University of Twente

EEMCS / Electrical Engineering
Control Engineering

Redesign of the CSP execution engine

Bart Veldhuijzen

MSc report

Supervisors:
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
ir. M.A. Groothuis

February 2009

Report nr. 036CE2008
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

Nowadays the world is getting “computerized”. Embedded systems are getting more numerous and also become more complex. The formal language Communicating Sequential Processes (CSP) was designed to aid developers of embedded systems.

At the Control Engineering (CE) group CSP is used in the graphical modeling tool called gCSP. This tool allows to generate code from a gCSP model, which can be compiled against the Communicating Threads (CT) library into an executable. The CT library is an execution engine for CSP constructs. This library has two major problems; execution of a blocking system call will block the entire application, and the library does not provide the accurate timing required for real-time applications. This assignment redesigns this library to solve these problems and make the library future proof.

The blocking problem originates from the use of user level threads and scheduling in the library, which are invisible to the Operating System (OS). The inaccurate timing is caused by the non-preemptive scheduler. Replacing the user level threads and scheduler by kernel level threads and a priority-based preemptive OS scheduler solves these problems. To be able to support real-time applications, the OS has to be real-time as well.

The CSP language is used to create formally correct software. To classify as dependable software, the application has also to be safe and reliable. The OS must meet these requirements to be able to create dependable systems. The currently used RTAI does not meet these requirements.

The analysis of different kernel architectures shows that the microkernel based platform is safe and extensible. It has the benefit of using message passing as inter process communication, which is very much like the CSP rendezvous method. QNX Neutrino is a microkernel-based real-time operating system which uses channels for message passing, as does CSP. QNX is open source, provides a great integrated development environment, an instrumented kernel, Adaptive Partitioning Scheduling (APS) and transparent distributed networking.

The new library is structured in such a way that the OS provides the implementation of the API calls where possible. The CSP rendezvous communication is implemented using QNX channels and processes run in parallel using POSIX threading. Kernel tracing using the instrumented kernel provides the tracing and monitoring functionality, while retaining the real-timeliness of the application. APS is used to guarantee a group of processes a minimum amount of CPU time, and make remote debugging possible even when the system is fully loaded.

Testing the library for functionality and timing accuracy shows that the new library and QNX perform according to the specifications. The production cell setup is used to show the usability of the library for real-time control of a mechatronic setup.

From these tests it is concluded that the new library in combination with QNX can provide the necessary platform to develop real-time control applications using the CSP based toolchain at the CE-group. The main recommendations are to implement the missing functionality in the library and to research the use of multi-core platforms with the library. The possibilities of APS should be further investigated. The kernel event tracing can be used to reimplement the animation framework.

Samenvatting

Computers maken steeds meer deel uit van ons dagelijks leven. Het aantal 'embedded' systemen wordt groter en de systemen steeds complexer. Om ontwikkelaars hierbij the helpen is de formele taal Communicating Sequential Processes (CSP) ontwikkeld.

De Control Engineering (CE) vakgroep gebruikt CSP in de grafische modellerings tool gCSP. Deze tool maakt het mogelijk om van een gCSP model broncode te generen. Deze code kan gecompileerd worden met de 'Communicating Threads' (CT) bibliotheek tot een uitvoerbare applicatie. De CT bibliotheek voert de CSP bouwblokken uit. De huidige biblotheek heeft twee grote problemen; het uitvoeren van een blokkerende 'system call' zorgt ervoor dat de hele applicatie blokkeert en de bibliotheek beschikt niet over de vereiste nauwkeurige timing voor real-time applicaties. Deze opdracht omvat het herontwerpen van de biblotheek om deze problemen te verhelpen en de bibliotheek voor te bereiden op de toekomst.

Het gebruik van user level threading and scheduling in de huidige bibliotheek zorgt voor het blokerende probleem. De user level threads en de scheduler zijn onzichtbaar voor het besturingssysteem. De onnauwkeurige timing wordt veroorzaakt door het gebruik van een 'non-preemptive' scheduler. Het vervangen van de user level threading en de scheduler door kernel level threads en een op prioriteit gebaseerde preemptive scheduler in het besturingssysteem, verhelpt deze problemen. Om real-time toepassingen mogelijk te maken zal het besturingssysteem ook real-time moeten zijn.

CSP wordt gebruikt om formeel correcte software te ontwerpen. Software kan pas als betrouwbaar bestempeld worden als het ook veilig is. Het besturingssysteem moet ook aan deze eisen voldoen om betrouwbare systemen te kunnen realiseren. Het huidige RTAI voldoet niet aan de eisen.

De analyse van verschillende kernel architecturen laat zien dat de microkernel betrouwbare en uitbreidbare systemen mogelijk maakt. Microkernels maken gebruik van rendezvous communicatie als interprocess communicatie. Dit is vrijwel gelijk aan de manier van rendezvous communicatie in CSP. QNX Neutrino is een real-time besturingssyteem gebaseerd op een microkernel. Het gebruikt kanalen voor communicatie, net als CSP. QNX heeft beschikbare broncode, een zeer goede geïntegreerde ontwikkelomgeving, een geïnstrumenteerde kernel, Adaptive Partitioning Scheduling (APS) en transparant gedistribueerde netwerk ondersteuning.

De nieuwe bibliotheek is zo opgezet dat de functionaliteit van het besturingssysteem gebruikt wordt waar mogelijk. De rendezvous communicatie van CSP wordt verzorgd door QNX kanalen en het parallel uitvoeren van processen wordt gedaan met behulp van POSIX threading. Kernel tracing wordt gebruikt met behulp van de geïnstrumenteerde kernel voor tracing en monitoring zonder het real-time gedrag van de applicatie aan te tasten. APS wordt gebruikt om een groep van processen een gegarandeerde CPU tijd te geven. Hierdoor is debugging altijd mogelijk, ook als het systeem volledig belast wordt.

The functionele en timing testen laten zien dat de nieuwe bibliotheek in combinatie met QNX aan de eisen voldoen. The productie cell is gebruikt om de bruikbaarheid van de bibliotheek aan te tonen voor het real-time regelen van een mechatronisch systeem.

Deze testen laten zien dat de nieuwe biblotheek, samen met QNX, het noodzakelijke platform biedt voor de ontwikkeling van real-time regel applicaties in combinatie met de bestaande tools van de CE vakgroep. De belangrijkste aanbevelingen zijn het implementeren van de nog missende functionaliteit in de bibliotheek en om onderzoek te doen naar het gebruik van meerdere processoren. De mogelijkheden van APS moeten verder onderzocht worden. Het gebruik van kernel tracing maakt het mogelijk het animatie framework opnieuw te implementeren.

Contents

1 Introduction

1.1 Context

The world is getting “computerized”. In the industry this is going on for some time but nowadays most household appliances, kitchen utensils and even toys contain a small computer. These devices evolve into so called embedded systems. An embedded system is a complete device which not only contains hardware and mechanical parts, but also a special-purpose computer, designed to perform one or a few dedicated functions to control the hardware. The design of these systems becomes more and more complex since the requirements are growing.

At the University of Twente, the embedded control systems project of the Control Engineering group deals with the realization of control schemes on digital computers. The process algebra CSP (Communicating Sequential Processes) developed by Hoare (1985) and Roscoe et al. (1997) forms the theoretical basis. It is used to describe systems with several computational processes running at the same time, called concurrent systems. Embedded systems typically are such systems.

To aid the modeling of these systems with CSP, a graphical tool called gCSP has been developed (Jovanovic et al., 2004). This tool is able to generate code from a model which can be executed on a hardware device. The generated code has to be compiled against the Communicating Threads (CT) library (Hilderink et al., 1997), which is an execution engine for CSP constructs. The application can be monitored in gCSP through an animation facility (van der Steen et al., 2008). An overview is shown in figure 1.1.

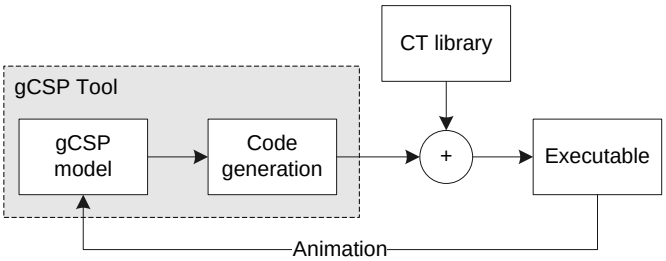


Figure 1.1: Overview of gCSP & the CT library

The current CT-library is a product of many years of research and development. In the early stages of development the decision was made to be platform independent where possible. That decision was very reasonable at the time, but results in undesired behavior nowadays:

- The CT-library is designed to do everything involving scheduling and threading by itself to be able to run on MS-DOS and DSP processors without an Operating System (OS). This means that if an OS is present, it runs in one OS thread. If some process in the CT-program executes a blocking system call the entire application is blocked. Furthermore the CT-library does not make use of multiple processors or cores if they are available.
- The Operating System has no knowledge of the scheduler and threads running in the CT-library. The CT-library has to deliver external events like timer interrupts to the appropriate process in the CT-program. The current scheduler cannot guarantee when this event is handled. This results in inconsistent and inadequate timing behavior (Maljaars, 2006; Deen, 2007).

1.2 Goals of the assignment

The goal of this project is to redesign the current CSP execution engine, the CT-library, to solve the problems and make it future proof. To determine a set of requirements for the execution

engine an analysis will be done to chart the needs for control software. From this set of requirements several software architectures will be investigated and a choice for a kernel architecture will be made. From a comparison of several operating systems built on the chosen kernel architecture an Operating System will be chosen. The new library is implemented on the chosen OS and should be compatible where possible with the current code generation in gCSP.

1.3 Report outline

In Chapter 2 the used terms and environment are explained in more detail. Chapter 3 describes the analysis and choices made resulting in the design of the new library as explained in Chapter 4. The new library is tested in Chapter 5 on the production cell setup.

Finally Chapter 6 summarizes the conclusions and presents the recommendations for further development and research.

More information and instructions on the use of the new library, kernel tracing, APS and QNet can be found in the appendices, including various gCSP models used throughout the report.

2 Background

This chapter discusses background information. The design methodology at the Control Engineering (CE) group and the currently used software are discussed first (Section 2.1). In Section 2.2 an overview is given of the hardware which is used at the CE group.Finally, different software architectures are explained in Section 2.3.

2.1 Design methodology

At the Control Engineering (CE) group embedded software is developed using the design trajectory as defined by Broenink and Hilderink (2001); Broenink et al. (2007), see Figure 2.1. This project falls in the *Embedded Control System Implementation* phase and the *Realization* phase. The entire trajectory consists of the four phases shown in figure 2.1. In the *Physical System Modeling* stage, models are made which describe the dynamic behavior of the system. The controllers for the system are made in the *Control Law Design* stage. These first two stages are performed in 20-sim (Broenink, 1999; Controllab Products, 2008). In the third *Embedded Control System Implementation* stage the control laws are implemented in software. This is done with 20-sim and gCSP (Jovanovic et al., 2004). At last the software is implemented on the target using the CT-library.

2.1.1 20-sim

20-sim is a graphical modeling and simulation program. It is possible to model a dynamic system using graphical representations and to simulate and analyze the behavior of the entire system. Using the Control Toolbox the controllers for the system can be designed. These controllers can be used in gCSP by generating code using the 20-sim Code Generation Toolbox.

2.1.2 (g)CSP

The theory of Communicating Sequential Processes (CSP), introduced by Hoare (1985), is a mathematical formalism for reasoning about patterns of communication in distributed systems. The system is represented by *processes* which engage in a sequence of *events*, which may include communication with another process via a *channel*. The set of all events that a process may engage in is called its *alphabet*. These can correspond to real-world occurences such as sensor-input, output, and so so on. Processes can define themselves in terms of other processes, including several processes running in parallel. The formalism provides for interprocess synchronization each time an event occurs that is in their common alphabet. This implies that processes synchronize around channel communication.

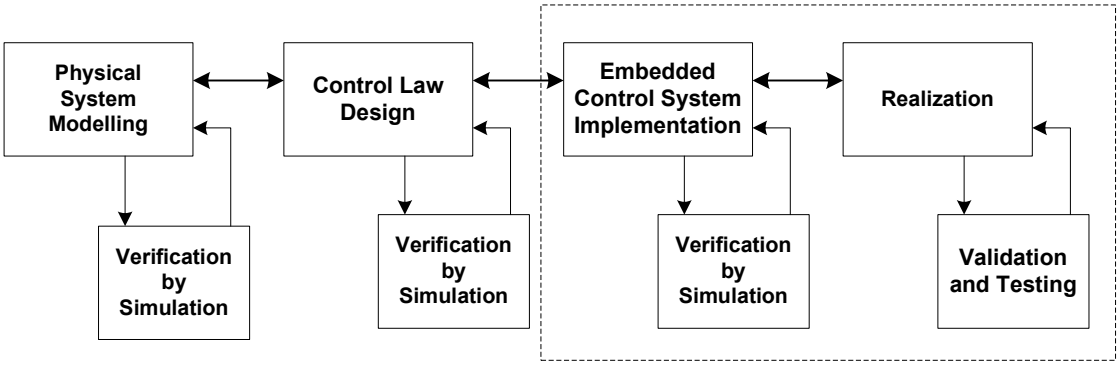


Figure 2.1: CE design methodology (Broenink and Hilderink, 2001; Broenink et al., 2007)

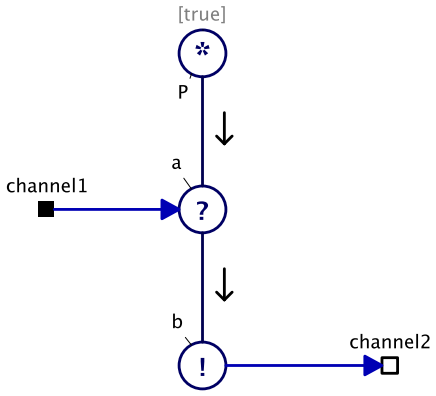


Figure 2.2: gCSP example model

Because performing manual analysis and verification of the system in CSP can be both tedious and error prone, automated tools are developed to formally check a design. For example the Failures-Divergence Refinement (FDR) tool developed by Formal Systems (Europe) Limited (2008).

The following is a short example of a process which first reads from a channel, afterwards it writes to another channel and then repeats itself.

$$P = channel1?a \rightarrow channel2!b \rightarrow P$$

At the CE group, CSP was first used on transputers using the Occam language (INMOS, 1988). When the production of the transputer ceased, a few universities developed an Occam API in libraries for mainstream programming languages. The Communicating Threads (CT) library was developed at the CE group (Hilderink et al., 2000; Orlic and Broenink, 2003; Hilderink, 2005). Similar libraries are developed at the University of Kent (JCSP and C++CSP) (Moores, 1999; Welch, 2002; Brown and Welch, 2003; Brown, 2007).

Hilderink (2005) introduced a graphical way to represent the CSP language. In the CASE tool gCSP (Jovanovic et al., 2004), systems can be modeled, visualized and animated (van der Steen et al., 2008). In figure 2.2 the graphical model of the earlier mentioned CSP example is given. From the model machine readable CPSm code can be generated for formal checking with FDR, or C++ code can be generated for compilation against the CT-library.

2.1.3 CT-library

The Communicating Threads library was developed to bring the Occam constructs, and inherently the CSP constructs, to platforms other then transputers. It was first developed in Java (Hilderink et al., 1997), after which versions in C and C++ were created. Later the Java and C versions were abandoned in favor of the C++ version. The library has been restructured a couple of times (Orlic and Broenink, 2004).

2.2 Hardware architectures

At the CE group, various custom setups and demonstrators are used, which are controlled by a few different standard hardware architectures. They can be divided in three categories, x86, ARM and others. The x86 group contains devices based around an x86 cpu. These can be normal PC hardware, or a PC104-stack, which is a small form-factor embedded computer containing various I/O boards. The tendency of multi-core processors is also noticeable in newer projects. The humanoid head and haptic demonstrator are both equipped with Intel Core2Duo processors. ARM and AVR based boards are used in smaller projects and are small embedded computing platforms. Research is also going on using FPGA chips containing PowerPC cores

for controlling setups.

2.3 Software architectures

2.3.1 Real-Time

A system is said to be real-time if the total correctness of an operation depends not only on its logical correctness, but also upon the time in which it is performed. In a hard real-time system, the completion of an operation after its deadline is considered useless - ultimately, this may lead to a critical failure of the complete system. A soft real-time system on the other hand will tolerate such lateness, and may respond with decreased service quality (e.g., dropping frames while displaying a video). This places some demands on the Operating System (OS) running on the system. The basic requirements according to Silberschatz et al. (2004) and Cooling (2000) of a Real-Time Operating System (RTOS) are:
- Preemptive, priority-based scheduling
- Preemptive kernel
- Fixed upper bound on latency
- Task structuring of programs
- Parallelism (concurrency) of operations

2.3.2 Kernel architectures

The kernel is the central component of most operating systems. Its primary purpose is to manage the resources available in the computer and allow other programs to run and use these resources. Typically, the resources consist of one or more CPU's, the memory and Input/Output devices, such as keyboard, disk drives, display. The kernel has full access to the system memory and must allow other processes to access safely this physical memory as they require it. Each process is given a separate virtual memory space, which is mapped to available physical memory. This virtual addressing also allows the creation of virtual partitions of memory. Typically, two partitions are available, one being reserved for the kernel (kernel space) and one for applications (user space). The separation is strict and enforced by the hardware which compares *every* address generated in user space to the allowed boundaries. An attempt to access an address in kernel space from user space results in a trap to the operating system (Silberschatz et al., 2004).

Nanokernel

Nanokernels are relatively small kernels which provide hardware abstraction, but offer no other system services. The term nanokernel has become analogous to microkernel with modern microkernels. En example of a kernel which calls itself still a nanokernel is Adeos (Adeos Project, 2004), used by RTAI (DIAPM, 2008) and Xenomai (Xenomai, 2008).

Microkernel

A microkernel is closely related to nanokernels. The first well-known microkernel was Mach (Rashid et al., 1989). It was intended to be a replacement for UNIX, but its performance was extremely low compared to UNIX. Microkernels were considered useless because of the low performance. Liedtke (1993) showed that the performance problems originated in bad design and implementation in the Mach kernel and proved with the L3 kernel that microkernels could perform very well. He formulated the minimality principle on which modern microkernels are build:

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality. (Liedtke, 1995)

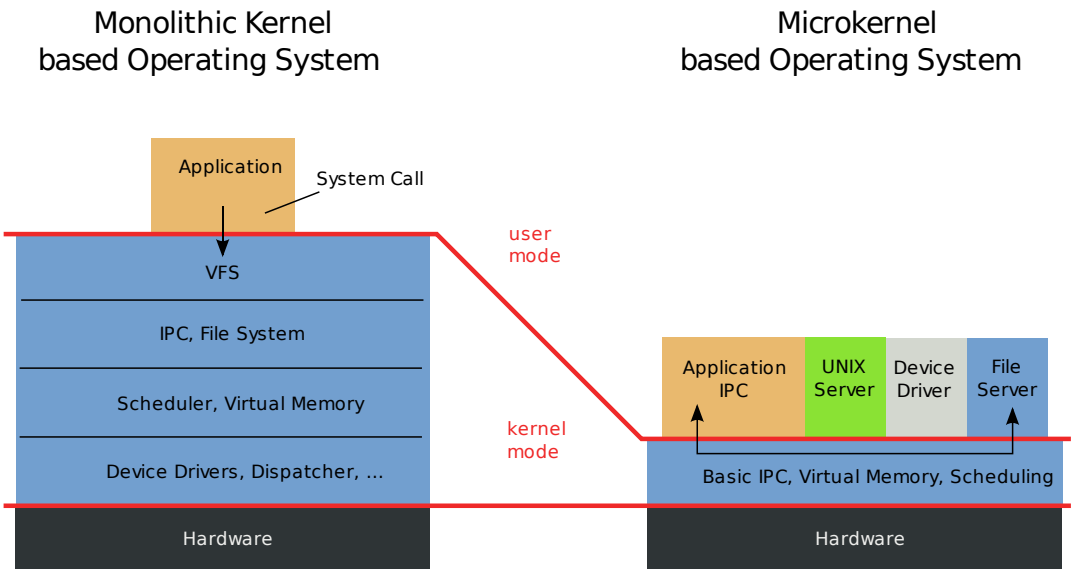


Figure 2.3: Kernel architecture overview

This comes very close to the definition of a nanokernel and is why the distinction between nanokernels and microkernels has faded. The minimality principle dictates that almost everything has to run in userspace, except the services which provide the mechanisms needed for the support of multiple processes:
- Managing memory protection
- Managing CPU allocation (threads or scheduling)
- InterProcess Communication (IPC)

All other OS services such as device drivers and filesystem drivers, run as normal tasks in user space, as can be seen in the right half of figure 2.3. This architecture relies heavy on an efficient means of communicating between different processes. The performance of the IPC implementation contributes for a great part to the performance of the entire OS. Existing microkernels and operating systems using microkernels are Mach, L4, QNX Neutrino, Minix, OpenRTOS, Drops, Symbian.

Monolithic kernel

The monolithic kernel design has all OS services running within the privileged mode (kernel space) of the processor. This is schematically shown at the left in figure 2.3. This makes communication between OS services efficient and fast because there are no switches between privileged mode and usermode. The drawback is that an error in a program in kernel space will likely corrupt and crash the kernel, and thus the entire system. Examples of operating systems using a monolithic kernel are Linux, FreeBSD, DOS, Windows 9x series.

2.3.3 Processes and threads

In Operating Systems terms, a process is a thread container. The process has its own address space which boundaries are guarded by the memory management unit in the cpu. A process groups the threads running in this address space. The threads themselves are the entities that are scheduled on the cpu by the scheduling algorithm. In Figure 2.4 this is schematically shown.

Switching between threads in a process is considerably faster compared to switching between threads in different processes. In the first case the address space in which the next thread runs

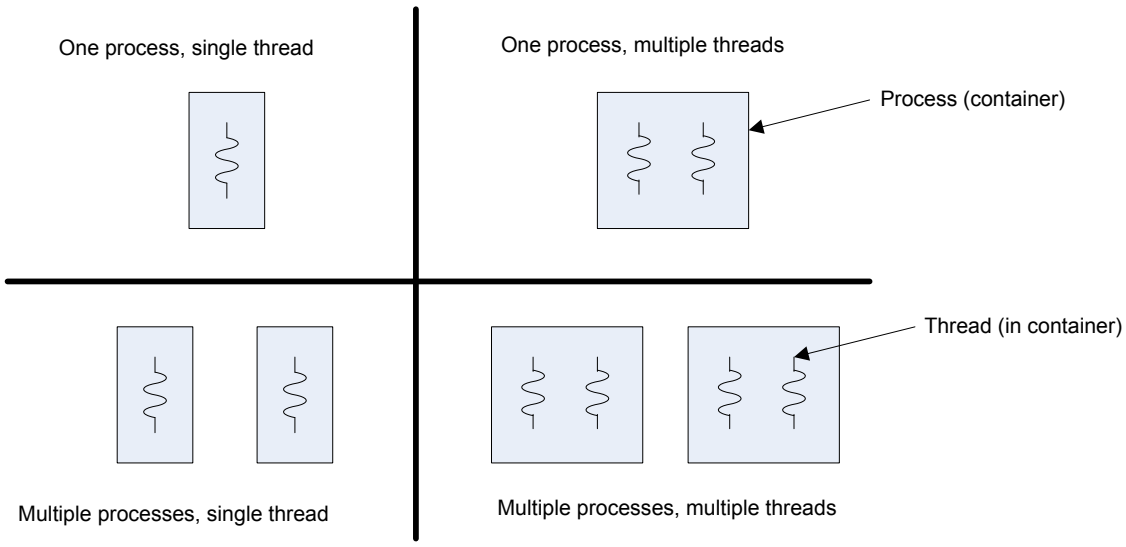


Figure 2.4: Thread and process types

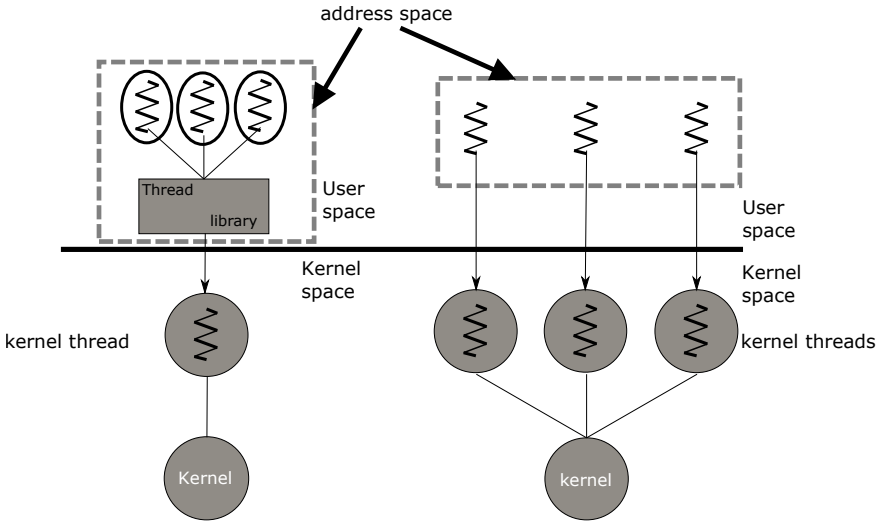


Figure 2.5: Diagram showing user level threads on the left and kernel level threads on the right

remains the same. When switching between processes the kernel has to change the address space on top of performing a context switch.

Thread types

Apart from the OS threads, also called kernel level threads, there are also user level threads. In Figure 2.5 the two types are shown. User level threads are unknown to the OS, and are all mapped to one OS thread. Kernel level threads are independent entities to the OS.

2.3.4 POSIX

POSIX stands for Portable Operating System Interface and is the name of a family of related standards specified by the IEEE. These standards define a standard operating *interface* and environment (POSIX.1). Several extensions to the standards exist, including real-time extensions (POSIX.1b) and threads (POSIX.1c, better known as pthreads). An operating system can be *POSIX conformant*, *Certified POSIX conformant* or *POSIX compliant*. Conformance means that the entire POSIX.1 standard is supported. Certified means it is accredited by an indepen-

dent certification authority and compliance means it provides partial POSIX support, which is indicated in its documentation.

Code which uses POSIX-calls can be compiled and run on any Operating Systems which is POSIX conformant, resulting in the same behavior.

POSIX scheduling

The POSIX standard defines four different modes in which threads can be scheduled by the OS: FIFO, Round-robin, Sporadic and Other. FIFO and round-robin only apply to threads at the same priority level. When FIFO scheduling is used, threads run until completion, or until preempted by a higher priority thread. Round-robin scheduling is identical to FIFO, but adds timeslicing. A thread is allowed to run until its timeslice is consumed, after which it is put at the back of the READY-queue. Sporadic scheduling can be used in combination with Rate Monotonic Analysis. The other scheduling mode is unspecified in the standard and is left to the OS.

2.3.5 Application profiling

Application profiling is used to investigate the behavior of a program using information gathered while the application executes. For application profiling several techniques are available, each with its strengths and weaknesses.

Sampling

Sampling uses the instruction pointer to record where the application is spending its time. It gives a rough estimate by using a target agent which periodically samples the instruction pointer. A separate tool, for example an IDE, can gather all the samples, aggregate them and present the results in the form of a table or annotated code. The strengths and weaknesses can be summarized as follows:

+ No recompilation of the application necessary
+ Low overhead
- Granularity depends on the periodic sampling
- Reliable results only over long period of time
- Possible incorrect results for timer based applications
- No call graph

Call count instrumentation

The call count technique requires instrumentation support in the compiler, linker, and libraries, and a recompilation of the application. It provides a precise call count of all functions and all function pairs. A separate tool can visualize the call graph and call counts. The strengths and weaknesses can be summarized as follows:

+ Precise call count information
+ Call pair information, aggregated as call graph
+ Relatively low overhead
+ Can extend sampling profiling
- Requires instrumentation
- No information for non-instrumented libraries

Function instrumentation

Function instrumentation is able to record precise function execution times and the runtime call graph. It uses hooks on entry and exit of each function, hence it needs a recompile of the application. It supports the visualization of the function table, threads tree, call graph, call tree and the annotated code. The strengths and weaknesses are:

+ Complete runtime call graph, including call counts and stack-frames
+ Precise function execution time
- Requires instrumentation
- Higher overhead

Kernel event tracing

Kernel event tracing allows to observe the system as a whole. It uses hooks in each kernel call to record each call. Together with the function instrumentation is provides a system-wide perspective of the target behavior. The strengths and weaknesses can be summarized as:

+ System-wide perspective
+ Precise information on context switches
- Due to the amount of available data, only relatively small timeframes possible
- Higher overhead when capturing the trace
- Requires an instrumented kernel

3 Analysis

In the first section an analysis is done of the requirements for the CT-library to support real-time systems and still fit in the tooling used at the CE-group. The current architecture of the library and its problems are discussed in Section 3.2 and proposed solutions are given. In Section 3.3 a closer look is given to various Operating Systems and the best fit to the requirements is chosen. The last section draws the conclusions from the earlier analysis.

3.1 Requirements

3.1.1 Real-time

The most important job of a real-time system is to run its real-time tasks. In control engineering the control loop is the most important task. The constraints are determined in the Control Law Design phase (figure 2.1) and are dictated by the controlled system. Typically, the control loop is designed to run on each sample moment of the hardware sensors. This is a hard deadline. If it does not finish before the next sample moment the controller could become unstable with potentially catastrophic results.

Apart from the real-time tasks, the system may have to perform other tasks. For example logging, handling a user interface and remote connections. These tasks could be classified as soft real-time, or even not real-time at all. For example writing logdata to a physical medium takes a long time, compared to sample times. This task should not influence the hard real-time tasks.

To be able to determine whether a system is real-time feasible, it is required to know the execution times of the tasks that are run. With this knowledge an analysis, e.g. Rate Monotonic (RM) or Earliest Deadline First (EDF), can be performed which indicates real-time feasibility (Marwedel, 2006). To determine the execution times the system has to be deterministic. This way the system behaves in a predictable manner and RM or EDF analysis can be used.

3.1.2 Operating System

The use of an Operating System greatly helps in the process of software development. The OS can take care of the most tedious tasks like booting the system, operating the hardware, managing resources and so on. The drawback is that it creates an extra layer between an application and the hardware, possibly affecting predictability and performance. An OS should therefore be real-time, or support real-time software. To classify as a real-time OS it should fulfill the requirements mentioned in Section 2.3.1.

3.1.3 CE-group

The new library should be backwards compatible with the old library where possible, to still be able to use the existing tools used at the CE-group. This means it should be usable in combination with the code generation in gCSP. It should also be available for the most used hardware platforms: x86, ARM and PowerPC.

3.1.4 Dependable software

The CSP theory is used to verify the correctness of the modeled application. To classify as dependable software, it has to be correct, reliable and safe (Cooling, 2000). The implementation of the application has to provide the reliability and safety, as does the system it is running on. After all, a chain is as strong as its weakest link. Software engineering techniques can help in developing, verifying and validating software parts. Safety layers can be provided through traditional operating system techniques, as well as language and compiler features.

3.2 Current architecture and problems

In the introduction (Chapter 1) the main problems with the CT-library were mentioned in short. In this section a more detailed analysis of the origin of the problems is given and where it fails to meet the requirements set in the previous section. Based on this analysis a solution is proposed.

3.2.1 CT-library

Threading

The current CT-library employs its own scheduler and threading system using user level threads (Hendriks, 1998; Hilderink, 2005). In Figure 2.5 the difference between kernel level threads and user level threads is shown schematically. The advantages of user level threads are:

- Operating System independent
- Usable on OS-less targets
- No kernel privileges needed for switching threads
- Fast thread switching possible

Whereas the negative aspects of user level threads compared to kernel level threads can be summarized as:

- All user level threads block when one thread does a blocking OS call
- Communication with non user level threads is hard to implement
- User level threads can not be distributed over multiple cores

The first two negative points make real-time dependable software nearly impossible to implement and outweigh the advantages the user level threads have to offer.

Scheduler

The internal user level scheduler in the CT-library is a prioritized FIFO scheduler without preemption and follows the OCCAM way of scheduling. For CSP behavior this is sufficient, but the lack of forced preemption causes non-deterministic behavior, which is unwanted for real-time applications. When a high priority process waits for example, on an external timer event, it is blocked and the scheduler will allow other processes to run (Figure 3.1). When the timer event arrives, the waiting process will be set in the READY state, but has to wait for the currently running process to finish before it can be scheduled to run. The CT-library has no means of preempting the running process, or even to limit the execution time of a process. As a result the latency to handling external events is unpredictable.

3.2.2 (Timed)CSP

CSP has no notion of timing. The current CT-library is based on CSP and has time support added by writing to external linkdrivers which use an OS timer. No mechanism is available for checking if the timing constraints are met.

There is substantial literature on Timed CSP (Hoare, 1985; Roscoe et al., 1997; Schneider, 1999), which adds a continuous time dimension to CSP. The main disadvantage of continues time is that it is infinite, which makes state verification impossible. Roscoe et al. (1997) therefore introduced the explicit time event *tock*, which implicitly introduces discrete time. In Istin (2007) and SystemCSP by Orlic (2007) the *tock* event is used to extend the CSP based models with timing. The improvements suggested by Istin (2007) are partly implemented in gCSP. SystemCSP is not yet implemented.

The *tock* event has been implemented in the CT-library by using TimerChannels, which use a linkdriver to wait for a systemtick. Due to the lack of preemption, as already explained in the previous section, acting on the timer event is undeterministic.

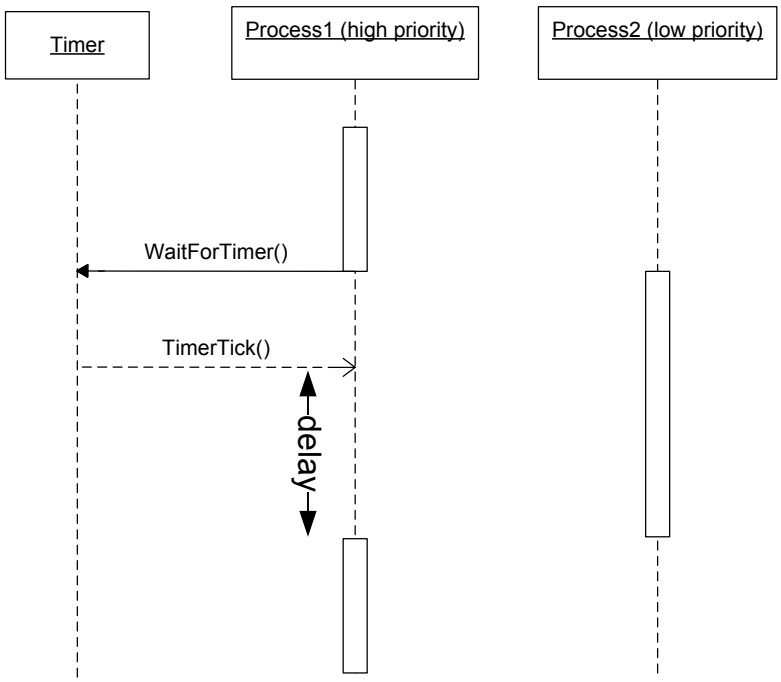


Figure 3.1: Unpredictable latency without preemption

3.2.3 RTAI

The Real Time Application Interface (RTAI) is strictly speaking not a real-time operating system but aims to add real-time capabilities to the standard Linux OS. It adds a hardware abstraction layer (Adeos) which implements an interrupt pipeline (ipipe). The RTAI kernel modules overtake Linux and are at the front of the interrupt pipeline. If RTAI does not handle the interrupts, they are passed on to Linux. Linux is a background task for RTAI and runs at a low priority. RTAI runs entirely in kernel space and real-time tasks run along RTAI, at a higher priority than Linux. In kernel space there is no memory protection between tasks (Section 2.3.2) and as a result tasks can have a direct impact on each other. Applications based on the current CT-library use LXRT to be able to use RTAI's hard real-time system calls while running in user space. This causes extra latency and the use of Linux system calls will cause undeterministic behavior. The big advantage of Linux, the great software and driverbase, is not available for real-time tasks.

3.2.4 Proposed solution

The usage of a priority based OS scheduler allows the removal of the internal CT-library scheduler and the usage of kernel level threads. This solves the problem of blocking system calls while still preserving the correct way of scheduling. The addition of a preemptive scheduler allows to react to external events with a predictable latency. This will also allow the improvement of the current implementation of the *tock* event.

The operating system for the CT-library should be reconsidered. RTAI fails to match the dependable software criteria of Section 3.1 and there may be better alternatives at the moment.

3.3 New architecture and approach

The choice for a specific architecture and Operating System should fulfill the requirements set in Section 3.1. In the next section different software architectures are examined. Section 3.3.2 inspects a number of operating systems based on those architectures, and set them out against the requirements and their properties. Section 3.3.3 introduces the modular structure for the new library.

3.3.1 Software architecture

There are three main software architectures used in real-time operating systems. The real-time executive, the monolithic kernel and the microkernel. A real-time executive is compiled as one big monolithic binary containing all required functionality normally found in an OS and the application. The significant downside to this approach is that all of the code, applications and kernel reside in one large address-space without protection.

The services and drivers provided by a monolithic kernel design (Section 2.3.2) reside all in kernel-space, without memory protection. This allows a high average throughput, and is easy to implement, but is very fault sensitive. A programming error in one part could crash or corrupt the entire system. The size of the kernel grows with the capabilities, in terms of binary size, as well as code size, making it hard to maintain and test. Debugging programs in kernel space requires special kernel debugging tools.

Microkernels (Section 2.3.2) have most services and drivers outside the kernel, running in user space, along normal applications. As a result they are guarded by the hardware memory management unit, can be relatively easy developed like normal applications, and debugged with regular debugging tools. The work of Molanus (2008) showed that the message passing paradigm used in microkernels is similar to the synchronization method in CSP compositional constructs.

3.3.2 Available operating systems

There are quite a few available operating systems which claim to be real-time, or support real-time applications. The ones mentioned in Table 3.1 are considered more closely because they are already used at the CE-group, are freely available, or stand out with respect to the others.

Property	RTAI	Xenomai	Open/FreeRTOS	DROPS/TUD:OS	QNX Neutrino
Hardware	++	++	-	-+	++
Drivers	++	++	-	-+	-+
Real-time	+	+	++	+	++
Scheduler	+	+	+	+	++
Safe	-	-	-	+	++
Documentation	-	-+	++	- -	++
Support	+	+	+	-+	++
Open Source	++	++	++	++	+
POSIX	+	+	-	+	++
Development tools	-	-	+	-	++
Debugging	- -	-	-+	+	++

Table 3.1: Requirements of different Operating Systems

RTAI (DIAPM, 2008) and Xenomai (Xenomai, 2008) originate from the same code-base, but have taken different paths. They support real-time applications in the monolithic kernel, but provide non real-time support through a separate Linux instance. Real-time applications have to run in kernel-space, and therefore they are not classified as safe and are hard to debug. Documentation for Xenomai is more up to date, but is not very extensive. They are both POSIX compliant. A great amount of hardware drivers is available.

OpenRTOS (High Integrity Systems, 2008) and its open source counter part FreeRTOS are real-time executives, but are mainly available for various microcontrollers. They provide a development environment including special debugging tools and very extensive documentation.

DROPS/TUD:OS (Technical University of Dresden, 2008) combines a real-time microkernel with a Linux instance. It is mainly used for research and has almost no documentation avail-

able. It supports the x86 architecture, support for ARM is still unstable.

QNX Neutrino (QNX Software Systems, 2008) is a microkernel based OS, available for a variety of hardware architectures. Specific hardware drivers may not be available, but driver development is relatively easy. There is no differentiation between real-time and non real-time tasks. Next to the normal priority based preemptive scheduler there is an additional and optional Adaptive Partitioning Scheduler. Very extensive documentation is available on the website. The source code is for a major part available, but is not licensed under a GPL-like license. It has a hybrid license, which allows the developer the choice of sharing their code. QNX is POSIX certified conformant. It offers an eclipse based development environment called Momentics with various debugging, tracing, profiling and monitoring tools.

Proposed architecture and OS

The match between CSP and the microkernel architecture make it the best choice. From Table 3.1 it can be concluded that QNX Neutrino, QNX for short, is the best match. QNX is the most extensive and mature microkernel based real-time operating system available. The drawback of a commercial license is taken away by the availability of (free) academic licenses. The extensive documentation, IDE and detailed tracing, debugging and profiling functionality make it absolutely superior to its competitors.

3.3.3 Library structure

The library provides the Application Programming Interface (API) to access the CSP execution engine. The operating systems provides the implementation details for this API. By using an OS which matches the CSP constructs, the implementation detail can be less. As seen in the previous section and Chapter 2, QNX requires less work to implement than RTAI or Xenomai, see Figure 3.2.

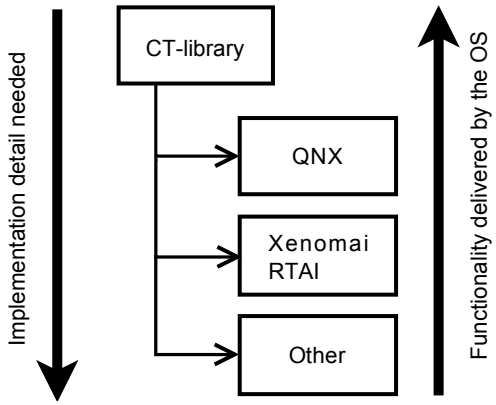


Figure 3.2: Relative amount of work needed to implement the CT-library

3.4 Conclusions

The current problems in the CT-library can be resolved by removing the internal scheduler and user level threading, and use the functionality offered by the operating system. This could result in a lower performance, which can be a drawback. A priority-based preemptive scheduler in combination with kernel level threads provides the same behavior as the current CT-scheduler with user level threads, but does not suffer from the blocking system call problem. Preemption is needed for deterministic interrupt latency.

The microkernel architecture depends on the message passing paradigm which is very similar to the CSP style synchronization. The match between microkernels and CSP make it easy to implement the CSP execution engine using the microkernel functionality. The design of a mi-

crokernel provides a safe computing platform because almost all services and applications run in userspace, guarded by the hardware mmu.

The microkernel based real-time OS QNX Neutrino fulfills the requirements for real-time systems and operating systems much better than the currently used RTAI. The available IDE and tools support the developer in creating dependable software.

By implementing the API of the library with native OS functionality, a matching OS will require less work than others.

4 Design and implementation

4.1 Introduction

A typical model of an application used in Control Engineering is shown in Figure 4.1. Several processes run in parallel and communicate with each other and interface with external hardware. Each process contains more subprocesses and constructs. Figure 4.2 shows the most used objects in gCSP and in control applications. The current CT-library and gCSP support and implement all the items, channels and constructs shown in Table 4.1. The new CT-library implements at this moment only the most used parts due to time constraints. In Table 4.1 is indicated which items, channels and constructs are implemented and which are not yet implemented.

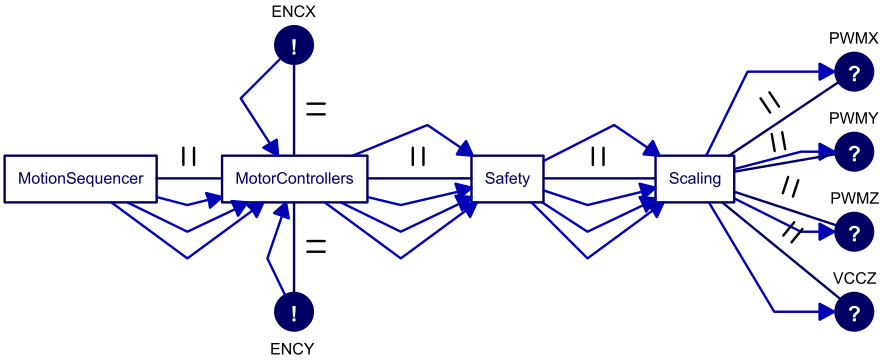


Figure 4.1: Typical gCSP model for control applications (Damstra, 2008)

In the next section, the design and implementation of Processes is presented, followed by the various Constructs in Section 4.3. The different types of Channels are discussed in Section 4.4. To show the additional functionality available in QNX, tracing and profiling is discussed in Section 4.5, while the possibility to use distributed systems with rendezvous channels is implemented in Section 4.6. A first attempt in using the advanced partitioning scheduler of QNX is explained in Section 4.7.

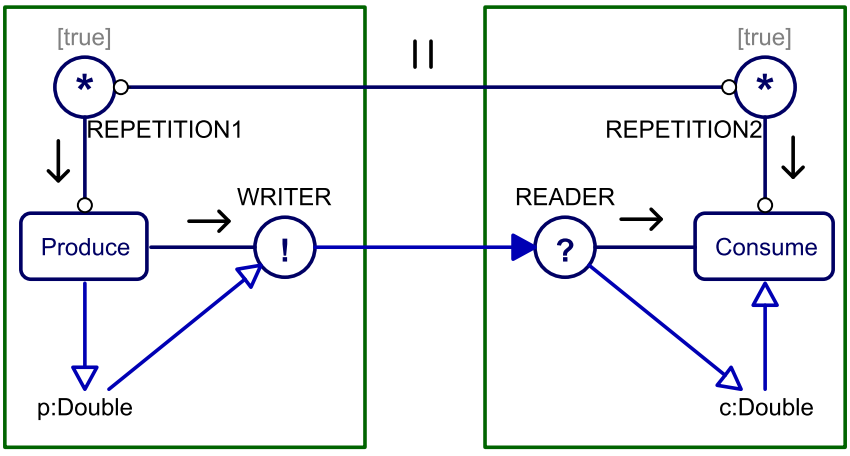


Figure 4.2: Most used gCSP constructs in exploded view

Item	Implemented	Construct	Implemented
Process	Y	Sequential	Y
Reader	Y	Repetition	Y
Writer	Y	Par	Y
		Pri Par	Y
Channel	**Implemented**	Alt	Y
Channel	Y	Pri Alt	N
TimerChannel	Y	Input-guard	Y
VarChannel	Y	Output-guard	N
BufferedChannel	N	SKIP-guard	N
ExternalChannel	Y	Watchdog	N
		Exception	N

Table 4.1: Available gCSP constructs in the new CT-library

4.2 Processes

A process in CSP terms is an object which can be executed and has references to the channels connected to it. The activity of the process is encapsulated in its `run()` method. Processes may interact with their environments only through their communication interfaces. A process itself can be composed of other processes and constructs. This implementation is equal to the one found in the old CT-library.

4.2.1 Readers and Writers

The READER and WRITER objects in Figure 4.2 are process instances whose only functionality is to communicate over a channel. The `run()` method is already filled. The WRITER puts a variable on the channel, the READER puts the received value in a variable.

4.3 Constructs

Constructs are implemented as processes without channel interfaces. Their children, processes composed in the constructs, are immediately connected to the channels. Each construct can be given an execution time-limit. If the processes in the construct are not finished within the time-limit, a notification is send to the construct.

4.3.1 Sequential

The sequential construct executes its child processes in sequence according to the order of processes in the declaration list of the construct. It terminates after the last child process has terminated.

4.3.2 Repetition

The repetition construct is a special form of a sequential construct. Instead of terminating after the last child process, it evaluates a predefined condition whether it has to repeat the sequence of processes or not.

4.3.3 Parallel

The parallel construct runs its child processes concurrently. To do so the child processes are dispatched to OS threads. The scheduler can decide to run the threads concurrently on one core, or real parallel on multiple cores, if they are available. The scheduler in the old CT-library uses OCCAM based scheduling, which closely resembles FIFO scheduling. In the new library threads are scheduled default according to the FIFO algorithm.

A threadpool is created at the construction of the parallel construct which holds one thread for each child process. All child processes are dispatched to their own thread, leaving the main thread available for monitoring timeouts on the execution time. When a child process terminates, the freed thread is returned to the pool. The threadpool is only destroyed on destruction of the parallel construct. For consecutive executions of the same parallel construct (calling the `run()` method), the threads in the threadpool are reused, eliminating the overhead of creating and destroying OS threads repetitively. This behavior differs from the old CT-library, where creating and destroying user level threads is a much cheaper operation.

4.3.4 PriParallel

Instead of dispatching each child process to a thread with the same OS priority, the threads are given a higher priority according to the order of processes in the declaration. This differs slightly from the original implementation where priorities were relative to each other, whereas in the new construct they are absolute OS priorities. To prevent overlapping priorities in nested PriParallel constructs, the step size between priorities can be adjusted. There is a limit of 256 priority levels in QNX, 0 being the idle thread, 255 the highest priority. In most applications this limit will not pose a problem.

4.3.5 Alt

The alternative construct offers the environment a choice between its child processes, based on which process can accomplish a channel communication. Each child process has a guard listening on the associated channel. At the moment only ChannelInput guards are implemented. Because ChannelOutput guards are difficult to implement without additional helper processes, and output guards are rarely used, they are not yet implemented. When a writer on the other end of a channel becomes ready to communicate, the alternative construct executes the child process connected to the channel.

On construction a threadpool is created with one thread for each guard. The guards are dispatched to their own thread and try to establish channel communication on a specific channel. On success the alternative construct is notified, the connected child process is started and given the established channel communication. The child process completes the rendezvous communication with the sender. The other guards are canceled and the channels they were listening on are released. The threads are returned to the threadpool. The pool is destroyed only when the alternative construct is destroyed by its parent.

4.4 Channels

In CSP, processes can become blocked on communication events, which is indicated by its state. A process can be RUNNING, READY, SEND blocked, or RECEIVE blocked. The thread states used by QNX in channel communication are nearly identical, but the rendezvous behavior differs from the CSP kind on one point. In QNX, the receiving end has to explicitly reply to the sender it has received the message. When a process writes a message to a channel and no reader is waiting, it is put in the SEND-blocked state as shown in Figure 4.3. When a reader becomes available the message is sent and the writer is put in the REPLY-blocked state, meaning it is waiting for an answer from the receiving end of the channel. A process could wait until it has finished its work before it sends the reply message. The CSP rendezvous does not support this behavior, so in the library a reply is send back to sender as soon as the message is received.

When a process wants to read from a channel, and no message is available yet, it is put in the RECEIVE-blocked state (Figure 4.4). After it is has received a message it has to reply to the sender. This is a non-blocking operation which puts the writer back in the READY-state. Multiple writers or readers on one channel are queued according to priority. Only one reader

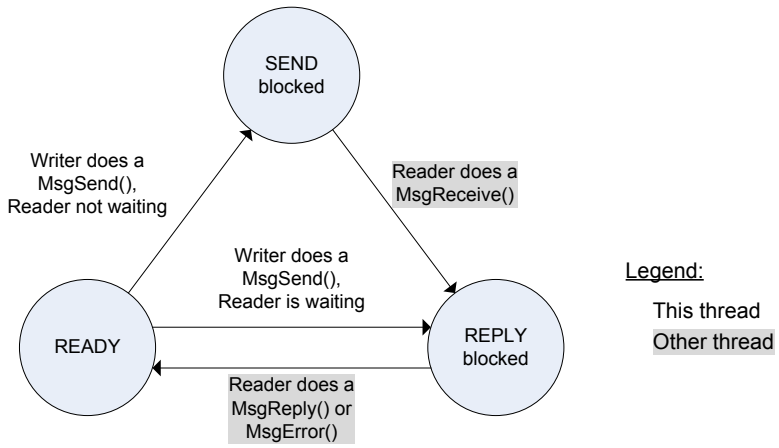


Figure 4.3: State changes in channel communication on the writer side (this thread)

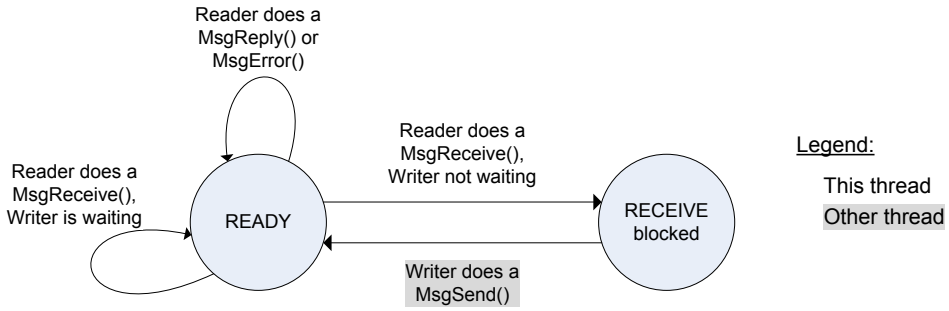


Figure 4.4: State changes in channel communication on the reader side (this thread)

and one writer can be active on a channel at all times.

A CSP channel supports multiple readers and writers concurrently. When they try to use the channel at the same time, they are ordered by their priority on a first come, first served basis. A QNX channel has the same properties as the required One2OneChannel, Any2OneChannel, One2AnyChannel and Any2AnyChannel, which means only one implementation suffices.

The priority inversion problem is prevented by using the priority inheritance protocol. The QNX kernel has this protocol standard implemented in the channels. This means that the priority of the process at the receiving end is temporarily boosted to the priority of the sender. After the reply message is sent, the receiver has to take care itself of returning to the original priority. In the new CT-library the priority is checked after each completed communication event and the priority is adjusted when necessary.

A major benefit of using QNX native channels, is the support for timeouts. Each potentially blocking operation can be guarded by a timeout, completely implemented in the QNX kernel. In the new CT-library, the read and write actions on a channel are extended with an additional timeout parameter. On a timeout, the kernel unblocks the thread and the read or write action returns an error. The return value of a read or write action has therefore to be checked.

4.4.1 TimerChannel

Processes should have the possibility to explicitly synchronize with the *tock* event (Section 3.2.2). This event has to be based on the OS timer for accurate system wide timing synchronization. QNX allows to program the system timer to deliver a message over a channel, send a signal to a thread, or create a thread on the occurrence of a timer tick. Using a channel matches with the CSP way of synchronizing with *tock* events.

A TimerChannel is implemented as a special type of channel which will block until a timeout has elapsed. It can be used to synchronize periodically on a *tock* event, or to wait for a specific amount of time to elapse.

To wait a specific amount of time, the process writes the desired amount to the TimerChannel, which programs the timer in one-shot mode. When the timer fires it delivers a message over the channel to the waiting process.

For periodic synchronization the timer is programmed in periodic mode. This gives a more accurate timing because it prevents the overhead of programming the timer multiple times. The timer sends a message over the TimerChannel on every timer tick. These messages accumulate on the channel, if they are not received by the process. It is not yet possible to check if there are multiple messages waiting on a channel.

4.4.2 External channels

The communication with the outside world is usually performed using a hardware device. The software driver provides the interface for applications to access the hardware device. QNX drivers are normal user-space programs. They run in user-space and request from the OS low-level access to the hardware they need. Because of the message passing used in QNX, the interface from the driver to other programs has to go through messages. Therefore a driver registers itself with a pathname (e.g. /dev/motordriver) by creating a channel. A program willing to communicate with the driver simply connects to the channel and sends messages over it. The driver replies with the appropriate response and activates the hardware.

This mechanism is very much like the normal channels described above, with the exception that the channel does not need to be created, only connected to. The ExternalChannel behaves after connecting exactly like a normal channel. The linkdrivers used in the old library are no longer necessary because the driver already provides the message passing interface. It is not yet possible to draw and use ExternalChannels directly in gCSP. The existing LinkDriver objects still have to be used, but are nothing more than a link between a READER or WRITER and the ExternalChannel.

4.5 Tracing and profiling

Tracing, monitoring (Posthumus, 2007) and animation (van der Steen et al., 2008) are used to monitor and visualize the execution of a system or application. Two levels at which information about the system is desired are the CSP level, and the code level. The CSP level uses the processes, constructs and the synchronized communication to evaluate the correct behavior of the system, according to its CSP specification. At code level it gives insight in the actual behavior of the program, the functionality enclosed in the `run()` method of a process.

The functionality to trace and profile the behavior of the program should not influence the real-time part. The facilities in the old library are not decoupled from the real-time parts, which results in unpredictability and are therefore not suitable for real-time systems. The current tracing functionality is putting messages on standard output, typically the screen. This is a blocking system call, which does not influence the custom scheduler in the old library, but will influence the QNX scheduler. A process which unblocks again will be put at the end of the ready queue for its priority level. Writing to standard output is a very time consuming operation, so it should be avoided in real-time applications.

The old tracing macros are still in place in the new library. They are a quick way to get a view of what the program is doing, but due to the influence on the scheduler, they also change the execution order of the processes slightly. In a relatively small program governed by communication events this will not be a problem, but the possibility exists that when a program relies on accurate timing, undesired results can appear. A less invasive way of gathering information

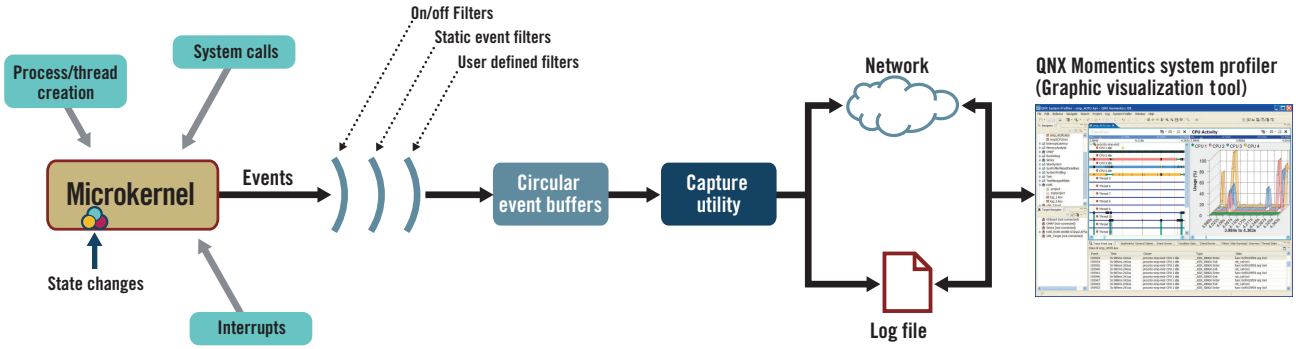


Figure 4.5: QNX instrumented kernel overview (QNX Software Systems (2008))

can be found in instrumentation of the code.

4.5.1 Instrumentation

When a recompilation of the application is possible, call count instrumentation or function instrumentation (Section 2.3.5) can provide a better way of tracing. The QNX Momentics IDE can visualize the data collected from these methods. Examples are shown in the next Chapter and in Figure C.2.

For a detailed view of the entire system, kernel event tracing has to be used. QNX has a special instrumented kernel (Figure 4.5) which allows to record information about what the kernel is doing, generating very precise time-stamped events that are stored in a circular linked list of buffers. Not only system calls like thread creation, but also interrupts and read/write actions on channels are recorded. In combination with the call count instrumentation or function instrumentation, this gives a really detailed view of the system and the application. The instrumented kernel runs at 98% of the speed of the regular microkernel QNX Software Systems (2008). The events can be captured by a tracelogger and visualized in the IDE. For an overview of the possibilities of the instrumented QNX kernel see Molanus (2008) and QNX Software Systems (2008). This functionality allows to trace and visualize the behaviour of the program and see the influence of other system parts on the application, without modifying the sourcecode of the application.

The instrumented kernel allows to insert user-generated events into the stream of kernel events. They are very small, integer-based events, which can replace the old tracing macros. To convert the numbered events to more meaningful messages, the IDE can decode the events using an XML-file which describes the events. There are no special functions to accomplish this implemented in the library yet. Adding a user-event has to be done by hand by calling the QNX `TraceEvent()` function. For more information about setting up the instrumented kernel and using the IDE for tracing, see Appendix C. The QNX Foundation Classes (QFC) (Allen, 2008) have preliminary support for using the instrumented kernel during runtime, but this is still a proof of concept and therefore it is not used in the library. Further research is needed to asses if the QFC is usable, or if the techniques used in the QFC can be reused in the CT-library.

4.6 Qnet

Distributed processing involves multiple nodes which have to communicate over some type of network link. QNX has its own protocol for distributed networking called Qnet. It extends the message passing architecture over a link, e.g. ethernet, resulting in transparent access to any resource on any node (Figure 4.6).

In the new CT-library, Qnet is used to implement RemoteChannels. They make systems like the one shown in Figure 4.7 possible. The reading end of the remote channel uses a QNX resource

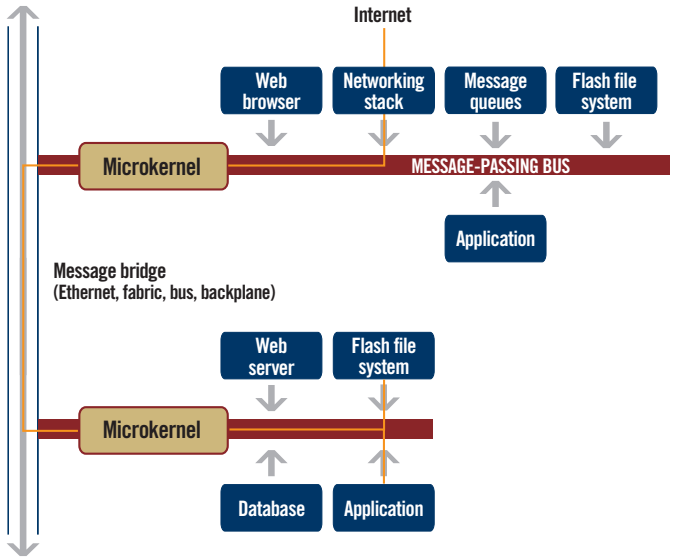


Figure 4.6: Transparent distributed processing overview (QNX Software Systems (2008))

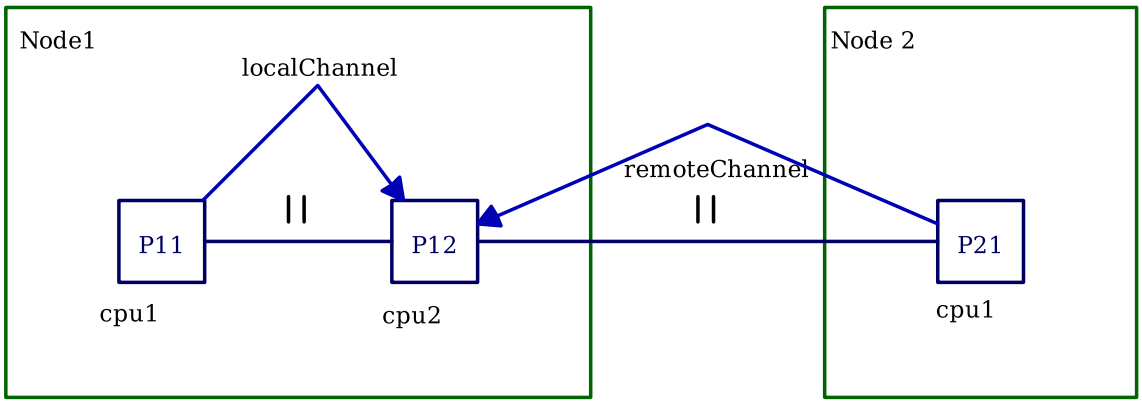


Figure 4.7: Remote channels in gCSP

```
#Create the reading end of a remote channel
Channel<int>* chan = new Channel<int>(REMOTEREAD, "/csp/channel");
#Normal read from the channel
if ((ret = chan->read(&value, timeoutvalue)) != EOK)
{
        std::cout << "Read failed: " << strerror(ret) << std::endl;
}
```

Listing 4.1: Creating RemoteChannels, reading end

```
#Create the writing end of a remote channel
Channel<int>* chan = new Channel<int>
                              (REMOTEWRITE, "/net/node1/csp/channel");
if ((ret = chan->write(&value, timeoutvalue)) != EOK)
{
        std::cout << "Write failed: " << strerror(ret) << std::endl;
}
```

Listing 4.2: Creating RemoteChannels, writing end

manager to attach itself to a pathname (e.g. /csp/channel1) (Listing 4.1). The writing end of the channel is able to open the pathname like a normal file (e.g. open /net/node1/csp/channel1) and write to it (Listing 4.2), just like a regular ExternalChannel. The established channel works like a normal local channel and can be used for rendezvous communication. The deterministic behavior of the channel is determined by the network link between the nodes. The procedure to setup Qnet is explained in Appendix E.

4.7 Adaptive Partitioning Scheduler

The QNX Adaptive Partitioning Scheduler is an optional thread scheduler that guarantees a minimum percentage of CPU time to groups of threads, processes, or applications. This means that even under full load, low priority processes in a different partition will still get their minimum percentage CPU time. In Figure 4.8 this is schematically shown. Each partition has its own guaranteed CPU time. If the system is not fully utilized, a partition is allowed to utilize more than its budget of CPU time, taken from other partitions.

In the new CT-library an application is started at default in a partition with a budget of 80% CPU time, which can easily be changed. The system has furthermore a debugging partition of 10%, which enables the posibility of remote or local debugging, even when an application is requesting 100% CPU time. The remaining percentage is automatically given to the system, and to all other programs which are eventually running. In Appendix D more information is given about setting up APS, and how to properly configure the debugging partition.

4.8 Conclusions

The use of functionality already present in the Operating System has slimmed down the CT-library compared to the old version. QNX provides the rendezvous channels, the threading mechanism and the scheduler for the library. The most often used constructs are implemented in the library.

The old tracing functionality should not be used anymore because of the negative influence on the execution order and real-time behavior. The QNX instrumented kernel fills this gap by offering detailed and precise tracing of events, which can be visualized in the IDE, without

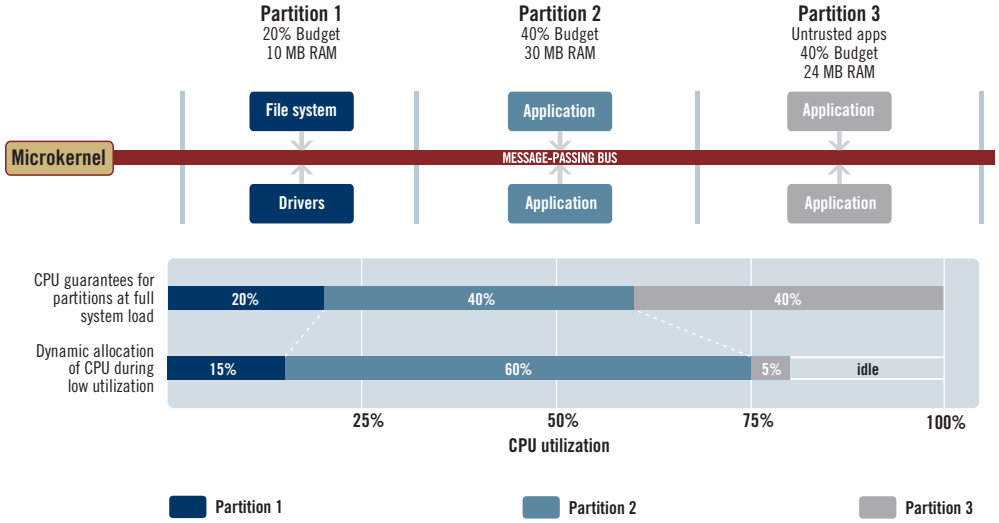


Figure 4.8: Adaptive partitioning (QNX Software Systems (2008))

disturbing the real-timeliness of the system.

Distributed processing is made available through RemoteChannels, which use Qnet and resource managers to extend the normal rendezvous channels over a network link.

Adaptive partition scheduling is used to guarantee a specific percentage of CPU time to a group of threads, meaning other applications cannot starve the critical threads. A special debugging partition is created to allow debugging even when the system is fully loaded.

5 Testing and Evaluation

5.1 Introduction

The new library designed and implemented in Chapter 4 is tested using functional tests to check the correct behavior of the library, and the timing subsystem is tested for performance. A part of the production cell setup is used to test the real-time capabilities on a hardware setup. The functional tests are presented in the next section, the timer tests are shown in Section 5.3. The test on the production cell setup is presented in Section 5.4. Finally the conclusions are presented in the last section.

All tests are performed on a PC104 stack, running QNX Neutrino 6.3.2 with the instrumented kernel as explained in Appendix C. Advanced Partitioning Scheduling is activated and configured as described in Appendix D. The stack is equipped with an Anything I/O board for the connection with mechatronic hardware.

5.2 Functional tests

5.2.1 Description

The API of the new library is backwards compatible with the old one, meaning the existing test examples in the old library can be reused easily. To validate the correct behavior, the following simple and more complex testcases are modeled in gCSP. The generated code is compiled and linked against the new library. The following tests are executed:

- Sequential (Figures A.1(a) and A.2)
- Parallel (Figure A.1(b))
- Priparallel (Figures A.1(c) and A.3)
- Producer consumer deadlock (Figure A.4(a))
- Producer consumer deadlockfree (Figure A.4(b))
- Alt (Figure A.5)
- Dining Philosophers problem (Figure A.6)
- ComsTime test (Figure A.7)

The used gCSP models can be found in Appendix A.1.

5.2.2 Results

Sequential and Parallel

All sequential and parallel models behave as expected. The trace of the simple sequential model (Figure A.1(a)) is shown in Figure 5.1. The two black lines in the circles indicate the moments where an userevent is injected in the kernel. The exact timestamps are shown in the trace tab below the timeline. First Process1 injects a userevent (1) when it gets run, Process2 does the same (2) when it starts running.

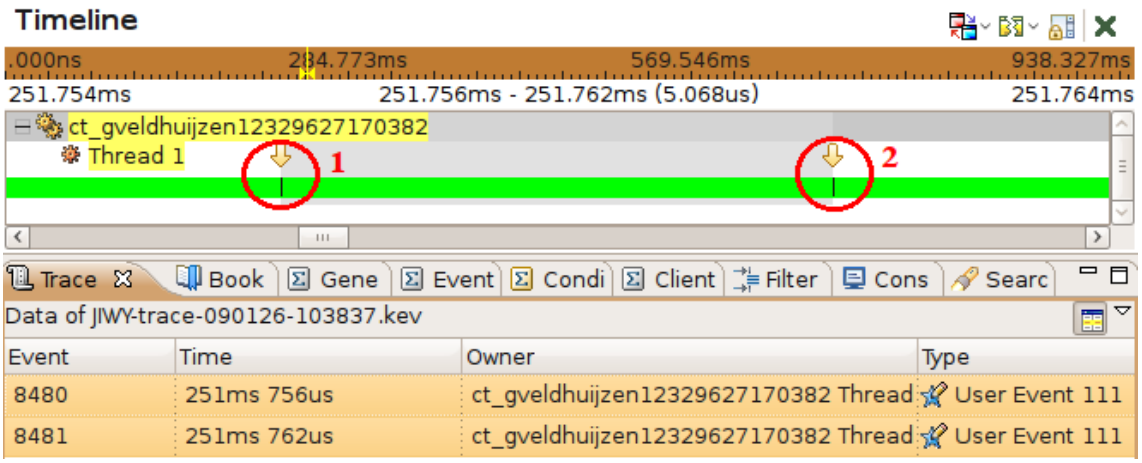


Figure 5.1: Sequential trace

In Figure 5.2 the trace from the simple parallel model (Figure A.1(b)) is shown. The application consists of 3 threads, the main Thread1, and two pooledthreads for the parallel processes. The black lines in the circles indicate again the userevents.

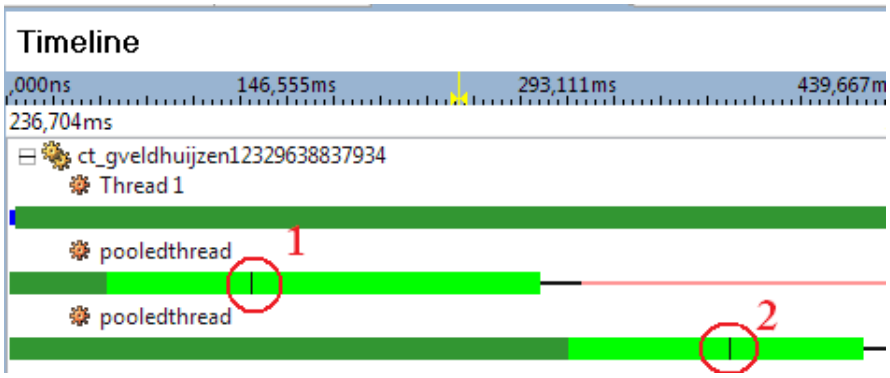


Figure 5.2: Parallel trace

The more complex sequential model of Figure A.2 is traced in Figure 5.3. The application runs the display process in one thread, and the sequential processes 1-4 in the other thread. The communication can be seen as the arrows between the threads. A process in the sequence writes to the waiting display process (downward arrow in blue (1)). The display process runs, receives the message and replies to the sending process (upward arrow in green (2)). When the display process tries to receive on the channel again, it gets RECEIVE blocked, visualized as the thin horizontal blue line (3). The REPLY blocked state of the sequential processes is indicated with a thick horizontal blue line (4). Dark-green bars mean the thread is in the READY state (5), lighter green indicates the RUNNING state (6).

Consumer Producer models

The deadlock free model (Figure A.4(b)) of the Consumer-producer system behaves as it should. The deadlock in the model in Figure A.4(a) can already be seen by looking at the running application in the System Information Perspective in the IDE (Figure 5.4). Both Consumer and Producer are blocked, the producer SEND blocked on one channel, and the consumer RECEIVE-blocked on another channel. Because the channels are different, the application is in deadlock and will not make any further progress.

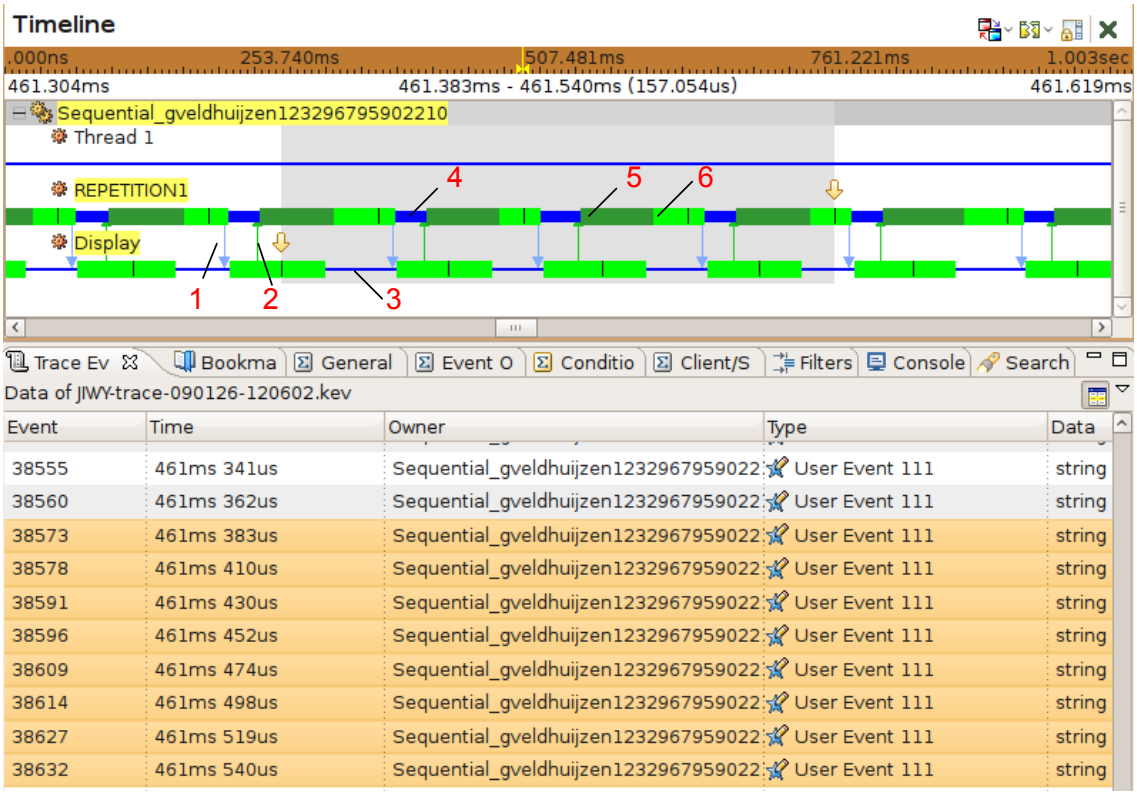


Figure 5.3: Complex sequential trace

Alt

The expected behavior of the alt example shown in Figure A.5 is as follows. The user gives in a number corresponding to the channel a message is sent on. The process listening on the other end of the channel is selected by the alternative construct, and runs. Displaying a message when a process runs confirms the correct behavior.

ComsTime

The ComsTime test (Figure A.7) is used in the OCCAM community to measure the context-switch and communication time. The new library on QNX performs one loop of the test in 63μs. In comparison, the old library on RTAI, on the same hardware, performs one loop in 35μs. The test has not yet support for measuring the context-switch and communication time separately.

5.2.3 Discussion

The traces made of the sequential models show that they are executed according to the specifications. The two processes in the parallel model execute in two separate threads, but because the PC104 has just one single core processor, only one thread can be running at any given time.

The ComsTime test indicates that the time it takes to switch threads in QNX is about two times higher than the highly optimized user-level thread switch in the old library. When using a lot of processes and short sampletimes, this could be a potential problem. The work of Bezemer (2008) describes methods to optimize the gCSP models, reducing the number of parallel processes, and thus the amount of necessary switching between threads.

During the design of the tests, the gCSP model in Figure 5.5 could be drawn. This is incorrect and should never be allowed to be drawn in gCSP. Rendezvous communication between two sequential processes is impossible and will deadlock. gCSP generates code from this model us-

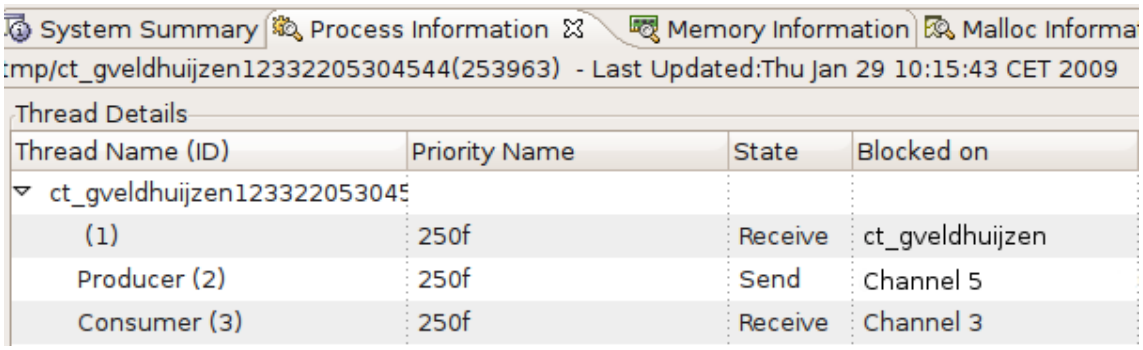


Figure 5.4: System information view indicating deadlock

ing a VarChannel (asynchronous communication), while the model still presents a rendezvous channel. The generated code for this situation and for explicit modeled VarChannels is incorrect.

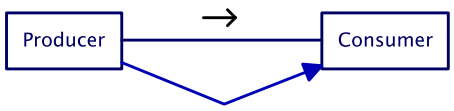


Figure 5.5: Incorrect gCSP model

5.3 Timing test

5.3.1 Description

The timing jitter of the TimerChannels using QNX timers is determined using kernel tracing. For verification with an oscilloscope, an output pin on the Anything I/O board is toggled on each period. A periodic timer of 1ms is requested. The used model is shown in Figure A.8.

5.3.2 Results

The exact timestamps of the timerticks are captured using kernel tracing and the jitter is plotted in Figure 5.6. The bulk of the measurements are below 2μs, which is confirmed with the oscilloscope. The latency from the point the hardware timer interrupt occurs, to the point the thread waiting for the timertick is unblocked, is extracted from the kernel trace and is summarized in Table 5.1. It averages around 7.7μs. Not displayed in the graph, but present in the measurements, is the missing of a timertick every 6535ms.

Event	**ns**
Interrupt entry	0
OS handler entry	600
OS handler exit	3100
Interrupt exit	4125
Timer pulse	6850
Thread running	7700

Table 5.1: Cummulative hardware interrupt latency

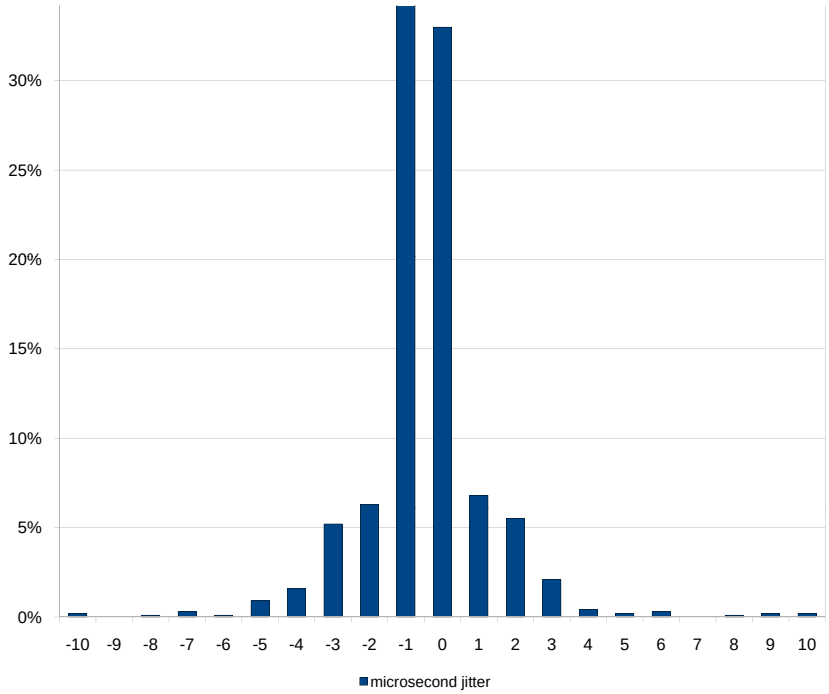


Figure 5.6: Jitter on a 1ms periodic timer (10000 samples)

5.3.3 Discussion

The timer test shows that the timing jitter is well below 1%, which is sufficient for most control applications at the CE-group. A similar test has been done by Lootsma (2008) using the same PC104 hardware, and thus the same hardware timer, but with RTAI as OS. His results are nearly identical, so presumably the jitter is the result of the limitations of the hardware, not from the OS. This shows that if the interrupt is handled immediately, the jitter is acceptable for most control applications.

The tests showed the missing of a timertick every 6535ms. This can be explained by looking at the timerchip used in x86 hardware. The timers are derived from a 1.1931816MHz clock which is divided by an *integer* divisor, before it is offered to the OS. As a result, the requested 1ms tick is not exactly 1ms, but 0.99847ms. After 6535 ticks, one tick will be missed, as can be seen in Figure 5.7. The lower row is the expected sequence, the upper row is the actual sequence of ticks. Tick 6535 will not be there, because internally the counter is still at 6534.999992. The missing of a timertick every 6535ms might not be a great issue for the systems at the CE-group, but for high accuracy applications like a waferstepper it could be a problem. If the design requires really precise timing, the only choice is to request a timer event of .999847ms and not 1ms, or use a different hardware platform.

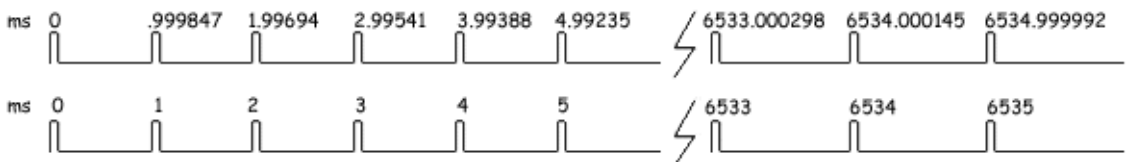


Figure 5.7: Difference between a 0.98847ms and 1ms clocktick. (QNX Software Systems, 2008)

The code generation from the used TimerChannel test gCSP model shown in Figure A.8 is not optimal. The modeled linkdrivers are not real processes, but are channel implementations.

gCSP still generates the parallel construct, but can only place the writer process in it, so it has only 1 child process. In the old library this does not cause extra overhead. In the new library for each child process in a parallel construct, a new thread is created, causing needless runtime overhead. For this test the generated code was edited by hand to remove the extra parallel constructs.

5.4 Production cell

5.4.1 Description

To test the combination of the software, timing and mechatronic hardware, the part of the production cell indicated by the red square in Figure 5.8 is used. The production cell setup is a model of a realistic plant, designed by van den Berg (2006), having several actuators that need to cooperate and synchronize their activities. The controllers are designed by Maljaars (2006) and implemented using 20-sim and gCSP.

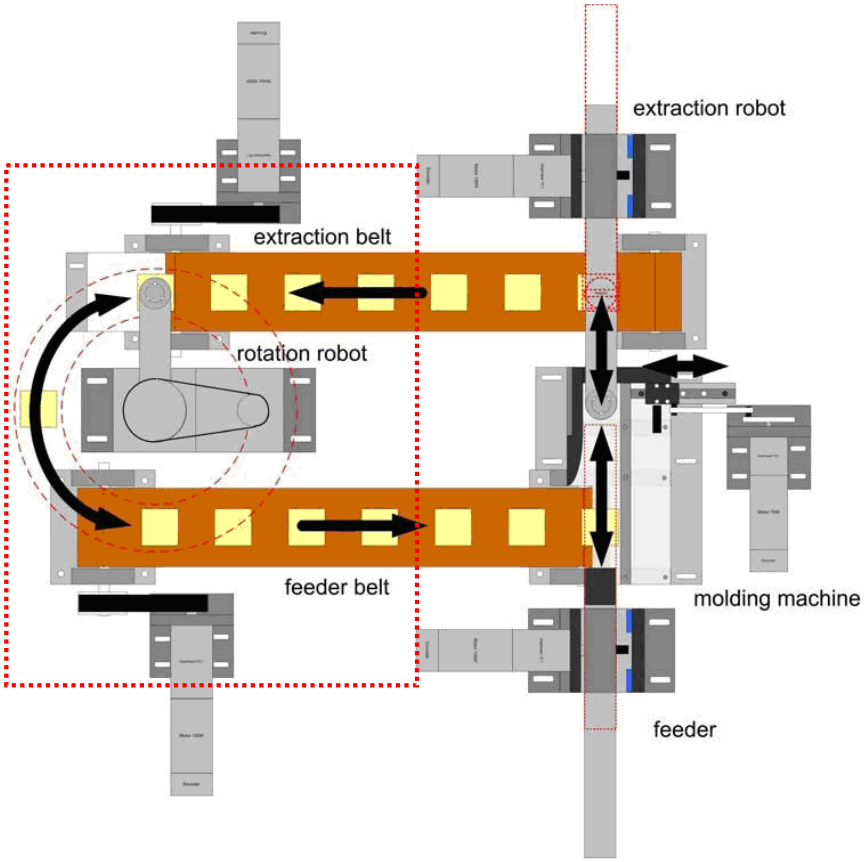


Figure 5.8: The production cell setup

The 20-sim models can still be used to generate valid C++ code. The gCSP model however, uses BufferedChannels, which are not yet implemented in the new library. The model lacks the sequence controller, which was implemented by hand by Maljaars.

For this test only a part of the production cell setup is used, namely the rotation robot, the feeder belt and the extraction belt. The belts are not actively controlled, just turned on or off. Parts of the model from Maljaars are recreated in gCSP, and slightly adapted to replace the missing BufferedChannels. The BufferedChannels were used by Maljaars to decouple the control loop from the motionprofile generator. This is at this moment accomplished using VarChannels. The controller for the rotation robot and the motionprofile generators are generated from

the 20-sim models from Maljaars and added to the gCSP model. The sequence controller is made by hand because gCSP is not suited for modeling sequence controllers. The gCSP model can be found in Figures A.9-A.15.

5.4.2 Results

The production cell runs smoothly. A small part of the kernel trace is shown in Figure 5.9. The control loop runs in the highlighted controller thread (1), at the highest priority. On every *tock* event, or timertick, the thread becomes READY, the running thread gets preempted and the controller runs. The loop finishes well before the next occurrence of a *tock* event. The control loop finshes in 70μs, an entire cycle including the calculation of the motionprofile finshes in 650μs. Analysis of all the recorded timestamps show a jitter comparable to Section 5.3.

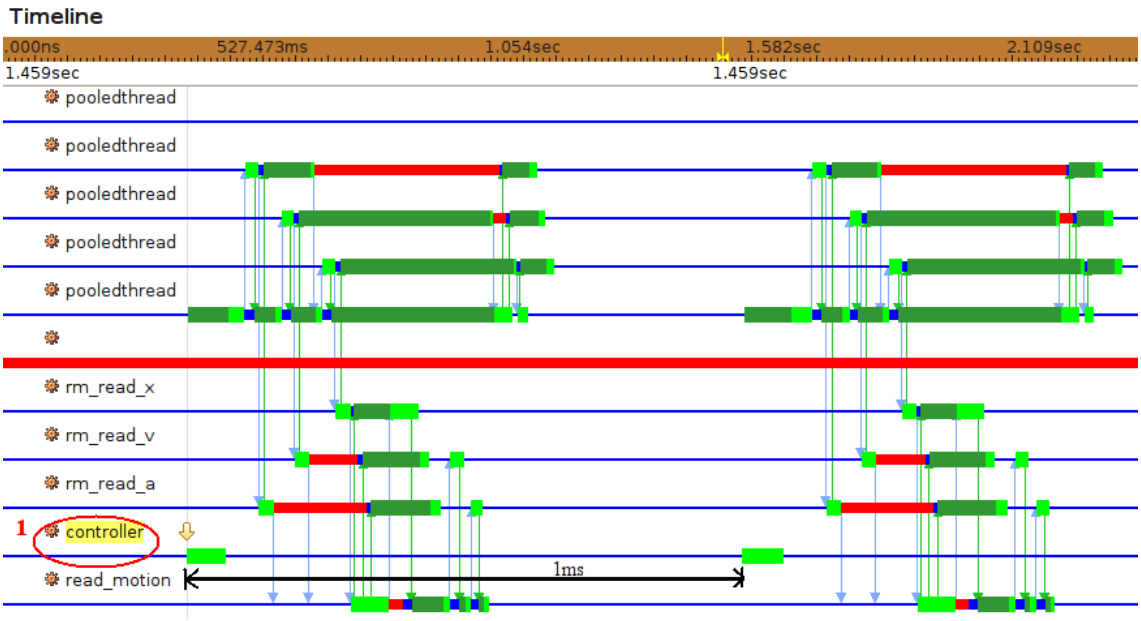


Figure 5.9: Timeline trace of the production cell setup

5.4.3 Discussion

The part of the production cell setup that has been implemented performs according to the specifications. The most important task, the control loop, runs uninterrupted and other processes are preempted if the control loop wants to run. This ensures the timing requirements of the control loop are met.

The code generated from the gCSP submodel shown in Figure A.12 is not correct. The code generator does not recognize the combination of an Any2One channel and a LinkDriver, and only generates the Any2One channel. This part of the code was edited by hand.

The trace shows that the control loop, the submodel in Figure A.15, finishes in 70μs. The remainder of the cycle time is spent calculating the next motionprofile output, and most of all communicating the values towards the control loop. The gCSP model is not optimal and needs to be optimized before the remaining part of the production cell can be implemented. The output of the motion profile generator process is communicated in parallel over three separate channels towards the controller, which is one process. Communicating the output over one channel at once will save a lot of threadswitches and time. A container channel, or composite channel could be used to make the communication more efficient.

The current total cycle time of 650μs prevents the implementation of the entire production

cell, containing a total of 6 control units, on one PC104 using the new CT-library and QNX. If the communication in the model can be optimized, the total cycle time for one controller will go down to approximately 100-140μs. This is sufficient to implement the entire production cell on one PC104.

The old library has timing problems while controlling the entire production cell when it becomes filled with blocks. To determine if the new library and QNX are able to perform better, the entire setup has to be modeled again in gCSP. The model of Maljaars should be followed more closely, but needs some adaptation before it can be compiled against the new library.

5.5 Conclusions

The functionality tests show that the new library executes the CSP constructs and communication events correctly and in the right order. Code generated from gCSP models can be compiled directly against the new library. The kernel event tracing proved a valuable tool for tracing the application and system and giving detailed information about the execution times and order. Although the switching between threads is slower compared to the old library, this will only be of concern when there are many different CSP processes running in parallel and the sample times are short. The work of Bezemer (2008) shows promising results in optimizing those models.

The timing tests showed that the library and QNX can reliably implement the *tock* event, although care has to be taken when a very high precision is needed on x86 hardware.The implementation of a part of the production cell setup showed that the original 20-sim models can be used without adaptation on the new library. The gCSP models had to be adjusted to circumvent the BufferedChannels, which are not yet implemented. Tracing the execution of the application proved it met the timing requirements, however to compare it to the original implementation using the old library and RTAI, the entire setup has to be modeled and deployed using the new library.

6 Conclusion and recommendations

6.1 Conclusions

The goal of this project was to redesign the current CSP execution engine, the CT-library, to solve the problems with blocking system calls and timing. Analysis of the problems results in the choice to remove the threading and scheduler from the library, and hand it over to the operating system. A priority based preemptive scheduler retains the correct CSP scheduling behavior, and makes deterministic response to interrupts possible.

The microkernel architecture meets the requirements for dependable real-time software and is compatible with the synchronization method of CSP. The choice is made for QNX Neutrino, a microkernel based real-time operating system. Added benefits are an instrumented kernel, an integrated development environment, adaptive partitioning scheduling and very extensive documentation.

The new CT-library is structured to let the operating system provide as much functionality as possible. QNX provides the rendezvous channels, the threading mechanism and the scheduler needed for the library. The most often used CSP constructs are implemented in the library, using the QNX functionality. It is shown that kernel event tracing using the instrumented kernel is a non-invasive way to acquire information about an application, without affecting the real-timeliness of the system. Qnet extends the rendezvous channels over a network link, making the implementation of distributed processing using RemoteChannels easy. Finally APS is used to give the control application a guaranteed percentage of CPU time, and make remote debugging possible even if the system is fully loaded.

To demonstrate the correct behavior of the library, several functional tests are modeled in gCSP and tested with the library. Using specific timer tests and the production cell setup it is shown that QNX and the new library are suitable for real-time control using the design trajectory used at the CE-group.

6.2 Recommendations

Not the entire feature set of gCSP and the corresponding CSP constructs are implemented. Specifically the PriAlt and SKIP guards should be implemented because the more advanced existing models use them.

The BufferedChannel is not implemented and should be reconsidered for adding asynchronous communication, aside the existing VarChannels.

The model of the production cell setup should be completed and deployed using the new library to be able to make a better comparison between the old library and the new one.

During the tests performed in Chapter 5, some problems with the code generation of gCSP were found which should be resolved:

- The generation of (parallel) constructs containing only one child process is needless, and generates extra runtime overhead.
- gCSP allows to draw invalid models, which generates incorrect code
- Code generation of VarChannels is incomplete
- Linkdrivers with multiple readers or writers are not generated correctly

The use of kernel level threads make context switching a more expensive operation. The work of Bezemer (2008) should be continued to optimize the amount of parallel constructs in a model.

A benefit of kernel level threads is the possibility to use multi core, or multi processor systems. The library supports true parallelism, but this has not been tested. Further research is needed

on this subject.

Kernel event tracing provides a wealth of information about a running system. The QNX Foundation Classes should be examined and used to give runtime information about the system and eventually make it possible to use the animation framework of van der Steen et al. (2008) again.

The newest version of QNX Neutrino, 6.4.0 provides substantial performance improvements, as well as major improvements in the IDE and tracing functionality. This new version should be used in further development.

The use of Adaptive Partitioning Scheduling should be further investigated. In version 6.4.0 of QNX Neutrino, the option is added to be notified when a partition exceeds its budget. This way APS could be used for deadline monitoring.

A gCSP models

In this appendix the models can be found which are used in Chapter 5.

A.1 Functional test models

(a) Sequential (b) Parallel (c) PriParallel

Figure A.1: Simple construct examples

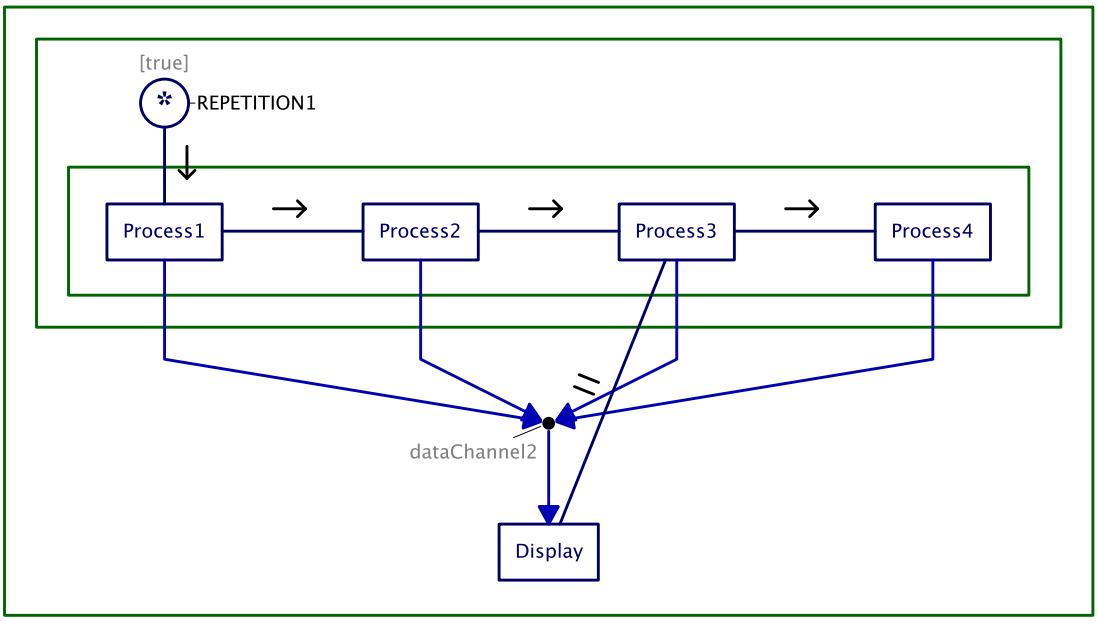


Figure A.2: Sequential with communication test

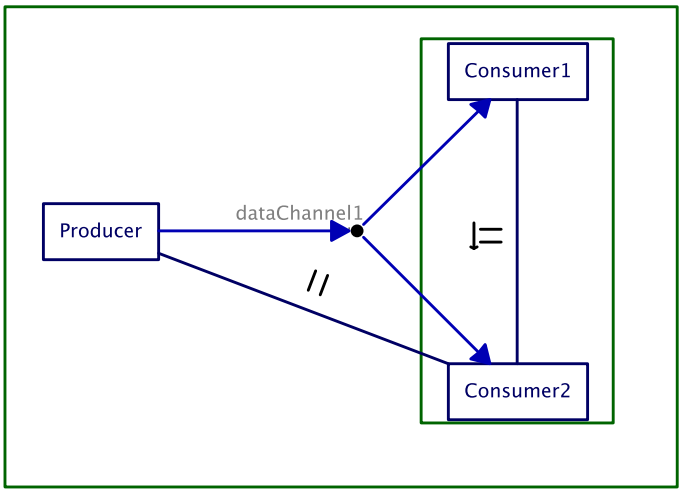
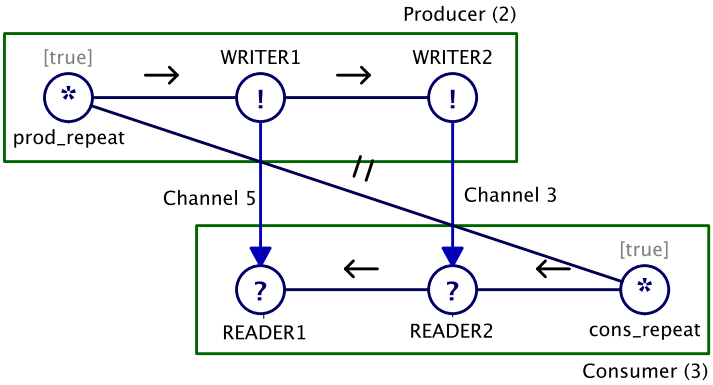
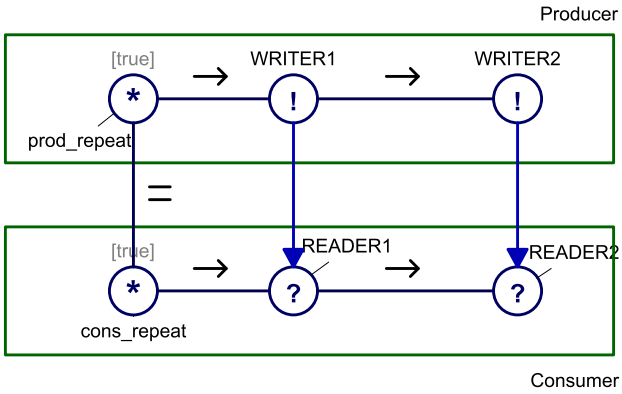


Figure A.3: Complex Pri-parallel test

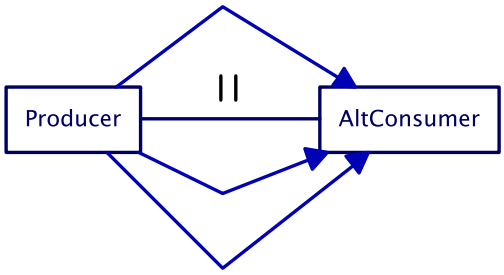


(a) Deadlock

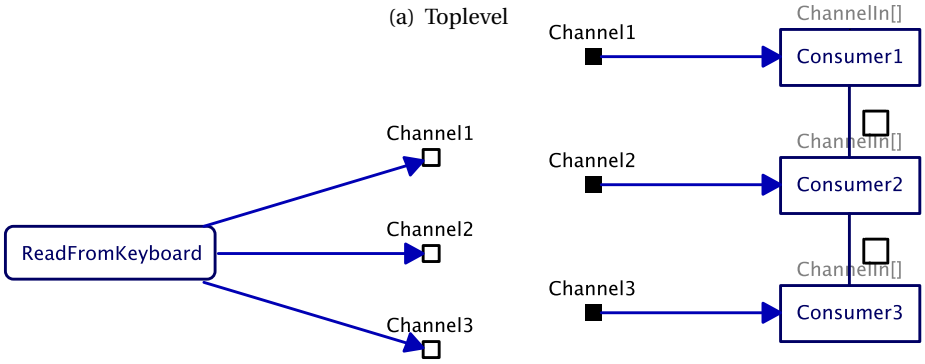


(b) Deadlock-free

Figure A.4: Producer Consumer example



(a) Toplevel



(b) Producer (c) AltConsumer

Figure A.5: Alting with inputguards example

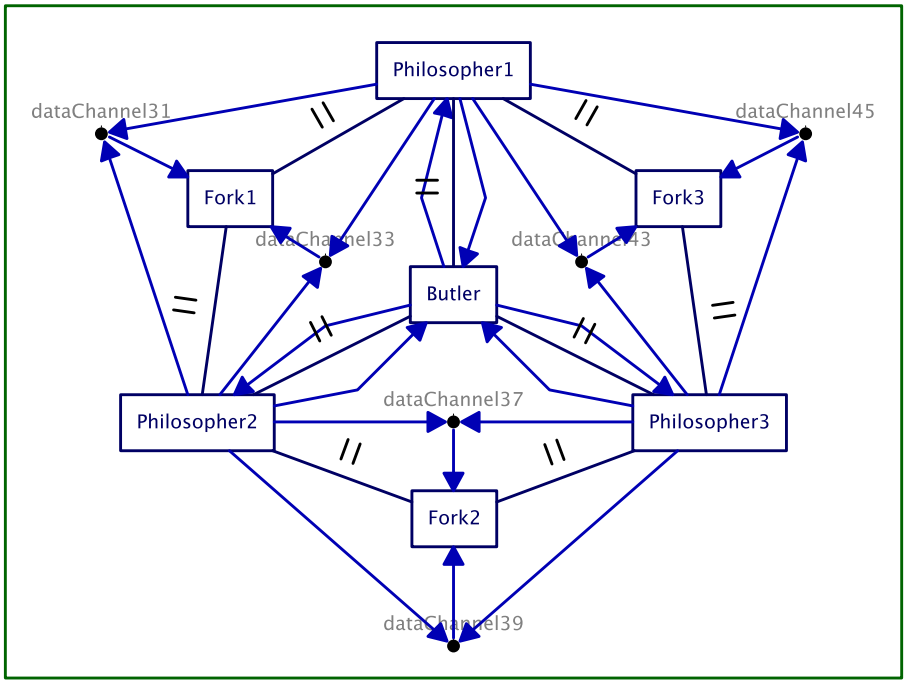


Figure A.6: Dining philosophers with butler test

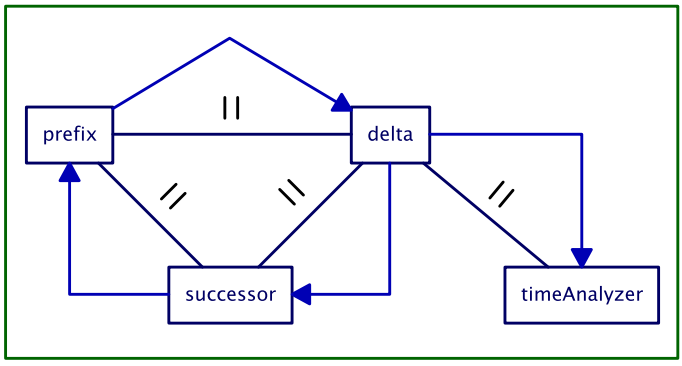


Figure A.7: ComsTime test

A.2 Timing tests models

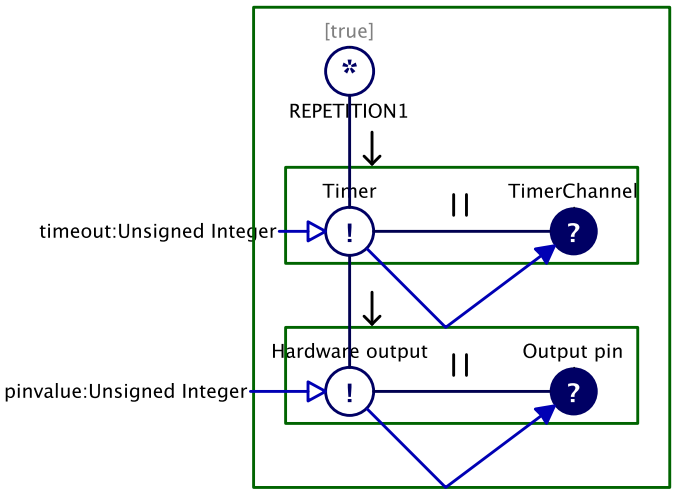


Figure A.8: TimerChannel test

A.3 Production cell model

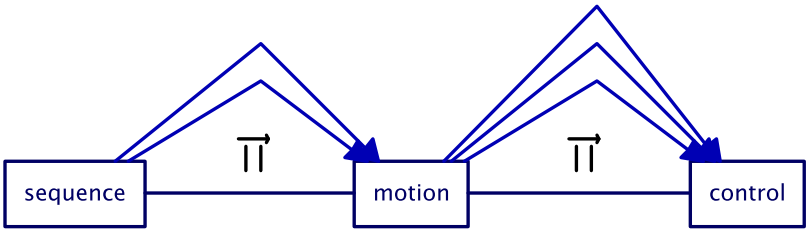


Figure A.9: Toplevel

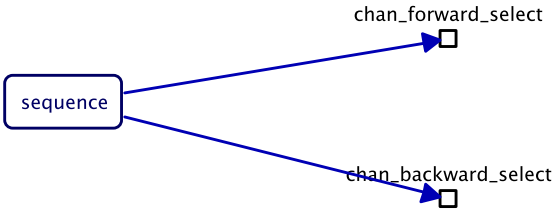


Figure A.10: Sequence controller

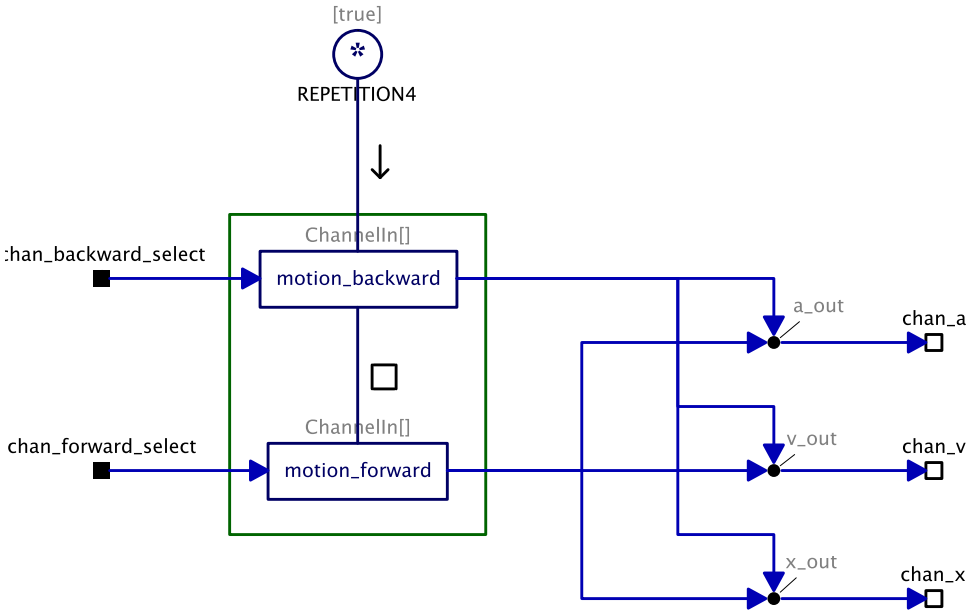


Figure A.11: Motion generator

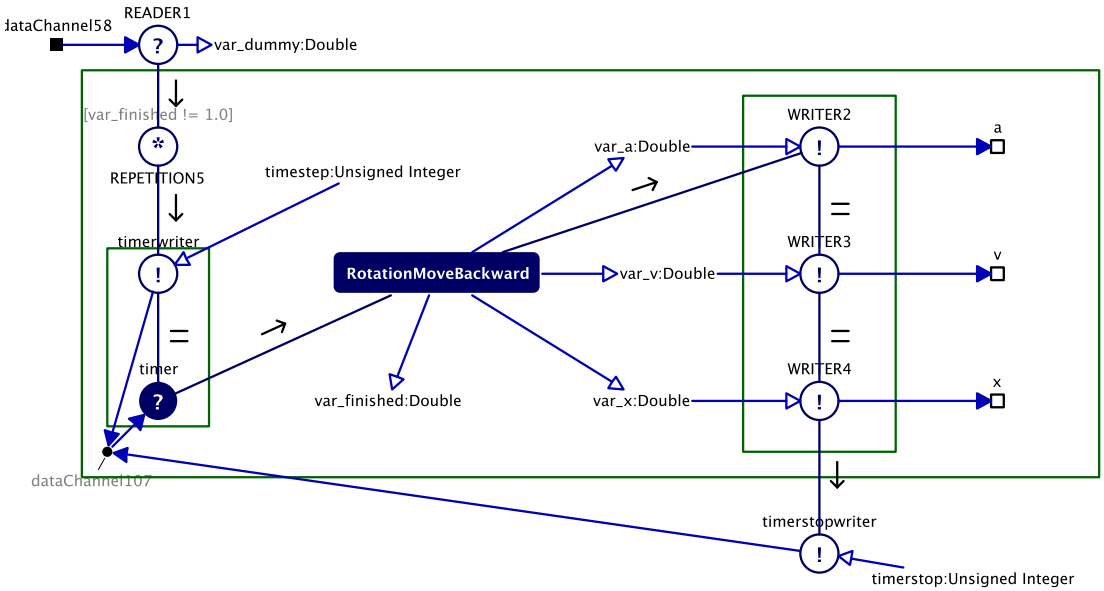


Figure A.12: Motion backward generator

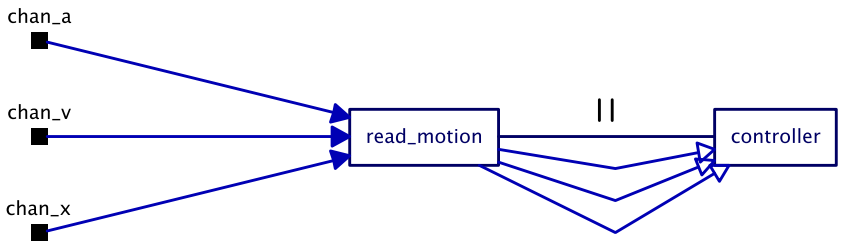


Figure A.13: Control process

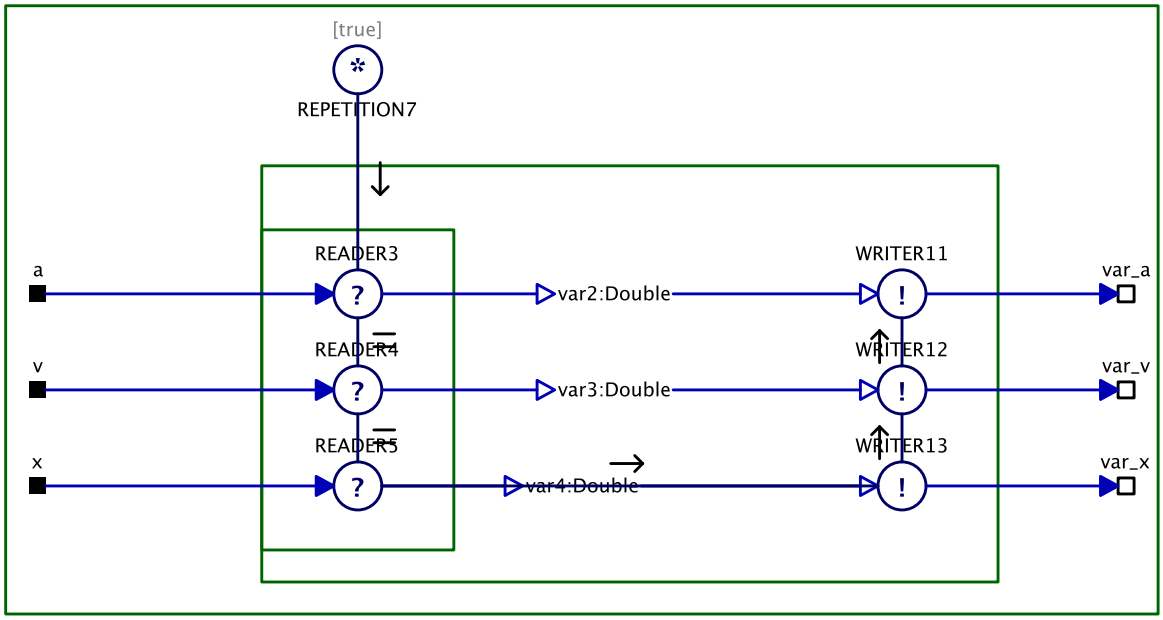


Figure A.14: read motion process

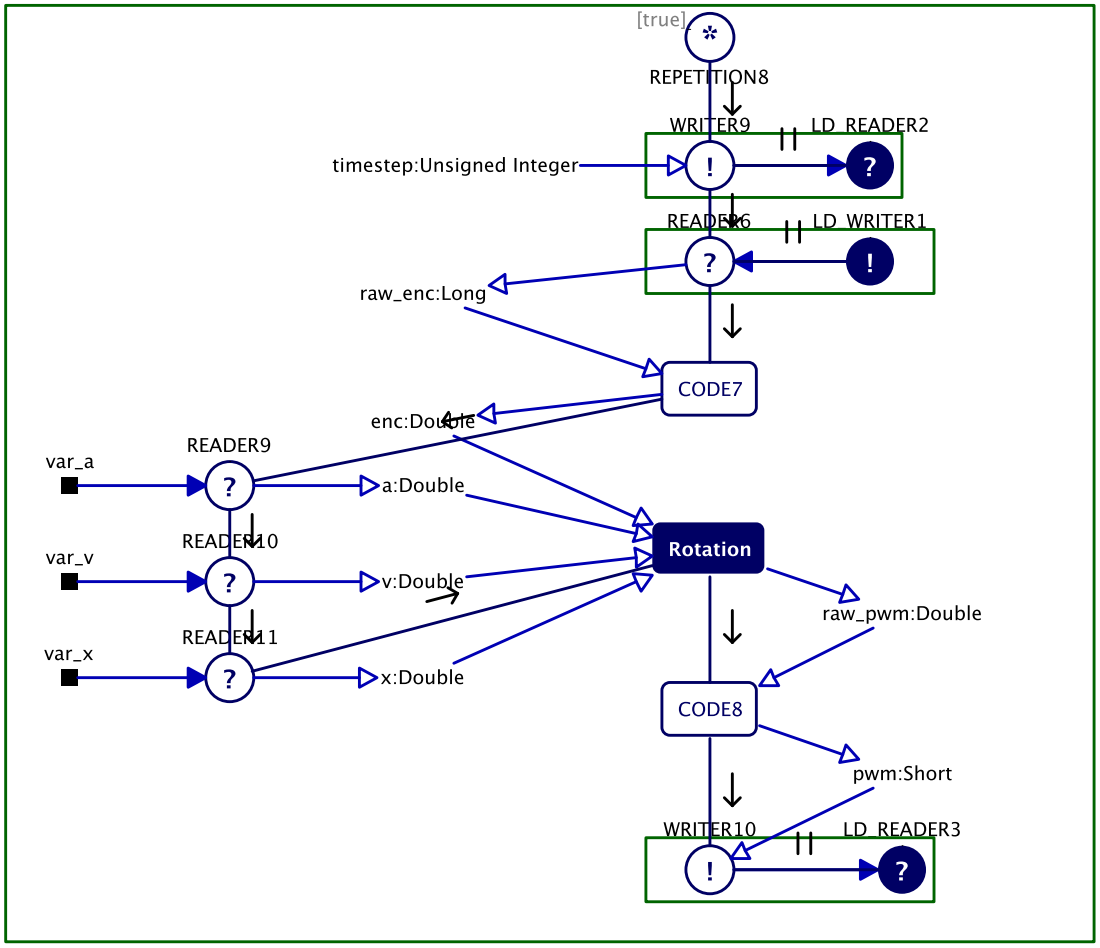


Figure A.15: Controller

B Compiling the ct-library

B.1 Introduction

The library is developed in the QNX Momentics IDE 4.5, using the GCC 3.3.5 compiler. The source can be found in svn, or in the files that come with this report. There is one dependency, the QNX Foundation Classes (QFC) (Allen, 2008), which are still continuously improving.

B.2 Checking out the source

If you do want to check out the source, use the IDE. Switch to the SVN Repository (**Window -> Open Perspective**) and add the repository. At this moment the source can be found in the develop svn, or in the files deliverd with this report. Check out the ct-library.

Next, add the QFC repository, as mentioned in the tutorial on the website. Checkout the sigc++ and qfc projects. Switch to the C/C++ perspective and set the correct build configuration for the two projects by right-clicking on the project and choosing **Build Configuration** and selecting **ntox86-gcc-3.3.5-debug**. Now first build the sigc++ project, and afterwards the qfc project. Transfer the two created libraries to the target and place them in `/lib`.

B.3 Compiling the library

The settings for the ct-library project should already be correct, otherwise add the the sigc++ and qfc library-headers to the configuration of the ct-library by opening the properties of the project. The headers of the libraries should be added under C/C++ Build -> Settings -> QCC Compiler -> Preprocessor and can be referenced from the existing projects. Add also the include directory of the ct-library to the Preprocessor. Next compile the ct-library.

B.4 Using the library

To create an application with the ct-library, create a new QNX C++ program. Give it a name in the dialog and on the next window, select the required build variants. Finish the dialog. Next edit the properties of the project. On the QNX C/C++ Project tab you can turn on profiling. On the Compiler tab select the correct compiler and add as extra include paths from the ct-library project the include directory, from the qfc and sigc++ projects the public directory. On the Linker tab select the Extra Libraries from the dropdown and add the libraries from the ct-library, qfc, and sigc++ libraries. Now the application can be compiled.

The basic BuildingBlocks used in the old library can still be used. They are available in the ct-library source-tree. To use them, copy the directory to your project, and add the source and header files to the build process.

Some examples are available in the examples directory in the ct-library source-tree. There is not yet an easy way to build them, so copy the desired example to a new project and compile it.

C Kernel event tracing

This appendix shows how to create a kernel image with the instrumented kernel and how to use kernel event tracing from the QNX Momentics IDE.

C.1 Configuring the instrumented kernel

To see if the normal kernel or the instrumented version is running, look in `/proc/boot` for the file `procnto`. If it is there, the normal kernel is running. If `procnto-inst` is there, the instrumented kernel is running.

To create a bootfile with the instrumented kernel, the following buildfile can be used:

```
#
# The build file for QNX Neutrino booting on a PC
#
[virtual=x86,bios +compress] boot = {
        # Reserve 64k of video memory to handle multiple video cards
        startup-bios

        # PATH is the *safe* path for executables (confstr(_CS_PATH...))
        # LD_LIBRARY_PATH is the *safe* path for libraries
        # (confstr(_CS_LIBPATH))
        #    i.e. This is the path searched for libs in
        #    setuid/setgid executables.
        PATH=/proc/boot:/bin:/usr/bin:/opt/bin \
            LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \
            procnto-instr
}

[+script] startup-script = {
        # To save memory make everyone use the libc in the boot image!
        # For speed (less symbolic lookups) we point to libc.so.2
        # instead of libc.so
        procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2

        # Default user programs to priorty 10, other scheduler (pri=10o)
        # Tell "diskboot" this is a hard disk boot (-b1)
        # Tell "diskboot" to use DMA on IDE drives (-D1)
        # Start 4 text consoles buy passing "-n4" to "devc-con" (-o)
        # By adding "-e" linux ext2 filesystem will be mounted as well.
        [pri=10o] PATH=/proc/boot diskboot -b1 -D0 \
                -odevc-con,-n4 -odevc-con-hid,-n4
}

# Include the current "libc.so". It will be created as a real file
# using its internal "SONAME", with "libc.so" being a symlink to it.
# The symlink will point to the last "libc.so.*" so if an earlier
# libc is needed (e.g. libc.so.1) add it before the this line.
libc.so
libhiddi.so
libusbdi.so
```

```
# Include all the files for the default filesystems
libcam.so
io-blk.so
cam-disk.so
fs-qnx4.so

# USB for console driver
devu-ehci.so
devu-ohci.so
devu-uhci.so
devh-usb.so
devh-ps2ser.so


# These programs only need to be run once from the boot image.
# "data=uip" will waste less memory as the ram from the boot
# image will be used directly without making a copy of the data
# (i.e. as the default "data=cpy" does). When they have been
# run once, they will be unlinked from /proc/boot.
[data=copy]
seedres
pci-bios
devb-eide
diskboot
slogger
fesh
devc-con
devc-con-hid
io-usb
io-hid
```

To create the bootfile, execute the following command:

```
mkifs buildfile bootimage.ifs
```

Replace the current boot image with the new one. First copy the current (working) bootfile to the alternate bootimage, then copy the new one.

```
cp /.boot /.altboot
cp bootimage.ifs /.boot
```

Note that the .boot and .altboot files cannot be moved or deleted, only replaced. After a reboot the system will be running the instrumented kernel.

C.2 Using the IDE for kernel tracing

You can gather trace events from the instrumented kernel in two different ways. You can run a command-line utility (e.g. tracelogger) on the target to generate a log file, and then transfer that log file back to the development environment for analysis. Or, you can capture events directly from the IDE using the Log Configuration dialog.

The easiest way to use kernel even tracing is from the IDE. Make sure the target is connected to the IDE, otherwise start `qconn` on the target.

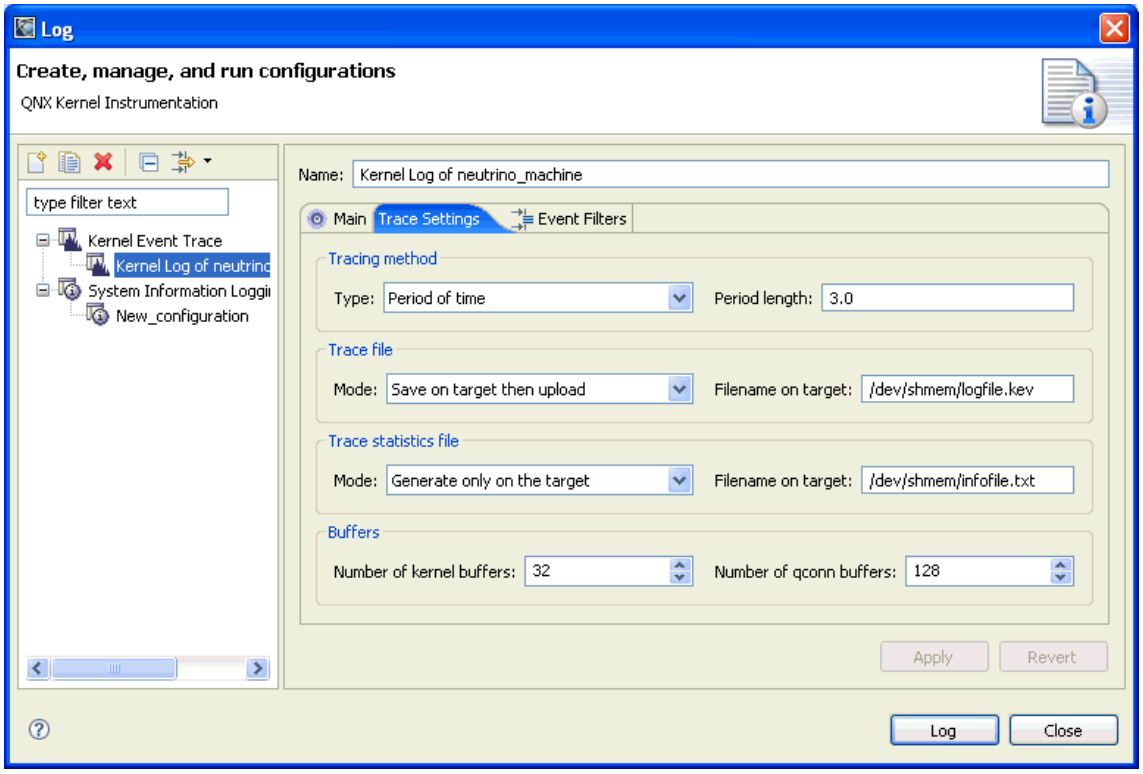


Figure C.1: Log Configuration dialog

In the Target Navigator view, right-click a target, then select **Log With...–>Kernel Event Trace** from the menu. If you don't have the Target Navigator view open, choose **Window–>Show View–>Other...**, then **QNX Targets–>Target Navigator**.

The Log Configuration dialog as in Figure C.1 will show. Do not set the Period of time too long, the amount of data will get very large. On the Event Filters tab you can specify what will be traced. When you click **Log**, the logging will start and afterwards the logfile will be transfered to the IDE. If it does not open automatically, open it and switch to the System Profiler Perspective. There a summary of the tracelog is given. If you switch to the timeline view, a detailed view of all the processes and threads can be seen, including the communication, threadstates, thread-names etc. An example is given in Figure C.2. For more information about kernel tracing and the IDE see the guides on the QNX website.

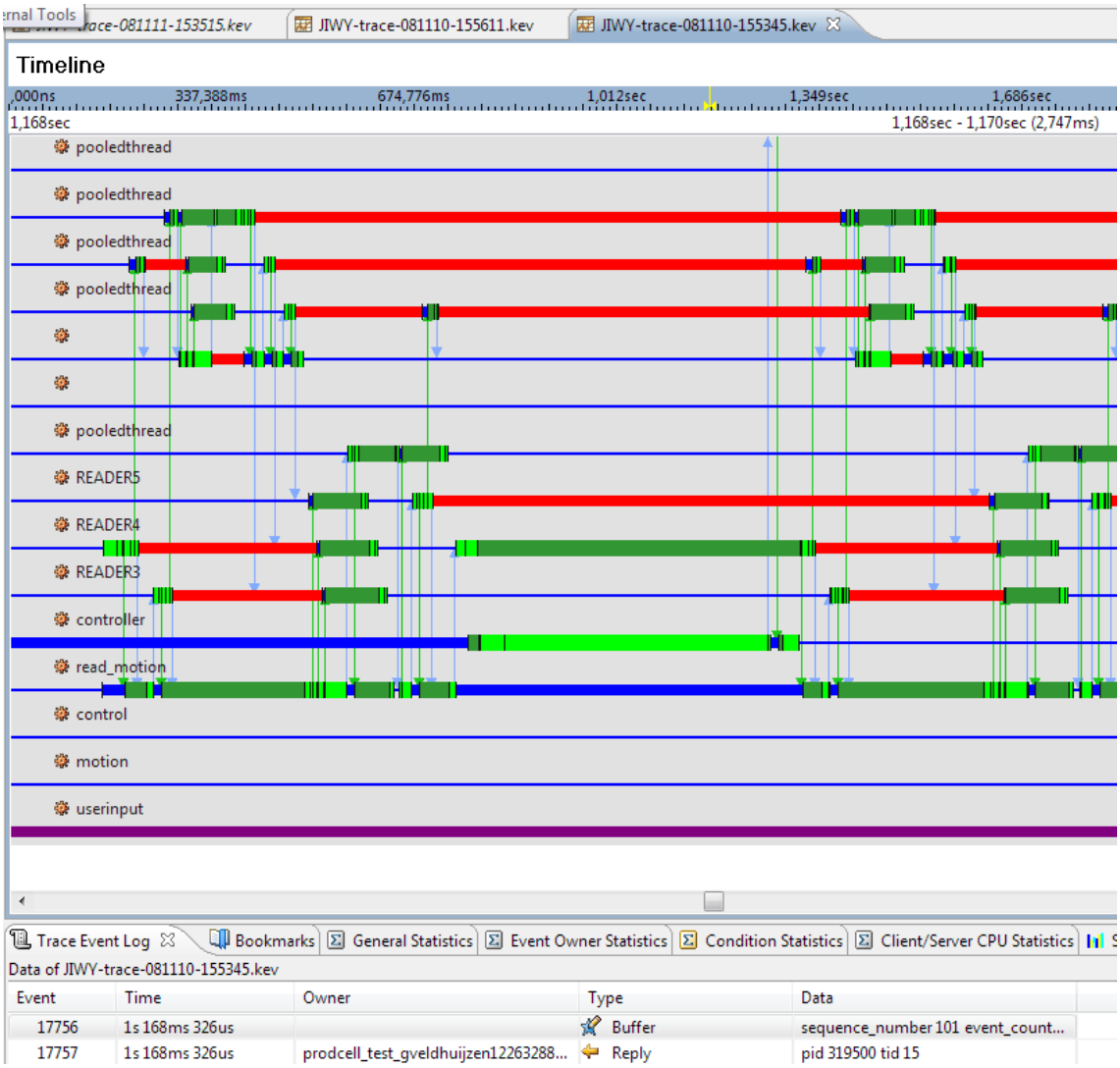


Figure C.2: Kernel event tracing timeline

D Adaptive Partitioning Scheduler

To use the adaptive partitioning scheduler the bootimage has to be modified. The buildfile from Appendix C can be used. The attribute `[module=aps]` has to be added to the command that launches `procnto`. In case of the mentioned buildfile it becomes:

```
[module=aps] PATH=/proc/boot:/bin:/usr/bin:/opt/bin \
LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \
procnto-instr
```

Rebuild the bootimage and copy it to the target, as mentioned in Appendix C.

To see if APS is activated correctly use `aps show`. If it is it will show something like:

```
$ aps show
                     +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System             0 |   100% |  36.24% |  100ms |   0.000ms
--------------------+------------------+-------------------
Total                |   100% |  36.24% |
```

At this moment there is only the default System partition which has a budget of 100%. From the command line new partitions can be created. For example:

```
aps create -b10 Debug
```

will create a partition named Debug, with a budget of 10%. To start a program in the newly created partition use the following command:

```
on -Xaps=Debug program
```

D.1 Remote debugging

APS can be used to ensure a system can always be reached and debugged remotely, even if some high priority process is utilizing the entire system (Say 100% CPU time). Therefore we need to edit the buildfile again, and add the following lines after ######

```
# Create an example scheduler partition
# Create a 10% p "Debugging"
sched_aps Debugging 10

# Start qconn in the Debugging partition
[sched_aps=Debugging] /usr/sbin/qconn
# Start telnet in the Debugging partition
[sched_aps=Debugging] /usr/sbin/telnetd -debug&
```

Because we want to use remote debugging, the `io-net` process has to be run in the debugging partition as well. This requires a bit more work, a special executable has to be created. This is the source code:

```
#include <stdio.h>
#include <sys/procmgr.h>
```

```
// start io-net for enumeration purposes, putting it into a partition

main(int argc, char *argv[])
{
system ("/bin/on -X aps=Debugging /sbin/io-net -ptcpip");
system ("/bin/waitfor /dev/io-net");

// move to background
procmgr_daemon(0,0);

sleep(60); // wait for 1 minute
exit(0);
}
```

Compile it with:

```
qcc startnet.c -o startnet
```

Copy the binary `startnet` to `/sbin/startnet` on the target. Next edit the file `/etc/system/enum/include/net` and change the line `set(IONET_CMD, io-net -ptcpip)` to `set(IONET_CMD, "startnet")`.

This extra operation is required because every time that the enumerator code decides to start a new network interface, it first looks to see if the program pointed to by IONET_CMD is running and starts it if it is not. If it is already running, then instead of starting a new instance, it uses `mount -Tio-net` to add the new interface to the existing io-net process. If startnet did not wait around, then it would get started again, and start up multiple io-net processes, which is not desired.

The final thing you need to do is to move the line in the build file that creates the partition, so that it is created before io-net gets started by diskboot. The bootfile will become:

```
#
# The build file for QNX Neutrino booting on a PC
#
[virtual=x86,bios +compress] boot = {
        # Reserve 64k of video memory to handle multiple video cards
        startup-bios

        # PATH is the *safe* path for executables (confstr(_CS_PATH...))
        # LD_LIBRARY_PATH is the *safe* path for libraries \
        # (confstr(_CS_LIBPATH))
        #    i.e. This is the path searched for libs in
        #    setuid/setgid executables.
        # The module=aps enables the adaptive partitioning scheduler
        [module=aps] PATH=/proc/boot:/bin:/usr/bin:/opt/bin \
            LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \
            procnto-instr
}

[+script] startup-script = {
        # To save memory make everyone use the libc in the boot image!
        # For speed (less symbolic lookups) we point to
        # libc.so.2 instead of libc.so
```

```
        procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2

        # Create an example scheduler partition
        # Create a 10% partition named "Debugging"
        sched_aps Debugging 10

        # Default user programs to priorty 10, other scheduler (pri=10o)
        # Tell "diskboot" this is a hard disk boot (-b1)
        # Tell "diskboot" to use DMA on IDE drives (-D1)
        # Start 4 text consoles buy passing "-n4" to "devc-con" (-o)
        # By adding "-e" linux ext2 filesystem will be mounted as well.
        [pri=10o] PATH=/proc/boot diskboot -b1 -D0 \
                -odevc-con,-n4 -odevc-con-hid,-n4

        # Start qconn in the Debugging partition
        [sched_aps=Debugging]/usr/sbin/qconn
        # Start telnet in the Debugging partition
        [sched_aps=Debugging]/usr/sbin/telnetd -debug&
}

# Include the current "libc.so". It will be created as a real file
# using its internal "SONAME", with "libc.so" being a symlink to it.
# The symlink will point to the last "libc.so.*" so if an earlier
# libc is needed (e.g. libc.so.1) add it before the this line.
libc.so
libhiddi.so
libusbdi.so

# Include all the files for the default filesystems
libcam.so
io-blk.so
cam-disk.so
fs-qnx4.so

# USB for console driver
devu-ehci.so
devu-ohci.so
devu-uhci.so
devh-usb.so
devh-ps2ser.so


# These programs only need to be run once from the boot image.
# "data=uip" will waste less memory as the ram from the boot
# image will be used directly without making a copy of the data
# (i.e. as the default "data=cpy" does). When they have been
# run once, they will be unlinked from /proc/boot.
[data=copy]
seedres
pci-bios
devb-eide
```

```
diskboot
slogger
fesh
devc-con
devc-con-hid
io-usb
io-hid
```

Create a new bootimage from the buildfile and transfer it to the target. Reboot and the new remote debugging in a separate partition is available.

D.2 Using APS from source code

The partitions can be modified from an application and an application can put its threads in specific partitions.

To create partitions and add the application to it the following code can be used:

```
// First join the System partition, otherwise we can't make new partitions
sched_aps_join_parms join_data;
memset(&join_data, 0, sizeof(join_data));
join_data.id = 0;
join_data.pid = 0;
join_data.tid = 0;
int ret = SchedCtl( SCHED_APS_JOIN_PARTITION, &join_data,
        sizeof(join_data));
if (ret != EOK)
{
   printf("Couldn't join partition %d: %s (%d).\n",
        join_data.id, strerror(errno), errno);
}
else
{
   printf ("Now in partition %d.\n", join_data.id);
}

// Create new partition for ourself
int partition_id;
sched_aps_create_parms creation_data;
memset(&creation_data, 0, sizeof(creation_data));
creation_data.budget_percent = 80;
creation_data.critical_budget_ms = 10;
creation_data.name = "ControlLoop";
ret = SchedCtl( SCHED_APS_CREATE_PARTITION, &creation_data,
        sizeof(creation_data));
if (ret != EOK)
{
   std::cout << "Couldn't create partition \""
                << creation_data.name << "\": "
                << strerror(errno) << " (" << errno << ")."
                << std::endl;
        // If the partition already exists,
        // we need to find out the id of it
```

```
        sched_aps_lookup_parms lookup_data;
        memset(&lookup_data, 0, sizeof(lookup_data));
        lookup_data.name = "ControlLoop";
        ret = SchedCtl( SCHED_APS_LOOKUP, &lookup_data,
        sizeof(lookup_data));
        if (ret != EOK)
        {
                std::cout << "Couldn't find partition \""
                << lookup_data.name << "\": "
                << strerror(errno) << " (" << errno << ")."
                << std::endl;
                //exit(-1);
        }
        partition_id = lookup_data.id;
}
else
{
   std::cout << "The new partition's ID is "
        << creation_data.id << "." << std::endl;
   partition_id = creation_data.id;
}
// Join the ControlLoop partition which we just created, or lookup
memset(&join_data, 0, sizeof(join_data));
join_data.id = partition_id;
join_data.pid = 0;
join_data.tid = 0;
ret = SchedCtl( SCHED_APS_JOIN_PARTITION, &join_data,
        sizeof(join_data));
if (ret != EOK)
{
   printf("Couldn't join partition %d: %s (%d).\n",
   join_data.id, strerror(errno), errno);
}
else
{
   printf ("Now in partition %d.\n", join_data.id);
}
```

E Qnet

Qnet is the protocol used by QNX to implement transparent distributed processing. The protocol assumes a trusted network, so do not connect it directly to the internet or UTnet. For information about the exact working of Qnet, look at the website of QNX.

E.1 Configuring

This how-to assumes a setup using ethernet as the network link.

The easiest way to start Qnet at boottime is by creating a `useqnet` file. Log in as root and execute the following:

```
touch /etc/system/config/useqnet
```

If you want to start Qnet yourself the following applies. To use Qnet over ethernet the driver has to be started first, after which the Qnet driver can be started:

```
mount -Tio-net devn-<network-driver> &
mount -Tio-net /usr/lib/npm-qnet.so &
```

At this point other Qnet enabled hosts appear in the `/net` directory. If the `/net` directory does not exits, Qnet is not running. A listing of `/net/node_name` shows the pathname of the specific node.

E.2 Using Qnet

To start a process on another node:

```
on -f none_name <command>
```

To gather diagnostic process information of qnet:

```
cat /proc/qnetstats | grep -e "ok" -e "bad"
```

QNX allows for direct resource access across different nodes. To access a remote anything I/O card a simple program will suffice. The file descriptor can be used as a channel for the msgSend and msgRecieve as well as read(),write(),devctl(). The following codesnippet is an example:

```
/*Open a serial device on a different node
fd = open("/net/node1/dev/ser1", O_RDWR);
if( fd == -1 )
{
        fprintf( stderr, "Unable to open remote device: %s\n",
        strerror( errno ) );
        return EXIT_FAILURE;
}

ret = MsgSend( fd, &msg, sizeof( msg ), msg_reply, 255 );
if( ret == -1 )
{
        fprintf( stderr, "Unable to MsgSend() to remote device: %s\n"
                , strerror( errno ) );
        return EXIT_FAILURE;
}
```

```
printf( "client: remote device replied: %s\n", msg_reply );

close( fd );
```

Bibliography

Adeos Project (2004), Adeos.
http://home.gna.org/adeos/

Allen, R. (2008), QNX Foundation Classes.
http://community.qnx.com/sf/projects/qfc

van den Berg, B. (2006), Design of a Production Cell Setup, MSc Thesis 016CE2006, Control Laboratory, University of Twente.

Bezemer, M. (2008), Analyzing gCSP models using runtime and model analysis algorithms, MSc report 034CE2008, Control Laboratory, University of Twente.

Broenink, J. (1999), 20-Sim software for hierarchical bond-graph/block-diagram models, *Simulation Practice and Theory*, **7**, pp. 481–492.

Broenink, J. F., M. A. Groothuis, P. M. Visser and B. Orlic (2007), A Model-Driven Approach to Embedded Control System Implementation, in *2007 Western Multiconference on Computer Simulation*, SCS, San Diego, CA.

Broenink, J. F. and G. H. Hilderink (2001), A structured approach to embedded control systems implementation, in *2001 IEEE International Conference on Control Applications*, Eds. M. Spong, D. Repperger and J. Zannatha, IEEE, México City, México, pp. 761–766.
http://www.ce.utwente.nl/rtweb/publications/2001/pdf-files/042R2001.pdf

Brown, N. and P. Welch (2003), An Introduction to the Kent C++CSP Library, in *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, Eds. J. Broenink and G. Hilderink, IOS Press, Amsterdam, The Netherlands, volume 61 of *Concurrent Systems Engineering Series*, pp. 139–156, ISBN 1-58603-381-6, ISSN 1383-7575.
http://www.cs.kent.ac.uk/pubs/2003/1784

Brown, N. C. C. (2007), C++CSP2: A Many-to-Many Threading Model for Multicore Architectures, in *Communicating Process Architectures 2007*, Eds. A. A. McEwan, W. Ifill and P. H. Welch, pp. 183–205, ISBN 978-1586037673.

Controllab Products (2008), 20-sim.
http://www.20sim.com

Cooling, J. (2000), *Software Engineering for Real-Time Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0201596202.

Damstra, A. (2008), Virtual prototyping through co-simulation in hardware/software and mechatronics co-design, MSc report 005CE2008, Control Laboratory, University of Twente.

Deen, B. (2007), CT - I/O linkdriver with COMEDI, Pre-doctoral assignment-Report 009CE2007, Control Laboratory, University of Twente.

DIAPM (2008), RealTime Application Interface for Linux.
http://www.rtai.org

Formal Systems (Europe) Limited (2008), FDR2.
http://www.fsel.com/software.html

Hendriks, A. (1998), Design of a Realtime and Embedded Scheduler in Java, MSc 057R98, Control Laboratory, University of Twente.

High Integrity Systems (2008), OpenRTOS.
http://openrtos.highintegritysystems.com

Hilderink, G., J. Broenink and A. Bakkers (1997), Communicating Java Threads, in *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, IOS Press, Netherlands, University of Twente, Netherlands, volume 50, pp. 48–76.
http://www.ce.utwente.nl/rtweb/publications/1999/pdf-files/014_R99.pdf

Hilderink, G. H. (2005), *Managing Complexity of Control Software through Concurrency*, Phd thesis, University of Twente, Netherlands.

Hilderink, G. H., A. W. P. Bakkers and J. F. Broenink (2000), A Distributed Real-Time Java System Based on CSP, in *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE Computer Society, Washington, DC, USA, p. 400, ISBN 0-7695-0607-0.

Hoare, C. (1985), *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, Prentice Hall.
`http://www.usingcsp.com/cspbook.pdf`

INMOS (1988), *occam 2 Reference Manual*, Prentice Hall.

Istin, C. (2007), Extend the link between modeling and formal verification for control systems, MSc report 020CE2007, Control Laboratory, University of Twente.

Jovanovic, D. S., B. Orlic, G. K. Liet and J. F. Broenink (2004), gCSP: A Graphical Tool for Designing CSP systems, in *Communicating Process Architectures 2004*, Eds. I. East, J. Martin, P. H. Welch, D. Duce and M. Green, IOS press, Oxford, UK, pp. 233–251.
`http://doc.utwente.nl/49238/1/jovanovic04gcsp.pdf`

Liedtke, J. (1993), Improving IPC by Kernel Design, in *Proceedings of the 14th Symposium on Operating System Principles (SOSP-14)*, Asheville, NC.
`http://l4ka.org/publications/`

Liedtke, J. (1995), On micro-kernel construction, in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, ACM, New York, NY, USA, pp. 237–250, ISBN 0-89791-715-4, doi:http://doi.acm.org/10.1145/224056.224075.

Lootsma, M. (2008), Design of the global software structure and controller framework for the 3TU soccer robot, Msc Thesis 014CE2008, Control Laboratory, University of Twente.

Maljaars, P. (2006), Control of the Production Cell Setup, MSc Thesis 039CE2006, Control Laboratory, University of Twente.

Marwedel, P. (2006), *Embedded system design*, Springer.

Molanus, J. (2008), Feasibility analysis of QNX Neutrino for CSP based Embedded Control Systems, MSc 032CE2008, Control Laboratory, University of Twente.

Moores, J. (1999), CCSP – a Portable CSP-based Run-time System Supporting C and occam, in *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, Ed. B.M.Cook, WoTUG, IOS Press, Amsterdam, the Netherlands, volume 57 of *Concurrent Systems Engineering series*, pp. 147–168, ISBN 90 5199 480 X.
`http://www.cs.kent.ac.uk/pubs/1999/753`

Orlic, B. (2007), *SystemCSP a graphical language for designing concurrent, component-based embedded control systems*, Phd-thesis, Control Laboratory, University of Twente.

Orlic, B. and J. F. Broenink (2003), Real-time and fault tolerance in distributed control software, in *Communicating Process Architectures 2003*, Eds. J. F. Broenink and G. H. Hilderink, IOS Press, Enschede, Netherlands, pp. 235–250.

Orlic, B. and J. F. Broenink (2004), Redesign of the C++ Communicating Threads Library for Embedded Control Systems, in *5th PROGRESS Symposium on Embedded Systems*, Ed. F. Karelse, STW, Nieuwegein, NL, pp. 141–156.

Posthumus, R. (2007), Data logging and Monitoring for real-time systems, MSc report 015CE2007, Control Laboratory, University of Twente.

QNX Software Systems (2008), QNX.
`http://www.qnx.com`

Rashid, R., D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub and M. Jones (1989), Mach: A

System Software kernel, in *Proceedings of the 34th Computer Society International Conference COMPCON 89*.

Roscoe, A. W., C. A. R. Hoare and R. Bird (1997), *The Theory and Practice of Concurrency*, Prentice Hall PTR, Upper Saddle River, NJ, USA, ISBN 0136744095.

Schneider, S. (1999), *Concurrent and Real Time Systems: The CSP Approach*, John Wiley & Sons, Inc., New York, NY, USA, ISBN 0471623733.

Silberschatz, A., P. B. Galvin and G. Gagne (2004), *Operating System Concepts*, John Wiley & Sons, ISBN 0471694665.

van der Steen, T. T. J., M. A. Groothuis and J. F. Broenink (2008), Designing Animation Facilities for gCSP, in *Communication Process Architectures 2008, York, United Kingdom*, volume 66 of *Concurrent Systems Engineering Series*, Elsevier, Amsterdam, volume 66 of *Concurrent Systems Engineering Series*, p. 447, ISSN 1383-7575.

Technical University of Dresden (2008), TUDOS.
`http://tudos.org`

Welch, P. (2002), Process Oriented Design for Java: Concurrency for All, in *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, Eds. P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra and A.G.Hoekstra, Springer-Verlag, volume 2330 of *Lecture Notes in Computer Science*, pp. 687–687, ISBN 3-540-43593-X, ISSN 0302-9743, keynote Tutorial.
`http://www.cs.kent.ac.uk/pubs/2002/1383`

Xenomai (2008), Xenomai: Real-Time Framework for Linux.
`http://www.xenomai.org`