happen and hence the analysis will be slightly pessimistic. In a local precedence chain of two tasks it is must be the second task that has the lowest priority to fulfil the precedence constraint (if not offsets are used).

A process in the Powertrain Management consists of software circuits with local

synchronous relations. To represent a process in ScheduLite we group tasks into a precedence chain. A process with branched synchronous relations corresponds to a branched precedence chain. A branched precedence chain must in ScheduLite be transformed to a set of precedence chains without branches. An example of a transformation is seen in Figure 4-1 below where chain *1* is transformed into the chains *2* and *3*. The chains *2* and *3* together represents the same constraints as chain *1* alone.

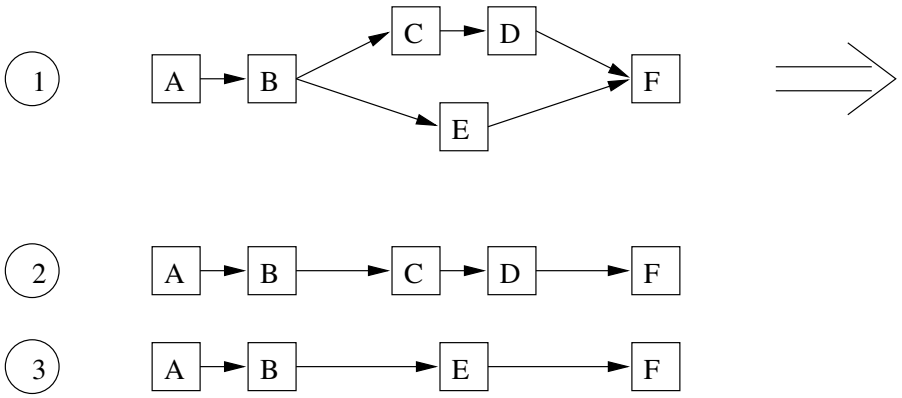


Figure 4-1 Transformation of a branched precedence chain into two unramified chains.

4.1.3 DISTRIBUTED ASYNCHRONOUS

A distributed asynchronous inter task relation has to be realised through a message sent over the CAN-bus. A sending task, the message and the receiving task always has a synchronous relation. This means that we can chose to put the asynchronous point either at the node of the sending task or at the node of the receiving task. (Remember that a sending task, the message and the receiving task always have to be put in a precedence chain to make it possible for ScheduLite to let the message and the receiving task to inherit jitter properly.)

If we place the asynchronous point at the receiving node we just let the producing task send a message each time a new value is produced. At the receiving node we create a server task that receives the message and puts the data in a buffer that it shares with the consuming task or tasks. The message and the receiving server task inherits the period time of the sending task. The tasks accessing the buffer lock the semaphore protecting the buffer as described in Section 4.1.1, i.e. we have a local asynchronous relation between those tasks.

If we instead place the asynchronous point at the sending node we create a server task that sends the message. The producing task places the data in a shared buffer according to the local asynchronous relation and the server task reads the buffer and sends the message. The consuming task is then triggered by the arrival of the message and receives the data in a synchronous way. This means that the sending server task, the message and the receiving consuming task has the same period time. We do here have a problem if the consuming task also have distributed relations with other tasks. In that case there is more than one message sent to the consuming task and we will have problems to determine what message that should trigger the task. We cannot

either determine from what message the task should inherit jitter. Because of these problems it is more general to put the asynchronous point at the receiving node.

If we can choose where to place the asynchronous point we should consider the period times of the producing task and the consuming task. Let us say that the producing task generates a new value each millisecond ($T_{prod}=1\ ms$). The value needs to be updated in this rate because of a synchronous relation with a local task. The producing task also has a asynchronous relation with a remote task allocated at another node. The remote task has a period time of one second ($T_{remote}=1000\ ms$). If we now place the asynchronous point at the node of the remote task a message is sent each $T_{prod}=1\ ms$, but only one of thousand values is used by the remote task. This will load the communication link more than needed. In this case it is better to have the asynchronous point at the producing task and send the message as often as the value is needed by the remote task. If we have the reversed situation that the producing task has a period time of one second ($T_{prod}=1000\ ms$) and the remote consuming task a period time of one millisecond ($T_{remote}=1\ ms$) then it is better to have the asynchronous point at the node of the remote task. Otherwise we will send the same value over the bus thousand times.

4.1.4 DISTRIBUTED SYNCHRONOUS

A distributed synchronous inter task relation could be realised by just letting the producing task send a message directly to the remote consuming task. The message and the consuming task are then inheriting the period time of the producing task. A consuming task cannot have distributed synchronous relations with more than one producing task since we then will have more than one message sent to the task and we will have triggering problems and attribute inheriting problems in the same way as described in Section 4.1.3. On the other hand a producing task can have distributed synchronous relations with more than one task, i.e. a task can trigger, but not be triggered by, more than one remote task.

A CAN-bus message can be sent to more than one node at the same time. This means that have the following precedence chains defined in the system (using the notation Nodename.Taskname):

1. `CPU1.SendM1` → `M1` → `CPU3.RecM1`
2. `CPU1.SendM1` → `M1` → `CPU2.RecM1`

The message `M1` inherits jitter from the sending task. In this case the sending task is the same in both precedence chain 1 and 2, so there is no contradiction from where to inherit this jitter. A contradiction will on the other hand occur if we have two different tasks sending the same message or if we have one task receiving two different messages. The analysis would in this case probably not converge since it is not distinct from where to inherit jitter. The last situation could be resolved by creating a server task for each message to be received.

4.2 DESCRIPTION OF THE WORK

This section describes the working method when doing scheduling analysis of the Powertrain Management application example.

First of all the four nodes MMICU, ECU, GCU and ABSCU was created. To represent the communication link a CAN-bus object was created. In the application example there are static allocateable processes and dynamic allocateable processes. We started with the static allocateable processes. For each software circuit a task was created. In the specification of the application example the period time, *T*, was given for each processes together with the worst-case execution time, *C*, for each software circuit. The process deadlines were not given since they are not yet derived. Therefore deadlines equal to the period time were used (*D*=*T*).

For each process consisting of more than one software circuit a precedence chain was created in ScheduLite to represent the synchronous relations between the software circuits (tasks). If the process had branched relations it was transformed to a set of precedence chains as described in Section 4.1.2.

The system analysis was then run with all the static allocateable processes to give a hint about the load balance between the different nodes before allocating the dynamic allocateable processes. We had the following result:

Node	**Utilisation**
MMICU	0,95%
ECU	28,10%
GCU	0,38%
ABSCU	1,80%

Table 4-1 Resource utilisation with dynamic allocateable processes excluded.

Just to check the load from the dynamic allocateable processes a temporary node was created where the software circuits (tasks) of the dynamic processes were entered. The utilisation of that temporary node was computed by the tool to be 18,42%.

Now it was time to decide where to allocate the dynamic allocateable processes. Since the processor load seemed to be low we could allocate them at any node. The MMICU had only a load of 0,95%, so by practical reasons we decided to allocate all dynamic allocateable processes there.

In the application example all relations between software circuits that are members of different processes are asynchronous relations. Since the allocation of the processes now was decided we could check whether those asynchronous relations were local or distributed. A local asynchronous relation was represented by creating and locking a semaphore as described in Section 4.1.1. Since we did not have any information about the length of the critical sections of the tasks we assumed that each task using a semaphore held it for 0,01 *ms*. Each distributed asynchronous relation was represented as described in Section 4.1.3, so for each such relation a message was created at the CAN-bus resource. We choose the more general method of placing the asynchronous point at the node of the receiving task. Therefore we had to create a server task receiving the message and a semaphore to handle the interaction between the server task and the consumer task(s) for each distributed asynchronous relation. The worst-case execution time of a server task receiving a message was assumed to be 0,05 *ms*. A precedence chain for each message was created, where the sending task, the message and the receiving task are members.

The utilisation of the CAN-bus depends on the bus speed used. The maximum speed of a CAN-bus is 1 *Mbit/sec*. With the bus speed 250 *kbit/sec* (equals to a bit time of τ_{bit}*=0,004ms*) the total utilisation of the bus was calculated to 82,68%. The utilisation of the different resources we retrieved is presented in the table below.

Node	Utilisation
MMICU	13,12%
ECU	39,3%
GCU	2,38%
ABSCU	2,80%
CAN	82,68%

Table 4-2 Resource utilisation with all processes and messages included.

Comparing Table 4-1 with Table 4-2 the increase in utilisation of the GCU and the ABSCU depends only on added server tasks receiving messages. For the MMICU the increase depends both on added server tasks and that the dynamic allocateable tasks are allocated at the MMICU. For the ECU there are server tasks added too, but we have also made one of the dynamic allocateable tasks static allocated to the ECU, namely the Engine Speed Algorithm software circuit (task) that is the only member of the Engine Speed Control Process. The Engine Speed Algorithm task is close related to the Engine Power Algorithm which is a member of the Engine Torque Control Process. The Engine Speed Algorithm generates the same control signals as the Engine Power Algorithm, i.e. Req_fuel, Req_InjectionPos and Req_ThrottlesPos. Which of the two algorithms that is active is decided from the signal Req_EngineRegMode. Since the Engine Power Algorithm is statically allocated at the ECU it seems logical that the Engine Speed Algorithm also has to be allocated at the ECU instead of being a dynamic allocateable process.

The tasks have been assigned their priorities according to the deadline monotonic algorithm, which is automatically done by ScheduLite if this option is chosen. Precedence constraints between local tasks could be met by using the priority mechanism. If the priorities decrease monotonically with the order in the precedence chain (the first task has the highest priority and the last task the lowest priority) the execution order will be correct if the tasks in the precedence chain are released simultaneously. Since the process deadlines are not yet specified in the application example, and we here use the process end-to-end deadlines equal to the period time, we do not have any information about the deadlines of the tasks earlier in a precedence chain. For local precedence chains the earlier tasks were given slightly shorter deadlines just in order to meet the precedence constraints. Another thing that should be mentioned is that ScheduLite produces a pessimistic analysis if tasks are assigned the same priority. This is the case here since there are processes with the same period time and we have assigned the tasks deadlines according to the process period time. The best is if each task has a unique priority. We will return to this problem in Section 5.

The result of the scheduling analysis of the application example done with ScheduLite is presented in tables in Section 6 where the analysis result tables of the different resources in the system are presented together with tables of the precedence chains in the system. We can see that the system is schedulable since no deadlines are missed. As

said earlier we have consequently chosen to put the asynchronous point at the node of the receiving task since this is a more general method (see Section 4.1.3). This causes the CAN-bus to be more loaded than necessary. In Section 6.2.2 it is pointed out for what messages it is possible to reduce the load of the bus by placing the asynchronous point at the node of the sending task. Running the analysis of the system in this situation shows that the bus utilisation will be reduced from 82,68% to 56,85% at the bus speed 250 *kbit/sec*.

5. FURTHER DEVELOPMENT OF SCHEDULITE

The work with the application example together with the ideas presented in [6] have generated some ideas on how the ScheduLite tool can be extended with new features to better perform the scheduling analysis. The new features would make it possible to represent the system on a higher level in the tool. We will however start to discuss priority assignment of the tasks.

5.1 PRIORITY ASSIGNMENT

A priority assignment algorithm commonly used is the deadline monotonic algorithm. This means that the tasks in a task set are assigned priorities monotonically to the deadlines of the tasks: the shorter deadline the higher priority. Now it often happens that the deadlines specified are defined end-to-end. For example, a task may sample a sensor and then send a message to a task allocated at another node. The receiving task may take some actions, for instance activating an actuator. A common requirement on this chain of tasks is to have a upper limit on how long time this chain should take to execute, i.e. an end-to-end deadline. The deadlines of each task in the chain are often not given. This causes at least two problems. Since the deadlines also may decide the priorities of the tasks it is important to have some value assigned to tasks. If we for instance assign all the tasks in the chain the end-to-end deadline the first task in the chain may get a too low priority on it's node causing the whole chain to miss it's end-to-end deadline. This is an optimisation problem not easy to solve for larger systems. Another problem is if we have a local precedence chain where we let the priorities control the execution order of the tasks. If we use deadline monotonic priority ordering and we only know the end-to-end deadline the deadlines of the earlier tasks have to be derived in some way, at least in a way that ensures the correct execution order. We will come back to a possible solution.

A problem related to the current version of ScheduLite is that if more than one process has the same priority value, which may happen if we use for instance deadline monotonic, is that the analysis will be pessimistic. The tool assumes that two tasks with the same priority both interfere with each other. If they have unique priorities (which should be the case in simple fixed priority scheduling) only one task will be interfered by the other. Tasks with the same priority can be assigned a unique priority that separates them in an arbitrary way among themselves. This means that a simple solution to this problem is to sort all task in priority order (for instance in deadline monotonic order) and place them in a list. Each task could then be assigned a number according to the ordering number of the task in the list. In this way tasks with the same priority would get a unique priority. A drawback with this solution that it may generate many priority levels which can cause implementation problems.

The above idea could be used to solve the problem with the execution order of tasks in a local precedence chain discussed above. First we simply assign the end-to-end deadline to all tasks in a precedence chain. The local tasks of the chain will first have the same priority. Then we sort the tasks in the deadline monotonic order. Before we assign the tasks their ordering number as the priority we use the information in the precedence chains to sort the tasks with the same deadline in a order that fulfils the precedence constraints. This method is of course not optimal in a holistic view (optimal

holistic priority assignment is a NP-hard problem), but it is a practical way to generate priorities that at least meet the precedence constraints.

5.2 SUPPORT OF INTER PROCESS RELATION REPRESENTATION

In Section 4.1 we described four types of inter process relations and how they could be represented in ScheduLite. When we entered the system into the tool we had to create the representation of each relations manually, i.e. semaphores, messages and server tasks were created. If we decide to move a dynamic allocateable process to another node some local relations may turn distributed and vice versa. We do then need to create or delete semaphores and messages from the system. This will be a tedious work if we are searching for an optimal allocation of dynamic allocateable tasks. Therefore it would be good if the tool could support this work. We will now give a suggestion of a structure that could be used to extend the tool with this feature. The structure uses the communicator objects described in [6]. There exist four types of communicators: local asynchronous, distributed asynchronous, local synchronous and distributed synchronous. The communicator object encapsulates the functions that implements the inter process relation, for instance tasks reading and writing shared buffers, messages and tasks sending and receiving messages.

An important observation that will be used is that a certain signal or certain data is generated by one and only one task, but may be consumed by one or more tasks. We could say that each output creates a logical channel from where other tasks can get information. A task may have more than one output, but each output corresponds to a unique channel. When another task is consuming information from the channel a relation is established between the producing task and the consuming task. This relation can, as described earlier in Section 4.1, be asynchronous or synchronous depending on what functionality the relation should realise and is decided by the system designer. The current allocation of the tasks in the relation decides if it is a local or distributed relation. The idea is now to have a relation object for each relation established in the system which contains information about the local as well as the distributed version of the relation, i.e. it has one local and one distributed communicator object. The communicator objects holds parameters like length of critical sections and message sizes. Depending on the current allocation of the tasks the tool could then automatically use information from the proper communicator and create semaphores and messages and server tasks needed for the moment. When a message is sent we have seen in Section 4.1.3 that the sending task, the message and the receiving task have to be members of a precedence chain to make it possible for the tool to derive inherited jitter. If we have a local asynchronous relation that becomes distributed the tool can just create a precedence chain for the message. If it is a local synchronous relation that becomes distributed the tool can just insert the sending task, the message and the receiving task into the already existing precedence chain (if a relation is synchronous a precedence chain must be defined).

The structure to use could be like this:

1. For each output existing in the system a channel object is created.
2. For each task consuming the information in the channel (i.e. having the channel as input) a relation object is created. The relation object is either of type synchronous or asynchronous.

3. The channel object holds a list of the relation objects.
4. The relation object holds the local and distributed communicator. The type of the communicators (asynchronous or synchronous) is the same as the type of the relation object. If both tasks in the relation are statically allocateable it is enough to have just one communicator (local *or* distributed).

The structure suggested above means that we are close to have information about the architecture of the system, i.e. the outputs and inputs of the tasks and how they are connected. In fact we will get a unique description of the architecture if use the the channel objects and also include information about which task that is producing the output. A further improvement is to have a graphical description of the system where the inputs and the outputs of the tasks are connected with lines. If asynchronous and synchronous connections are represented in different ways in the graphical description it would be possible to automatically generate the tasks as well as channel objects in the tool. From the channel objects and the relation objects would it also be possible to create precedence chains automatically. The time parameters must though still be entered manually.

6. RESULTS OF THE SCHEDULING ANALYSIS

This section contains tables presenting the scheduling analysis result of the Powertrain Management application example. The task set tables for each resource are first presented in Section 6.1. In Section 6.2 tables showing the precedence chains in the system are given. All time values given in the tables are given in milliseconds (*ms*).

To include the tables of this section the ScheduLite tool was first extended with the function CopyToClipboard. This function copies the current contents of the string grid table to the Windows clipboard as text with each column separated with a tab-character and each row separated with a paragraph character. This text was then copied into Word where the text was converted to a Word table. To format the table a macro was created.

6.1 ANALYSIS RESULT TABLES

In this section the analysis result tables are given for the four nodes MMICU, ECU, GCU, ABSCU and the CAN-bus. The first column lists the names of the tasks allocated at the node. The variables then come as follows:

- *T*– the period time of the task.
- *D*– the deadline of the task.
- *C*– the worst-case execution time of the task and the worst-case transmission time for a message.
- *J*– the jitter of the task. In the application example all non zero jitter values originates from inherited jitter when messages are sent.
- *B*– the blocking factor. A task using a semaphore can get blocked by a lower priority task the maximum time *B*.
- *R*– the worst-case response time of the task. If $R>D$ for any task it misses it's deadline. In this way the analysis tells us that a task set is unschedulable.

In fixed priority scheduling each task must have a priority, denoted *P*. Since the deadline monotonic priority assignment algorithm is used, we have that $P=D$.

The semaphores are listed after the variables in the task tables. If a task is holding a semaphore the worst-case execution time of the critical section is found in that cell of the table. Here we have assumed that the critical section is 0,01 *ms* in all cases. Some tables are so big that they are divided into more than one table.

6.1.1 MMICU

Here follows the analysis result table of the MMICU:

Name	T	D	C	J	B	R
SenseAccPedalPos	50	25	0,025	0	0,01	0,91
DriverReq WhTorqAlg	50	50	0,05	0	0,01	1,9
PresentEngRPM	30	30	0,01	0	0,01	0,93
PresentGear	1000	1000	0,01	0	0	2,12
PresentVehicleSp	30	30	0,01	0	0,01	0,93
SenseGBSelPos	200	200	0,01	0	0,01	2,11
SenseDriveModeSelPos	400	400	0,01	0	0,01	2,12
SenseDriverCruiseCom	50	49,9	0,01	0	0,01	1,2
SenseBrakePedPos	5	5	0,025	0	0,01	0,435
CruiseController	50	50	0,1	0	0,01	1,9
CalcMaxAlloWheelTorq	5	5	0,05	0	0,01	0,435
CalcSpeedandAcc	5	4,7	0,2	0	0,01	0,21
CalcWheelSlip	5	4,8	0,025	0	0,01	0,235
CruiseWhTorqReg	50	50	0,4	0	0,01	1,9
GearShiftStateMan	5	4,9	0,025	0	0,01	0,31
EngTorqAlg	50	49	0,2	0	0,01	1,19
WheelTorqDemMan	50	48	0,01	0	0,01	0,94
EstimateFric	5	4,9	0,05	0	0,01	0,31
DecideDriveState	200	200	0,2	0	0,01	2,11
DecideGearShift	50	49	0,05	0	0,01	1,19
RecM_EngTorq	10	10	0,05	9,23	0,01	9,915
RecN_EngSpeed	10	10	0,05	8,78	0,01	9,465
RecZeroTorq	10	10	0,05	9,23	0,01	9,915
RecN_Wheel	5	5	0,05	3,165	0,01	3,6
RecSynchSpeed	10	10	0,05	9,29	0,01	9,975
RecClutchPos	10	10	0,05	8,455	0,01	9,14
RecActualGear	20	20	0,05	8,99	0,01	9,875

Table 6-1 The MMICU analysis result table.

6.1.1.1 SEMAPHORES AT MMICU

The semaphores of MMICU are divided into three different tables:

Name	Req_DriveWhe	a_VehAcc	Req_CruiseSpe	Req_CrusieWhe	M_EngTorq	N_EngSpeed
SenseAccPedalPo	0	0	0	0	0	0
DriverReq	0,01	0	0	0	0	0
PresentEngRPM	0	0	0	0	0	0,01
PresentGear	0	0	0	0	0	0
PresentVehicleSp	0	0	0	0	0	0
SenseGBSelPos	0	0	0	0	0	0
SenseDriveMode	0	0	0	0	0	0
SenseDriverCruis	0	0	0	0	0	0
SenseBrakePedP	0	0	0	0	0	0
CruiseController	0	0	0,01	0	0	0
CalcMaxAlloWh	0	0	0	0	0	0
CalcSpeedandAc	0	0,01	0	0	0	0
CalcWheelSlip	0	0,01	0	0	0	0
CruiseWhTorqRe	0	0,01	0,01	0,01	0	0
GearShiftStateM	0	0	0	0	0	0
EngTorqAlg	0	0	0	0	0	0
WheelTorqDem	0,01	0	0	0,01	0	0
EstimateFric	0	0	0	0	0,01	0
DecideDriveState	0	0	0	0	0	0
DecideGearShift	0	0	0	0	0	0
RecM_EngTorq	0	0	0	0	0,01	0
RecN_EngSpeed	0	0	0	0	0	0,01
RecZeroTorq	0	0	0	0	0	0
RecN_Wheel	0	0	0	0	0	0
RecSynchSpeed	0	0	0	0	0	0
RecClutchPos	0	0	0	0	0	0
RecActualGear	0	0	0	0	0	0

Name	ZeroTorq	V_VehSpeed	Max_WheelTor	Req_WheelTorq	SynchSpeed	ClutchPos
SenseAccPedalPos	0	0	0	0	0	0
DriverReq WhTorqAlg	0	0	0	0	0	0
PresentEngRPM	0	0	0	0	0	0
PresentGear	0	0	0	0	0	0
PresentVehicleSp	0	0	0	0	0	0
SenseGBSelPos	0	0	0	0	0	0
SenseDriveModeSelPos	0	0	0	0	0	0
SenseDriverCruiseCom	0	0	0	0	0	0
SenseBrakePedPos	0	0	0	0	0	0
CruiseController	0	0,01	0	0	0	0
CalcMaxAlloWheelTor	0	0	0,01	0	0	0
CalcSpeedandAcc	0	0,01	0	0	0	0
CalcWheelSlip	0	0,01	0	0	0	0
CruiseWhTorqReg	0	0	0	0	0	0
GearShiftStateMan	0,01	0	0	0	0,01	0,01
EngTorqAlg	0	0	0,01	0,01	0	0
WheelTorqDemMan	0	0	0	0,01	0	0
EstimateFric	0	0	0	0	0	0
DecideDriveState	0	0	0	0	0	0
DecideGearShift	0	0	0	0	0	0
RecM_EngTorq	0	0	0	0	0	0
RecN_EngSpeed	0	0	0	0	0	0
RecZeroTorq	0,01	0	0	0	0	0
RecN_Wheel	0	0	0	0	0	0
RecSynchSpeed	0	0	0	0	0,01	0
RecClutchPos	0	0	0	0	0	0,01
RecActualGear	0	0	0	0	0	0

Name	ActualGear	Req_DrivGea	Req_DrivMode	BrakePedalPos	ActualDri	Req_Gear
SenseAccPedalPos	0	0	0	0	0	0
DriverReq WhTorqAlg	0	0	0	0	0	0
PresentEngRPM	0	0	0	0	0	0
PresentGear	0,01	0	0	0	0	0
PresentVehicleSp	0	0	0	0	0	0
SenseGBSelPos	0	0,01	0	0	0	0
SenseDriveModeSelPos	0	0	0,01	0	0	0
SenseDriverCruiseCom	0	0	0	0	0	0
SenseBrakePedPos	0	0	0	0,01	0	0
CruiseController	0	0	0	0,01	0	0
CalcMaxAlloWheelTorq	0	0	0	0	0	0
CalcSpeedandAcc	0	0	0	0	0	0
CalcWheelSlip	0	0	0	0	0	0
CruiseWhTorqReg	0	0	0	0	0	0
GearShiftStateMan	0	0	0	0	0,01	0,01
EngTorqAlg	0	0	0	0	0,01	0
WheelTorqDemMan	0	0	0	0	0	0
EstimateFric	0,01	0	0	0	0	0
DecideDriveState	0	0,01	0,01	0	0,01	0
DecideGearShift	0	0	0	0	0	0
RecM_EngTorq	0	0	0	0	0	0
RecN_EngSpeed	0	0	0	0	0	0
RecZeroTorq	0	0	0	0	0	0
RecN_Wheel	0	0	0	0	0	0
RecSynchSpeed	0	0	0	0	0	0
RecClutchPos	0	0	0	0	0	0
RecActualGear	0,01	0	0	0	0	0

6.1.2 ECU

The analysis result table of the ECU:

Name	T	D	C	J	B	R
EngPowAlg	10	10	1	0	0,01	3,345
EngineSpeedAlg	10	10	1	0	0,01	3,345
EstEngTorq	10	9	0,4	0	0,01	0,91
SensAmbTemp	10	8	0,025	0	0,01	0,46
SenseEngSpeed	10	8	0,025	0	0,01	0,46
SenseMainFoldPr	10	8	0,025	0	0,01	0,46
KnockAlg	10	10	0,2	0	0,01	3,345
SenseKnockInten	10	9	0,05	0	0,01	0,91
SenseThrottlepos	2	1,9	0,025	0	0	0,025
ControlThrottle	2	2	0,1	0	0,01	0,135
CalcEngPos	5	4,8	0,05	0	0,01	0,185
SychManager	5	4,9	0,1	0	0,01	0,285
ActivateFuelInj	5	5	0,025	0	0,01	0,385
ActivateIgnition	5	5	0,025	0	0,01	0,385
SenseGBoutSpeed	10	9,8	0,01	0	0,01	0,92
SynchSpeedAlg	10	9,9	0,025	0	0,01	0,945
DecideWhenSynSpeed	10	9,95	0,025	0	0,01	0,97
Rec_ReqEngTorq	50	50	0,05	10,55	0	13,985
RecReq_Gear1	50	50	0,05	10,55	0	13,985
RecReq_EngRegMode	5	5	0,05	3,43	0,01	3,815

6.1.2.1 SEMAPHORES AT ECU

The semaphores of the ECU are diveded into two tables:

Name	Req_Fuel	Req_InjPos	Req_ThrottlePos	N_EngSpeed	M_EngTorq	R_Fuel+O
EngPowAlg	0,01	0,01	0,01	0	0,01	0
EngineSpeedAlg	0,01	0,01	0,01	0	0	0
EstEngTorq	0	0	0	0,01	0,01	0
SensAmbTemp	0	0	0	0	0	0
SenseEngSpeed	0	0	0	0,01	0	0
SenseMainFoldPr	0	0	0	0	0	0
KnockAlg	0,01	0,01	0,01	0	0	0,01
SenseKnockInten	0	0	0	0	0	0
SenseThrottlepos	0	0	0	0	0	0
ControlThrottle	0	0	0	0	0	0
CalcEngPos	0	0	0	0	0	0
SychManager	0	0	0	0	0	0
ActivateFuelInj	0	0	0	0	0	0,01
ActivateIgnition	0	0	0	0	0	0
SenseGBoutSpeed	0	0	0	0	0	0
SynchSpeedAlg	0	0	0	0	0	0
DecideWhenSynSpeed	0	0	0	0,01	0	0
Rec_ReqEngTorq	0	0	0	0	0	0
RecReq_Gear1	0	0	0	0	0	0
RecReq_EngRegMode	0	0	0	0	0	0

Name	R_InjPos+O	R_ThrPos+O	Req_EngSpeed	Req_Gear1	Req_EngRegMode
EngPowAlg	0	0	0	0	0,01
EngineSpeedAlg	0	0	0,01	0	0
EstEngTorq	0	0	0	0	0
SensAmbTemp	0	0	0	0	0
SenseEngSpeed	0	0	0	0	0
SenseMainFoldPr	0	0	0	0	0
KnockAlg	0,01	0,01	0	0	0
SenseKnockInten	0	0	0	0	0
SenseThrottlepos	0	0	0	0	0
ControlThrottle	0	0,01	0	0	0
CalcEngPos	0	0	0	0	0
SychManager	0	0	0	0	0
ActivateFuelInj	0	0	0	0	0
ActivateIgnition	0,01	0	0	0	0
SenseGBoutSpeed	0	0	0	0	0
SynchSpeedAlg	0	0	0,01	0,01	0
DecideWhenSynSpeed	0	0	0,01	0	0
Rec_ReqEngTorq	0	0	0	0	0
RecReq_Gear1	0	0	0	0,01	0
RecReq_EngRegMode	0	0	0	0	0,01

6.1.3 GCU - TASK SET AND SEMAPHORES

The analysis result table of the GCU including semaphores:

Name	T	D	C	J	B	R	Req_Gear	Req_ClutchPos
ControlClutch	10	9	0,025	0	0,01	0,135	0	0,01
ControlGearShift	20	19	0,025	0	0	0,15	0,01	0
RecReq_Gear	5	5	0,05	3,43	0,01	3,54	0,01	0
RecReqClutchPos	5	5	0,05	3,43	0,01	3,54	0	0,01

6.1.4 ABSCU- TASK SET AND SEMAPHORES

The analysis result table of the ABSCU including semaphores:

Name	T	D	C	J	B	R	N_Wheel	BrakePedalPos
ConHydrauValves	5	4,9	0,005	0	0,01	0,05	0,01	0,01
ConBrakeFluidPress	5	5	0,05	0	0	0,14	0	0
SenseBrakeFluid	5	4,81	0,01	0	0	0,01	0	0
SenseWheelSpeed	5	4,82	0,025	0	0,01	0,045	0,01	0
RecBrakePedalPos	5	5	0,05	3,555	0	3,695	0	0,01

6.1.5 CAN

The analysis result table of the CAN-bus:

Name	T	D	C	J	B	R
M_EngTorq	10	10	0,52	0,91	0,52	9,23
N_EngSpeed	10	10	0,52	0,46	0,52	8,78
ZeroTorq	10	10	0,52	0,91	0,52	9,23
N_Wheel	5	5	0,52	0,045	0,52	3,165
Req_EngTorq	50	50	0,52	1,19	0	10,55
SynchSpeed	10	10	0,52	0,97	0,52	9,29
ClutchPos	10	10	0,52	0,135	0,52	8,455
ActualGear	20	20	0,52	0,15	0,52	8,99
Req_Gear1	50	50	0,52	1,19	0	10,55
Req_EngRegM	5	5	0,52	0,31	0,52	3,43
Req_Gear	5	5	0,52	0,31	0,52	3,43
Req_ClutchPos	5	5	0,52	0,31	0,52	3,43
BrakePedalPos	5	5	0,52	0,435	0,52	3,555

6.2 PRECEDENCE CHAINS

The precedence chains are divided into two subsections: Section 6.2.1 contains precedence chains representing the synchronous connections of the processes given in the application example and Section 6.2.2 contains the precedence chains created for each message in the system. Those precedence chains are used by ScheduLite to define what tasks that should inherit jitter.

6.2.1 PRECEDENCE CHAINS FOR PROCESSES

Here follows the precedence chains for those processes that have synchronous relations, i.e. all processes with more than one software circuit.

6.2.1.1 MANUAL SPEED CONTROL PROCESS

Name	T	D	C	J	R
MMICU.SenseAccPedalPos	50	25	0,025	0	0,91
MMICU.DriverReq WhTorqAlg	50	50	0,05	0	1,9

6.2.1.2 ENGINE TORQUE CONTROL PROCESS

The Engine Torque Control Process has a branched synchronous chain. The SenseAmbTemp, SenseEngSpeed and the SenseMainFoldPressure tasks produce values all consumed by the EstimateEngineTorque task. Therefore we have three precedence chains to represent the branching. If we use the method of triggering tasks the EstimateEngineTorque task may not be triggered by more than one process, so we could let the three Sense tasks execute in the same order each time by releasing them at the same time and let the priority mechanism decide the execution order. The last executing task could then trigger EstimateEngineTorque.

Name	T	D	C	J	R
ECU.SensAmbTemp	10	8	0,025	0	0,46
ECU.EstEngTorq	10	9	0,4	0	0,91
ECU.EngPowAlg	10	10	1	0	3,345

Name	T	D	C	J	R
ECU.SenseEngSpeed	10	8	0,025	0	0,46
ECU.EstEngTorq	10	9	0,4	0	0,91
ECU.EngPowAlg	10	10	1	0	3,345

Name	T	D	C	J	R
ECU.SenseMainFoldPr	10	8	0,025	0	0,46
ECU.EstEngTorq	10	9	0,4	0	0,91
ECU.EngPowAlg	10	10	1	0	3,345

6.2.1.3 KNOCK CONTROL PROCESS

Name	T	D	C	J	R
ECU.SenseKnockInten	10	9	0,05	0	0,91
ECU.KnockAlg	10	10	0,2	0	3,345

6.2.1.4 THROTTLE CONTROL PROCESS

Name	T	D	C	J	R
ECU.SenseThrottlepos	2	1,9	0,025	0	0,025
ECU.ControlThrottle	2	2	0,1	0	0,135

6.2.1.5 ACTIVATE FUEL INJECTION AND IGNITION PROCESS

The Activate Fuel Injection and Ignition Process has a branch that is represented with two precedence chains:

Name	T	D	C	J	R
ECU.CalcEngPos	5	4,8	0,05	0	0,185
ECU.SychManager	5	4,9	0,1	0	0,285
ECU.ActivateFuelInj	5	5	0,025	0	0,385

Name	T	D	C	J	R
ECU.CalcEngPos	5	4,8	0,05	0	0,185
ECU.SychManager	5	4,9	0,1	0	0,285
ECU.ActivateIgnition	5	5	0,025	0	0,385

6.2.1.6 ANTI-SPIN REGULATION PROCESS

Name	T	D	C	J	R
MMICU.CalcSpeedandAcc	5	4,7	0,2	0	0,21
MMICU.CalcWheelSlip	5	4,8	0,025	0	0,235
MMICU.EstimateFric	5	4,9	0,05	0	0,31
MMICU.CalcMaxAlloWheelTorq	5	5	0,05	0	0,435

6.2.1.7 ANTI-BLOCK SYSTEM PROCESS

The two precedence chains representing the branched synchronous relation in the Anti-Block System Process:

Name	T	D	C	J	R
ABSCU.SenseBrakeFluid	5	4,81	0,01	0	0,01
ABSCU.ConBrakeFluidPress	5	5	0,05	0	0,14

Name	T	D	C	J	R
ABSCU.SenseWheelSpeed	5	4,82	0,025	0	0,045
ABSCU.ConHydrauValves	5	4,9	0,005	0	0,05
ABSCU.ConBrakeFluidPress	5	5	0,05	0	0,14

6.2.1.8 WHEEL TO ENGINE TORQUE CONVERSION PROCESS

Name	T	D	C	J	R
MMICU.WheelTorqDemMan	50	48	0,01	0	0,94
MMICU.EngTorqAlg	50	49	0,2	0	1,19

6.2.1.9 SPEED SYNCHRONISATION PROCESS

Name	T	D	C	J	R
ECU.SenseGBoutSpeed	10	9,8	0,01	0	0,92
ECU.SynchSpeedAlg	10	9,9	0,025	0	0,945
ECU.DecideWhenSynSpeed	10	9,95	0,025	0	0,97

6.2.1.10 SENSE DRIVER CRUISE COMMAND PROCESS

Name	T	D	C	J	R
MMICU.SenseDriverCruiseCom	50	49,9	0,01	0	1,2
MMICU.CruiseController	50	50	0,1	0	1,9

6.2.2 PRECEDENCE CHAINS FOR MESSAGES

Here follows the precedence chains including the sending task, the message and the receiving task for each message. All distributed relations in the application example are asynchronous, and the asynchronous point is placed at the node of the receiving task for all messages. The consuming task of the distributed asynchronous relation is also presented, including it's period time. A remark is made where it would be better to put the asynchronous point at the node of the sending task in the sense to decrease the load of the CAN-bus, i.e. if the consuming task has a longer period time than the producing process it is sufficient to send a message with new data at the rate of the consuming task.

6.2.2.1 N_ENGINESPEED

The precedence chain for the message N_EngineSpeed:

Name	T	D	C	J	R
ECU.SenseEngSpeed	10	8	0,025	0	0,46
CAN.N_EngSpeed	10	10	0,52	0,46	8,78
MMICU.RecN_EngSpeed	10	10	0,05	8,78	9,465

Remote consumer task:

MMICU.PresentEngineRPM, *T=30*.

With the asynchronous point at the sending node the message N_EngineSpeed could be sent with the rate *T=30* instead of *T=10*.

6.2.2.2 M_ENGINETORQUE

The precedence chain for the message M_EngineTorque:

Name	T	D	C	J	R
ECU.EstEngTorq	10	9	0,4	0	0,91
CAN.M_EngTorq	10	10	0,52	0,91	9,23
MMICU.RecM_EngTorq	10	10	0,05	9,23	9,915

Remote consumer task:

MMICU.DecideGearShift, *T=50*.

With the asynchronous point at the sending node the message M_EngineTorque could be sent with the rate *T=50* instead of *T=10*.

6.2.2.3 ZEROTORQUE

The precedence chain for the message ZeroTorque:

Name	T	D	C	J	R
ECU.EstEngTorq	10	9	0,4	0	0,91
CAN.ZeroTorq	10	10	0,52	0,91	9,23
MMICU.RecZeroTorq	10	10	0,05	9,23	9,915

Remote consumer task:

MMICU.GearShiftStateManagerFunction, *T=5*.

6.2.2.4 N_WHEEL

The precedence chain for the message N_Wheel:

Name	T	D	C	J	R
ABSCU.SenseWheelSpeed	5	4,82	0,025	0	0,045
CAN.N_Wheel	5	5	0,52	0,045	3,165
MMICU.RecN_Wheel	5	5	0,05	3,165	3,6

Remote consumer task:

MMICU.CalculateSpeedandAcceleration, *T=5*.

MMICU.CalculateWheelSlip, *T=5*.

In the channel specification of [1] it is listed that ECU.EnginePowerAlgorithm (*T=10*) also is a consumer of N_Wheel, but that is not consistent with the process descriptions, so we assume that the channel list is false.

6.2.2.5 REQ_ENGINETORQUE

The precedence chain for the message Req_EngineTorque:

Name	T	D	C	J	R
MMICU.EngTorqAlg	50	49	0,2	0	1,19
CAN.Req_EngTorq	50	50	0,52	1,19	10,55
ECU.Rec_ReqEngTorq	50	50	0,05	10,55	13,985

Remote consumer task:

ECU.EnginePowerAlgorithm., *T=10*.

6.2.2.6 SYNCHRONOUSSPEED

The precedence chain for the message SynchronousSpeed:

Name	T	D	C	J	R
ECU.DecideWhenSynSpeed	10	9,95	0,025	0	0,97
CAN.SynchSpeed	10	10	0,52	0,97	9,29
MMICU.RecSynchSpeed	10	10	0,05	9,29	9,975

Remote consumer task:

MMICU.GearShiftStateManager, *T=5*.

6.2.2.7 CLUTCHPOS

The precedence chain for the message ClutchPos:

Name	**T**	**D**	**C**	**J**	**R**
GCU.ControlClutch	10	9	0,025	0	0,135
CAN.ClutchPos	10	10	0,52	0,135	8,455
MMICU.RecClutchPos	10	10	0,05	8,455	9,14

Remote consumer task:

MMICU.GearShiftStateManager, *T=5*.

6.2.2.8 ACTUALGEAR

The precedence chain for the message ActualGear:

Name	T	D	C	J	R
GCU.ControlGearShift	20	19	0,025	0	0,15
CAN.ActualGear	20	20	0,52	0,15	8,99
MMICU.RecActualGear	20	20	0,05	8,99	9,875

Remote consumer task:

MMICU.EstimateFriction, *T=5*.

MMICU.PresentGear, *T=1000*.

MMICU.DecideGearShift, *T=50*.

In the channel list of [1] the DriveStateFunction is listed as a consumer. It should instead be DecideGearShift.

The message ActualGear has to be sent with the rate *T=20* since EstimateFriction needs a value each 5th *ms*.

6.2.2.9 REQ_GEAR1

The precedence chain for the message Req_Gear1:

Name	**T**	**D**	**C**	**J**	**R**
MMICU.DecideGearShift	50	49	0,05	0	1,19
CAN.Req_Gear1	50	50	0,52	1,19	10,55
ECU.RecReq_Gear1	50	50	0,05	10,55	13,985

Remote consumer task:

ECU.SynchronousSpeedAlgorithm, *T=10*.

6.2.2.10 REQ_ENGINEREGMODE

The precedence chain for the message Req_EngineRegMode:

Name	T	D	C	J	R
MMICU.GearShiftStateMan	5	4,9	0,025	0	0,31
CAN.Req_EngRegMode	5	5	0,52	0,31	3,43
ECU.RecReq_EngRegMode	5	5	0,05	3,43	3,815

Remote consumer task:

ECU.EnginePowerAlgorithm, *T=10*.

ECU.EngineSpeedAlgorithm, *T=10*.

With the asynchronous point at the sending node the message Req_EngineRegMode could be sent with the rate *T=10* instead of *T=5*.

6.2.2.11 REQ_GEAR

The precedence chain for the message Req_Gear:

Name	T	D	C	J	R
MMICU.GearShiftStateMan	5	4,9	0,025	0	0,31
CAN.Req_Gear	5	5	0,52	0,31	3,43
GCU.RecReq_Gear	5	5	0,05	3,43	3,54

Remote consumer task:

GCU.ControlGearShift, *T=20*.

In the channel list [1] it is said GearboxControlFunction where it should be ControlGearShift.

With the asynchronous point at the sending node the message Req_Gear could be sent with the rate *T=20* instead of *T=5*.

6.2.2.12 REQ_CLUTCHPOS

The precedence chain for the message Req_ClutchPos:

Name	T	D	C	J	R
MMICU.GearShiftStateMan	5	4,9	0,025	0	0,31
CAN.Req_ClutchPos	5	5	0,52	0,31	3,43
GCU.RecReqClutchPos	5	5	0,05	3,43	3,54

Remote consumer task:

GCU.ControlClutch, *T=10*.

With the asynchronous point at the sending node the message Req_ClutchPos could be sent with the rate *T=10* instead of *T=5*.

6.2.2.13 BRAKEPEDALPOS

The precedence chain for the message BrakePedalPos:

Name	T	D	C	J	R
MMICU.SenseBrakePedPos	5	5	0,025	0	0,435
CAN.BrakePedalPos	5	5	0,52	0,435	3,555
ABSCU.RecBrakePedalPos	5	5	0,05	3,555	3,695

Remote consumer task:

GCU.ControlHydraulicValves, $T=5$.

7. REFERENCES

[1] Sven Larsson and Magnus Lindahl, "*System Specification - ASTEC Application Example*", MEG-8033-DOC-001SS, Mecel AB, Göteborg.

[2] Jens Larsson, "*ScheduLite - A Fixed Priority Scheduling Analysis Tool*", Master of Science Thesis DoCS 96/73, ASTEC 96/01, Department of Computer Systems, Uppsala University, 1996.

[3] Tindell K.W., Hansson H. and Wellings A. J., "*Analysing Real-Time Communications: Controller Area Network (CAN)*", IEEE Real-Time Systems Symposium 1994 (RTSS '94), p 259-263.

[4] Tindell, K., Burns, A., "*Guaranteeing Message Latencies on Control Area Network (CAN)*", Department of Computer Science, University of York, 1994.

[5] Hans A. Hansson, Harold Lawson, Mikael Strömberg and Sven Larsson, "*Basement: A Distributed Real-Time Architecture for Vehicle Applications..*", Journal of Real-Time Systems, Kluwer Academic Publishers, Vol. 11, No. 3, p 223 - 244, November 1996.

[6] Jens Larsson, "*Information Interface to the Scheduling Level of a Hard Real-Time System Design Model*", Technical Report, DoCS 96/??, ASTEC 96/??, Department of Computer Systems, Uppsala University, 1996.

[7] Tindell, K., "*Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets,*" YCS 182, Department of Computer Science, University of York (1992).

[8] Tindell, K., "*Fixed Priority Scheduling of Hard Real-Time Systems*", PhD Thesis, Department of Computer Science, University of York (1994).

[9] Ken Tindell and John Clark, "*Holistic Schedulability Analysis For Distributed Hard Real-Time Systems",* Technical Report YCS197, Real-Time Systems Research Group, Department of Computer Science, University of York, 1993.

[10] Ken Tindell, *"Adding Time Offsets to Schedulability Analysis"*, Technical Report YCS221, Department of Computer Science, University of York, November 1994.

[11] Allan Burns and Andy Wellings, "*Real-Time Systems and Their Programming Languages*", Second Edition, Addison-Wesley Publishing Company, 1996. To be published.

[12] Mathai, Joseph (editor), "*Real-Time Systems - Specification, Verification and Analysis*", Chapter 3: Burns, A., Wellings, A. (authors), "*A Computational Model For Fixed Priority Scheduling*", Prentice Hall, 1996, ISBN 0-13-455297-0.