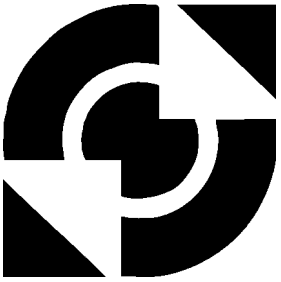
University of Twente

EEMCS / Electrical Engineering
Control Engineering



CTC++ enhancements towards fault tolerance and RTAI

Thiemo van Engelen

M.Sc. Thesis

Supervisors
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
M.Sc. D. Jovanovic

August 2004

Report nr. 022CE2004
Control Engineering
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

This report describes two enhancements to the CT library that has been developed at the Control Engineering department of the University of Twente.

As control systems are inherently concurrent, the Control Engineering department does research on the design trajectory of concurrent controllers. In the past, transputers were used together with the CSP based programming language Occam. CSP is a process algebra for describing and analyzing concurrent systems. By nature, it encourages partitioning and allows one to reason about and to verify a design at an early stage of the design process. This provides a large advantage in the design trajectory of concurrent controllers.

As transputers became obsolete, research shifted to a new CSP based design methodology. The CT library, GML models and the gCSP program allow users to design and implement a CSP based design in an easy way. GML models are a graphical representation of a CSP design. These can be drawn using the gCSP tool, which can generate code from the models that uses the CT library.

For any embedded system, safety is an important issue. Safety can be provided by fault-tolerance. This means that the occurrence of a fault does not have to lead to a failure of the complete system. Fault-tolerance can be provided both in hardware and in software.

The main tool for fault-tolerance in software is an exception handling mechanism. This provides the possibility to have a piece of code guarded by an exception handler. If an exception is raised, the exception handler is executed. Because no language for which the CT library has been written provides an adequate exception handling mechanism for concurrent programs, the CT library has been extended with an exception handling mechanism. The mechanism closely resembles the try/throw/catch mechanism of C++ and Java.

The added exception handling mechanism and its usage is explained in a number of examples. Some of the examples use the JIWY setup. For these examples, extra hardware was needed to have interrupt-driven, digital inputs.

The implemented behavior of the parallel construct could lead to deadlock. This can be solved by using channel rejection, which is also explained in a number of examples. Because of the hazard of deadlock in the current parallel implementation, it is recommended that a different behavior is researched.

The programs compiled with the CT library need an operating system be able to run on a PC platform. Until now, the usable operating systems were DOS (Windows) and Linux. The second enhancement to the CT library that this report describes is the possibility run programs compiled with the CT library under RTAI, a real time Linux variant. The changes needed to the library are described. The implementation of these changes was straightforward.

Issues that have to be taken care of when writing programs that use the CTC++ library under RTAI are the inability to use dynamic libraries and the inability to include Linux kernel header files. These could be solved by making it possible for CTC++ programs to run under LXRT, which is recommended for further research. Furthermore, one has to take care that the CT processes at one time all stop or block on a channel, because otherwise, the Linux kernel will not run and Linux will no longer be responsive.

As an example, a CTC++ program that controls the JIWY setup running under DOS has been ported to a version that runs under RTAI.

Samenvatting

Dit verslag beschrijft twee toevoegingen aan de CT bibliotheek, welke ontwikkeld is bij de Control Engineering vakgroep van de Universiteit Twente.

Omdat regelsystemen van nature parallel zijn, doet the Control Engineering vakgroep onderzoek naar een ontwerp traject voor parallelle regelaars. In het verleden werden transputers gebruikt, samen met de op CSP gebaseerde programmeer taal Occam. CSP is een proces algebra waarmee parallelle systemen beschreven en geanalyseerd kunnen worden. Van nature moedigt het partitioneren aan en het maakt het mogelijk in een vroeg stadium over een ontwerp te redeneren en het te controleren. Dit levert een groot voordeel op in het ontwerp traject van parallelle regelaars.

Omdat transputers niet meer verkocht werden, verschoof het onderzoek naar nieuwe CSP gebaseerde ontwerp methodieken. The CT bibliotheek, GML modellen en de gCSP tool bieden gebruikers de mogelijkheid om een CSP gebaseerd ontwerp te maken en te implementeren. GML modellen zijn een grafische notatie van een CSP ontwerp. Deze kunnen getekend worden met de gCSP tool. De gCSP tool kan code genereren die gebruikt maakt van de CT bibliotheek.

Voor elk embedded systeem is veiligheid erg belangrijk. Veiligheid kan bereikt worden door fouttolerantie. Dit houdt in dat het optreden van een fout niet leidt tot fout gedrag door het hele systeem. Fouttolerantie kan zowel door zowel hardware als software bereikt worden.

Het belangrijkste gereedschap voor fouttolerantie in software is een *exception handling* mechanisme. Dit biedt de mogelijkheid om een gedeelte van de code beschermd te hebben door een exception handler. Als een exceptie optreedt in de beschermde code wordt de exception handler uitgevoerd. Omdat de talen waarvoor de CT bibliotheek geschreven zelf onvoldoende mogelijkheid bieden voor exceptie afhandeling is er een exception handling mechanisme toegevoegd aan de CT bibliotheek. Dit mechanisme lijkt sterk op het `try/throw/catch` mechanisme van C++ en Java.

Het exception handling mechanisme dat is toegevoegd en het gebruik ervan wordt uitgelegd met behulp van een aantal voorbeelden. In een aantal van deze voorbeelden wordt de JIWY setup gebruikt. Voor deze voorbeelden was extra hardware nodig zodat er digitale ingangen met interrupt mogelijkheid beschikbaar waren.

Het exception handling mechanisme dat geï mplementeerd is kan in een parallel resulteren in *deadlock*. Dit kan voorkomen worden door het gebruik van *channel rejection*. Het gebruik hiervan wordt uitgelegd in een aantal voorbeelden. Omdat het gedrag van de *Parallel* construct kan leiden tot deadlock, is een aanbeveling om ander mogelijk gedrag van de *Parallel* construct te onderzoeken.

Programma’s die gebruik maken van de CT bibliotheek hebben een besturingssysteem nodig om te kunnen draaien op een PC. Tot nu toe waren alleen DOS (Windows) en Linux te gebruiken als besturingssysteem. De tweede toevoeging aan de CT bibliotheek die beschreven wordt in dit verslag is de mogelijkheid om programma’s die gebruik maken van de CT bibliotheek te kunnen draaien onder RTAI, een real time Linux variant. De aanpassingen die nodig waren aan de bibliotheek worden beschreven. Het implementeren van deze aanpassing ging rechttoe rechtaan.

De problemen waar men mee te maken krijgt als men een programma gaat schrijven die RTAI gaat draaien zijn het niet kunnen gebruiken van dynamische bibliotheken en het niet kunnen *includen* van *kernel header files*. Dit zou geen probleem meer zijn als CTC++ programma’s onder LXRT kunnen draaien. Ook moet men uitkijken dat de CT porcessen op een gegeven ogenblik allemaal geblokkeerd zijn of beëindigd zijn, omdat anders de Linux kernel geen tijd krijgt en Linux dus niet meer reactief is.

Om als voorbeeld te dienen is een CTC++ programma voor DOS dat de JIWY setup bestuurd omgezet naar een versie die onder RTAI draait.

Preface

Now that my report is finished, I would like to thank everybody who helped in this assignment. Some people deserve special mentioning:

First, I would like to thank Gerald Hilderink for the numerous discussions about CT. From this, the idea of this assignment has risen. More thanks for the guidance during the start of this assignment.

Then I would like to thank Dusko Jovanovic. He invested a lot of time towards the end of the project, which made it possible for me to finish this assignment before September. Furthermore, his input to this project greatly increases the completeness of my work.

Next I would like to thank Jurrie Kraai for his feedback on my report.

It goes without saying that I want to thank my parents for their support during all my years of studying.

Finally a great deal of thanks goes to my girlfriend Marianne Kraai, not the least for being there for me these years.

Thiemo van Engelen

Zwolle, August 2004

Contents

1 Introduction

1.1 Introduction

The focus of the Control Engineering department of the University of Twente lies in the field of robotic/mechatronic applications. The focus is both on controller design and on design methodology.

Real-time

A property of the studied control systems is that they all are hard real time systems. This means that actions have to be performed before a certain deadline, otherwise failure will occur. If a controller is required to be hard real time, it requires that all the services it uses are hard real time. This means that for a controlling hard real time program, not only the program needs to be programmed for this demand, but also the operating system (OS) it runs in and its communication with the outside world needs to be hard real time.

Concurrency

Because it was recognized that a control task is inherently concurrent, the focus has long been on concurrent controller design.

The CSP (Communicating Sequential Processes) language is a process algebra for describing and analyzing concurrent systems (Hoare, 1985). A CSP based design methodology encourages partitioning of the design and allows for good reasoning about the design at every level.

A CSP based design trajectory is described in (Hilderink, Gerald H. *et al.*, 2003). It uses the CT library that has been developed at the Control Engineering department (Hilderink, G.H. *et al.*, 1997). Using this library, CSP based designs can be implemented easily, because the CSP concepts can be directly translated into the programming language. Section 1.3 will give a short description of the CSP library.

Safety

Safety is an important issue in embedded systems. Think about large industrial applications or embedded systems in transportation usage. When they fail, they can cause a lot of damage or human injury. To ensure fail-safe operation in hardware, redundancy is an often-used method.

In software, an important tool to ensure a fail-safe device is to use proper design techniques. If the design can also be checked before it is implemented, a large number of errors can be removed on forehand. A CSP based design offers these advantages, because it can be checked by a model checker, for example FDR (FormalSystems, 2004).

When using C++, the code itself cannot be checked, unless perhaps only a safe subset of C++ is used and in accordance to strict rules. If however the program is based on CSP concepts, it can be proven that the design is correct. Because by nature the CSP design is already partitioned, only the implementations of the separate parts have to be checked for their correct behavior.

Fault tolerance

When a system is designed, a certain behavior of its environment and of the system itself is assumed. The actual behavior can however differ with the possible result that a part of the system fails. But for a system to be safe, not all parts of the system have to be error free. If the behavior of a system continues to be correct, even when a part of the systems fails, the system is said to be fault tolerant. It can tolerate failure of a certain part of the system, because it is known how to handle and cope with the failure. This way, the safety of the device can be increased. An example of fault tolerance is redundancy in hardware: failure of one piece of hardware is tolerated, because another piece takes over.

An important tool to make software fault tolerant is an exception handling mechanism. An exception handling mechanism offers the possibility to have parts of a program guarded by an exception handler. The exception handler can handle errors that occur in the guarded part of code, hereby providing fault tolerance to these errors.

1.2 Objectives

1.2.1 RTAI CT extension

On a PC architecture, programs compiled with the CT library still run under a certain OS. Until now, this OS is either DOS or Linux. While Linux does provide a very useful development environment, real time behavior is not guaranteed. Under DOS, programs can execute in a real time environment, but there is no support for multithreading, severely limiting its use as a development environment.

One objective of this assignment is to make it possible to run CT programs under RTAI, a real time Linux variant (RTAI, 2004). This way, a CT program can be run in a multithreaded, real time environment. Because it is multithreaded, the user can run the CT program and do other things beside it, for example monitor the system.

1.2.2 Exception handling CT extension

The second objective for this research is to add an exception handling mechanism to the CT library. Although an exception handling mechanism is often provided by the programming language, for CT based designs, the programming language mechanisms are not enough for proper exception handling. The reason being that they lack proper semantics when used in a concurrent environment.

By including an exception handling mechanism into the library, exception handling becomes an integral part of the software development process and can be properly incorporated into the design.

1.3 The CT library and GML models

To provide a graphical view on a CSP based design, a modeling language was developed called GML (Hilderink, Gerald H., 2002). A program called gCSP has been written to make the models on a computer and have code generated from the model (Jovanovic, Dusko S. *et al.*, 2004). In this section, elements of the CT library will be explained, together with their GML models.

1.3.1 Processes

A process is an independent entity, whose purpose it is to perform a certain task within its private workspace. During its progress, it may interact with its environment by means of channel communication. Processes do not know each other's name and cannot directly alter each other's state. A process itself can be a collection of multiple other processes.

1.3.2 Constructs

Constructs are processes that regulate the execution of processes they contain. Because constructs are also processes, they can also be used in other constructs. Within the CT library, the following constructs are available:

- (Pri)Parallel
- (Pri)Alternative
- Sequential

The behavior of these constructs will now be discussed.

(Pri)Parallel

The *(Pri)Parallel* construct causes the processes it contains to run in parallel. The P*arallel* construct has no preference over which process is allowed to run. The *PriParalle*l gives a priority to its processes. The *(Pri)Parallel* construct terminates when all the processes it contains have terminated.

Figure 1 shows on the left-hand side a *Parallel* construct containing two processes, with both being considered of equal priority. On the right-hand side, it shows a *PriParallel* construct containing two processes. In this figure, a higher priority is given to PROCESS1.

Figure 1. Two processes in a *Parallel* construct (left) and two processes in a *PriParallel* construct, giving priority to PROCESS1 (right)

(Pri)Alternative

The *(Pri)Alternative* construct does not contain processes, but contains *Guards*. A *Guard* however has a process associated with it.

The *Alternative* construct make a choice about which process to run, based on information provided by its *Guards*. They guard a channel they are associated with, and if a communication attempt has been observed on that channel, the *Guard* signals this to the *Alternative* construct. This can then decide to run the process associated to the *Guard*, which would perform communication over the channel. The *(Pri)Alternative* terminates when a *Guard* has been chosen and the associated process has been executed.

The *Alternative* construct gives no priority to the guards. The *PriAlternative* prioritises the guards. When making a decision on which process to run, the one with the highest priority is chosen if multiple *Guards* signalled a communication attempt.

Figure 2 shows on the left-hand side guards in an *Alternative* construct. Both guards are *input guards*, so they signal the *Alternative* construct when another process tries to write to the channel. The right-hand side of Figure 2 shows two guards in a *PriAlternative* construct. If a write is attempted in both channels, the *PriAlternative* construct will execute PROCESS1.

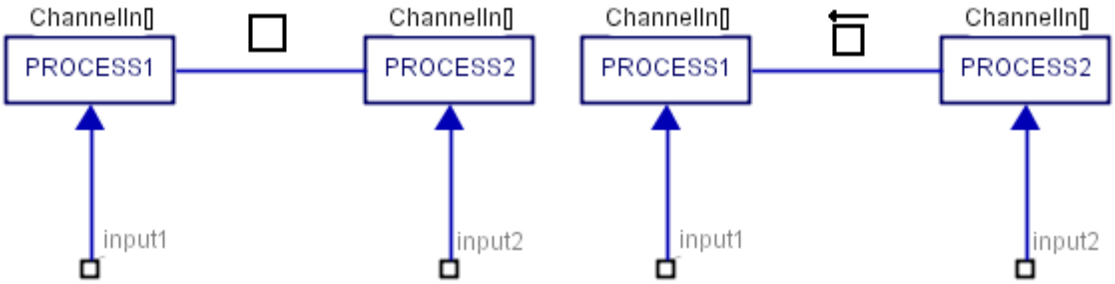


Figure 2. Two processes in an Alternative construct (left) and two processes in a PriAlternative construct (right), favoring PROCESS1 over PROCESS2

Sequential

The processes that are contained by a *Sequential* construct are executed in order. Figure 3 shows a *Sequential* construct containing two processes. The sequential relation prescribes that PROCESS1 should be run first. The *Sequential* construct terminates when all the process that it contains have terminated.

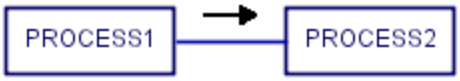


Figure 3. Two processes in a *Sequential* construct, executing PROCESS1 before PROCESS2

By connecting a μ-process to a *Sequential* construct, repetition of a process can be graphically shown in the model. Figure 4 shows an example. In this example, PROCESS1 is executed first. When

PROCESS1 terminates, the guard of the μ-process is evaluated. As long as the guard is true, PROCESS1 will be repeated.

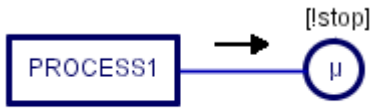


Figure 4. A *Sequential* construct with a μ-process

1.3.3 Parenthesizing in GML

A GML model allows processes to be grouped. This is shown by means of an open dot with an index. This parenthesis symbol shows the boundary of a group. When a number of processes are grouped into one process, the index of the parenthesis symbol on a construct that does not belong to the group is increased by one. Processes that are connected via a construct without parenthesis symbols belong to one group.

Figure 5 shows a GML model with parenthesizing. Which processes can be grouped according to the indices is shown according to the thick box and the dashed box. One can see that the number of times a construct is crossed equals the index of the parenthesis symbol on that side of the construct.

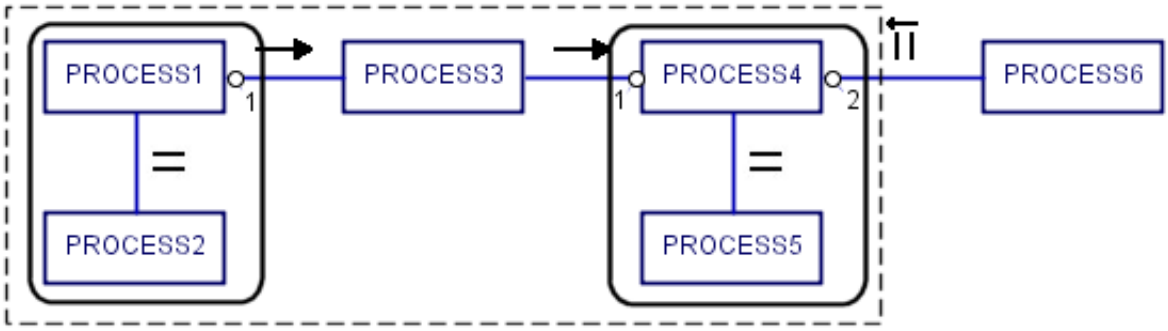


Figure 5. GML model with parenthesizing

Between process 1 and 4 and process 2 and 5 are no parenthesis symbols, so 1 and 4 can be grouped and 2 and 5 can be grouped. This is shown using the thick rounded box. When these are grouped, the indices at process 1 and process 2 are decreased by one (See Figure 6).

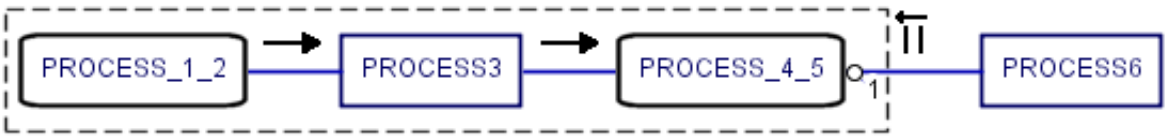


Figure 6. GML model with the processes 1 and 4 and 2 and 5 grouped

Now there are no parenthesis symbols between 1_4, 3 and 2_5, so these can be grouped. This is shown using the dashed box. The index on process 2_5 is decreased by one (See Figure 7). Now there are no parenthesis symbols left in the model.

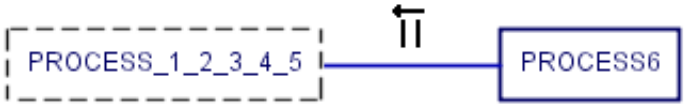


Figure 7. GML model with the processes 1, 4, 3, 2 and 5 grouped

1.3.4 Channels

A channel is an object that performs communication between processes or between a process and the environment. Channels provide synchronization, scheduling and the actual data transfer. Each channel supports one-way point-to-point synchronous communication. The notion of a channel itself abstracts away from the actual physical implementation of the communication link.

Channels that are used for communication within the program are called *internal channels*, while channels that are used to communicate with the environment are called *external channels*. The part within an external channel that provides the actual communication with the environment is called the *linkdriver*. The left-hand side of Figure 8 shows a GML model of two processes communicating over an internal channel and the right-hand side shows a GML model of process that has an external channel as input.

Internal channels in CT by default provide rendezvous communication, meaning that both a reader and a writer need to participate in the communication. When the first one wants to communicate, it is suspended until the other one also wants to communicate. At this moment, data transfer takes place and they both can resume execution.

Rendezvous channel communication can only take place between processes that run in parallel. The reason is that when they do not run in parallel, they can never participate in the communication at the same time. Therefore, if communication has to take place between sequential or alternative processes, a non-rendezvous channel has to be used.

Processes can communicate through channels by calling their read and write method. For communication, this is the only interface to the process. In (Hilderink, Gerald H. and Broenink, 2003) and (Brown and Welch, 2003), a way to change the state of the channel has been proposed. In this state, the channel throws an exception when an action is performed on the channel. In this report, placing the channel in this state is called channel rejecting, because after the change, the channel rejects all actions.

Figure 8. Two processes communicating over a channel (left) and a process that communicates through an external channel (right)

1.4 Outline report

As said, this research has two objectives, both aimed at extending the functionality of the CT library. The first goal is to make it possible for CT programs to run under RTAI. Results of this goal will be discussed in chapter 2. The second objective of this research is to add an exception handling mechanism to the CT library. The results of this can be found in chapter 3. In chapter 4 examples of how the exception handling mechanism in CT can be used are given. In chapter 5, conclusions that are drawn from this research and recommendations for future research are given.

2 CTC++ under RTAI

2.1 Outline

This chapter discusses the changes needed to CTC++ to make it run under RTAI.

First, section 2.2 will give a small introduction into Linux and mentions what kernel modules are. Section 2.3 gives an introduction into what RTAI is and how it fits into Linux. It explains which kernel modules of RTAI are important for CTC++ under RTAI. Problems that are encountered when writing kernel modules in C++ will be discussed in section 2.4.

In section 2.5 the changes to CTC++ that were needed to run it under RTAI are mentioned. It also gives points to consider when writing programs that use CTC++ for RTAI.

As a demonstrator a CTC++ for DOS program that controlled a mechatronic setup called JIWY was converted to a CTC++ for RTAI program. Section 2.5.5 mentions the changes to the program as an example of how to convert a program written for the DOS version of CTC++ to a version for RTAI.

2.2 Linux

Linux is an OS, based on UNIX, originally designed by Linus Torvalds (LinuxOnline, 1994). It has a modular structure in which a basic kernel is loaded at boot time and modules for certain hardware or software support is later on loaded into the kernel on demand. These modules are called kernel modules. When the processor is running code inside the kernel, the code can execute any command and directly call other functions that are inside the kernel.

Because of its good network support and its stability, it is used for servers around the world. This means that there is a large user base of Linux users and, consequently, a lot of expertise is available. This expertise is shared is on the Internet. There are many forums on which questions can be asked and problems be posted.

The Linux kernel and many Linux applications are distributed under the GNU/GPL license (Foundation, 1989). This means they can be copied freely. This makes it interesting for the industry because there are not any license costs involved.

Another advantage of the GNU/GPL license system is that the source should be freely available to the user. This means that when a bug is suspected in the kernel or in a program, the users can search for the error them selves and fix it.

The main disadvantage of Linux is the lack of proper documentation. Many things are at first written for personal use only, and documentation is then the last point of focus.

2.3 RTAI

The standard Linux kernel is does not allow hard real time behavior. This can be solved by a set of kernel modules and a patch of the Linux kernel. One of these solutions is called RTAI, which has been developed at the Department of Aerospace Engineering at the university of Milan (RTAI, 2004).

The core of RTAI consists of three parts:

- A patch to the Linux kernel
- A real time interrupt handling module
- A scheduler module.

Figure 9 depicts the situation when the Linux kernel is patched and the RTAI interrupt handler module and the RTAI scheduler module are loaded. It also shows where the CTC++ programs will be placed.

Patch

The patch is a text document containing changes that have to be applied to the source code of the Linux kernel. When the changes are made, the Linux kernel has to be recompiled and reinstalled.

Interrupt handling module

Using the patch alone doesn't change the functioning of Linux. The changes to the kernel take effect when the main RTAI module (rtai.o) is loaded. From this point on, all interrupts are first handled by RTAI interrupt-handling code in this module and are then forwarded to the Linux kernel. Other kernel modules can install a real time interrupt handler by registering them to the RTAI interrupt-handling module. If the associated interrupt occurs, the registered interrupt handler is called by the RTAI code. If a real time interrupt handler has been registered, then the associated interrupt will no longer be automatically forwarded to Linux. This will only be done if the installed interrupt handler tells RTAI to do so. More on interrupts can be found in section 2.5.4.

Scheduler

Next to the RTAI interrupt handling module, RTAI also consists of a scheduler module. This module implements a priority-based scheduler. When this module is loaded, it continues to run the Linux kernel as the lowest priority task (from now on called the *Linux task*). If there are no tasks running besides the Linux task, the Linux task gets all the CPU time. When a kernel module starts a new RTAI

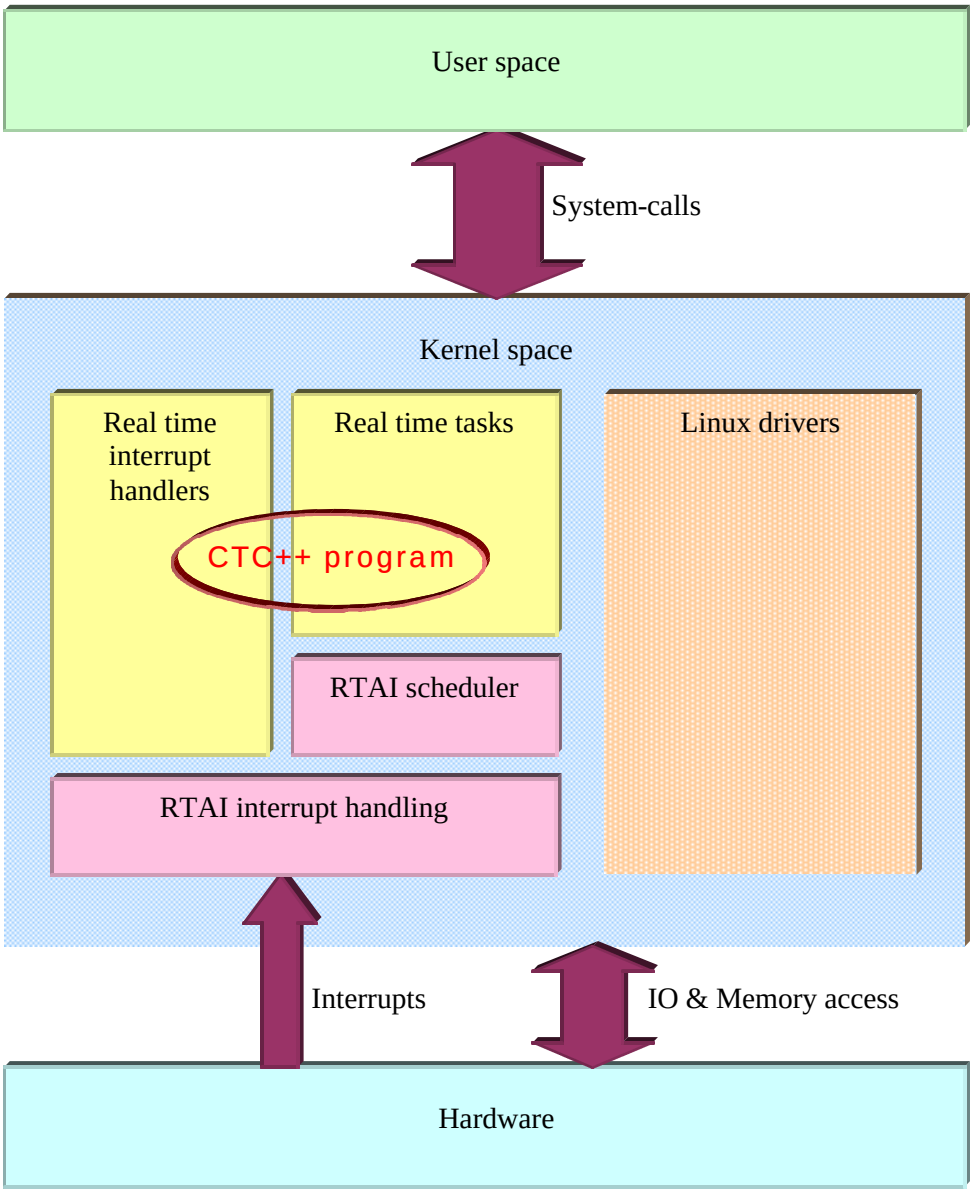


Figure 9. Structure of Linux and RTAI

task, the Linux task is scheduled out by the RTAI scheduler and the new task starts to execute. The Linux task does not get any CPU time, until the new task is either blocked or terminated. This means that when a RTAI task never ends and never blocks, the Linux task gets no more CPU time, which means the system is does no longer respond to input from the user which is handled by the Linux kernel (and thus the Linux task).

2.4 Kernel modules and C(++)

There are certain differences between writing kernel modules and normal programs, especially when using C++. These differences will be discussed in this section.

When writing kernel modules, not all normal C functions like `printf` or `malloc` are available. There are two reasons for this. The first is that these functions are in a shared library that is dynamically loaded when a program is run. The loading of the library is done before calling the `main` function by a piece of code that is automatically linked to the program by the linker. When a kernel module is compiled, this part of code is not linked to the kernel module. Therefore, the library is not loaded when a module is loaded.

The second reason is that these functions assume that they are executed in user space. Most of them use a system-call to call functions inside the kernel. Because it is not allowed do make system-calls inside the kernel, this code should not be used by a kernel module.

Because of the two reasons that are mentioned, many normal C functions cannot be used in kernel modules. For some of them there are usable replacements, which will be mentioned in section 2.5.3.

The problem with dynamic libraries applies to C and to C++ programs. For C++ programs the problem is more extensive because commands like `new, delete, try, catch and throw` use functions in dynamic libraries and are thus normally not available in the kernel. For the C++ functions `new` and `delete` replacements are available in a RTAI kernel module called rtai_cpp.o. This module must be loaded before a C++ written kernel module can be loaded. Until now support for `try`, `catch` and `throw` is not provided by this module. The main reason for this is that the implementation and thus which functions are needed are for `try`, `catch` and `throw` are different between compiler versions.

For kernel modules written in C++ another problem exists. The Linux kernel is mostly written in C, some assembly code, but no C++ code. The effect of this is that not all header files in the Linux kernel directory are compatible with C++ and cannot be included in C++ code. A problem can for example be that a variable is called `new`, which is a reserved keyword in C++. This means that when a kernel module written in C++ wants to use variables or call functions in the kernel, a prototype is not available. This gives problems when compiling the C++ code. This problem can only be solved by copying the prototypes from the Linux kernel header files to the C++ code or a new header file that can be used in a C++ program.

2.5 Changes to CTC++

2.5.1 Wrapper for RTAI CTC++ programs

A normal program has a `main` function where execution will start. Linux kernel modules do not have a `main` function where execution is started. Kernel modules have an `init_module` function that is executed when the module is loaded, and a `cleanup_module` that is executed just before the module is unloaded. To make it easier to write CTC++ programs for RTAI, a wrapper was written, that contains these functions. When a program is compiled with the CTC++ for RTAI library this wrapper is automatically linked to the program. The following steps are performed when a program is loaded and, after running, unloaded from the kernel:

1. `init_modules` is called when a module is loaded into the kernel
2. The C++ support of RTAI is initialized
3. The constructors of all static object are invoked
4. The RTAI scheduler is told to start a new real time task
5. The RTAI scheduler starts the execution of the task at the function `main_c`
6. The C++ `main` function is called and the execution of the program starts

7. `cleanup _module` is called when a module is unloaded from the kernel
8. The RTAI C++ support is told to destruct the static variables
9. The destructors of all static objects are called

A RTAI CTC++ kernel module should not be removed from the kernel when the RTAI CTC++ task has not been terminated. This will result in memory leakage and possibly a kernel crash.

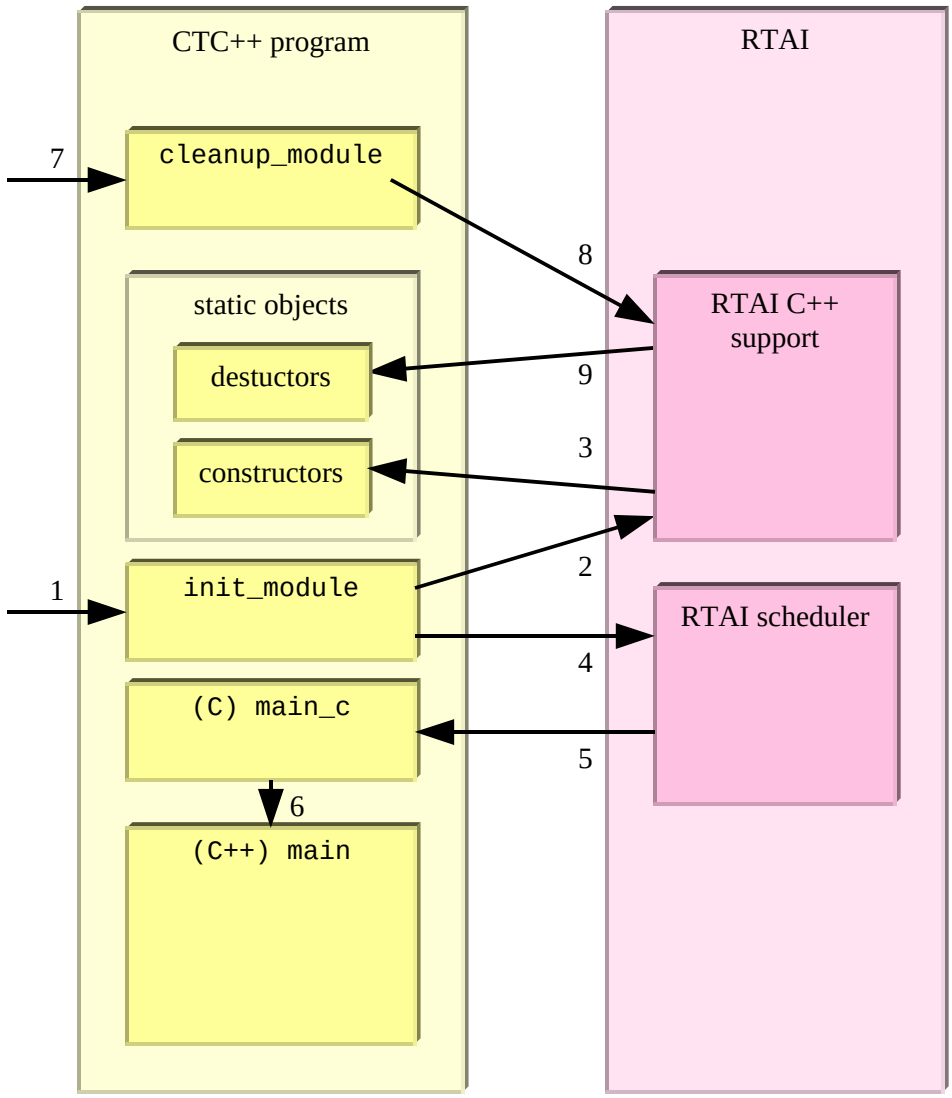


Figure 10. Sequence of the loading and unloading of a CTC++ for RTAI kernel module

2.5.2 Suspend & Resume

CTC++ contains an *Idletask*, which is normal process. An *Idletask* is always created by the scheduler. The *Idletask* runs when all other processes are either blocked or terminated. If all other processes are terminated, the *Idletask* also terminates and thus the CTC++ program terminates. If all processes are blocked, the DOS/Linux implementation of the *Idletask* goes into a busy loop. In this loop, it checks if it should terminate or schedule to a process that is no longer blocked.

If the DOS/Linux implementation of the *Idletask* would be used in the RTAI CTC++ implementation, the RTAI task where CTC++ runs in keeps running when the *Idletask* is active. This means that the RTAI scheduler would not schedule to the Linux task. This in turns means that the Linux kernel is no

longer responsive to input from the user. This is solved by letting the *Idletask* call a function, which is included in the RTAI CTC++ wrapper, which suspends the RTAI task CTC++ runs in.

If the RTAI task of the CTC++ program is suspended, it means all CTC++ processes are blocked on a channel. When a process becomes unblocked, the RTAI task of CTC++ should be resumed. The only processes that can become unblocked are processes that are blocked on an external channel or on a timed channel. For a timed channel, the unblocking is done inside the Timer interrupt handler. For external channels, the unblocking can be done inside an interrupt routine or inside a function that is called by the Linux kernel. The moment of resumption of the RTAI task is different between resuming in an interrupt handler and resuming in a function called from the Linux kernel.

If the unblocking is not done in an interrupt, like in a `FifoChannel` (see section 2.6), the RTAI task of CTC++ is immediately resumed. Because the task was suspended in the *Idletask*, it will also resume in the *Idletask*. Here the CTC++ scheduler schedules to the unblocked task.

Figure 11 shows a sequence diagram when the unblocking is done outside an interrupt routine. In this case, the following steps are performed:

1. *IdleTask* signals the wrapper to suspend the RTAI task it runs in
2. The wrapper signals RTAI to suspend the RTAI task of CTC++
3. A (callback) function in the CTC++ program is called by a Linux user space process
4. The callback function calls the unlock method in *linkdriver*, which signals the wrapper that it should resume the CTC++ RTAI task immediately
5. The wrapper signals to RTAI that the CTC++ RTAI task can be resumed.
6. Execution continues in the *IdleTask*, which then immediately yields, scheduling to the highest priority thread
7. The CTC++ program runs until all processes are blocked again. The *IdleTask* signals the wrapper to suspend the RTAI task it runs in
8. The wrapper signals RTAI to suspend the RTAI task of CTC++ and execution of the Linux

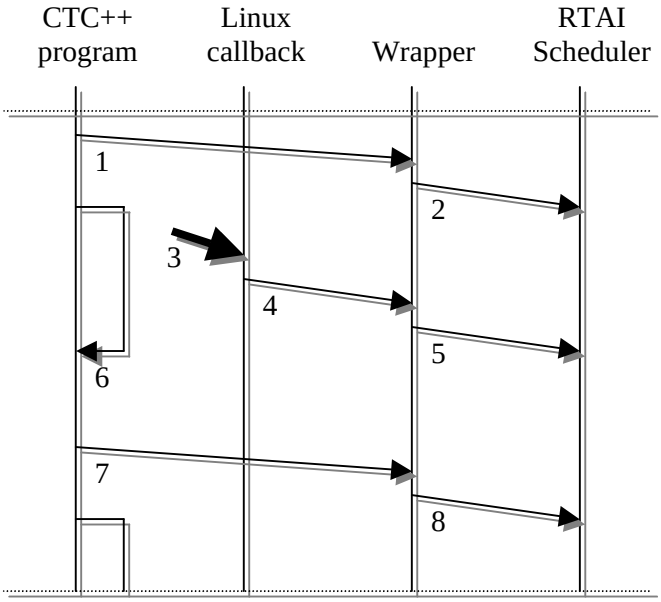


Figure 11. Sequence diagram for unblocking outside an interrupt routine

kernel continues

If the unblocking is done in an interrupt routine, the RTAI task of CTC++ should not be resumed until all interrupt routines for the interrupt are executed. This is because RTAI resumes a task immediately when the `resume` function is called. It does not wait for the interrupt routine to finish. This means that when the RTAI CTC++ task would be resumed on the first unblock action, the RTAI CTC++ is immediately scheduled in without finishing the interrupt routine. If the interrupt routine would also

unblock a higher priority task, this task would remain blocked until the interrupt routine is finished, which happens when the Linux task is scheduled in again.

The erroneous behavior described above is pictured in the top part of

Figure 12. The numbers correspond with the following actions:

1. *IdleTask* signals the wrapper to suspend the RTAI task it runs in
2. The wrapper signals RTAI to suspend the RTAI task of CTC++
3. An interrupt occurs and the interrupt manager is started
4. The interrupt manager calls 1st interrupt handler
5. Interrupt Handler1 calls `unlock` in *linkdriver* which signals the wrapper that it should resume the CTC++ RTAI
6. The wrapper signals to RTAI that the CTC++ RTAI task can be resumed.
7. Execution continues in the *IdleTask*, which then immediately yields, scheduling to the highest priority thread.
8. The CTC++ program runs until all processes are blocked again. The *IdleTask* signals the wrapper to suspend the RTAI task it runs in
9. The wrapper signals RTAI to suspend the RTAI task of CTC++
10. Execution continues in the interrupt manager that now calls Interrupt Handler2

One can see in the figure that the time between Interrupt Handler1 and Interrupt Handler 2 is as large as the time it takes for the CTC++ program to be blocked again. This problem is solved by checking in the unblock routine if it is called from an interrupt routine. If this is the case, it only sets a bit. After all interrupt handlers are executed, this bit is checked and if it is set, the RTAI CTC++ task is resumed. This then continues its execution inside the *Idletask*. Here the CTC++ scheduler schedules to the highest priority task that was unblocked. This (correct) behavior is shown in the bottom part of

Figure 12 where the following actions are performed:

1. *IdleTask* signals the wrapper to suspend the RTAI task it runs in
2. The wrapper signals RTAI to suspend the RTAI task of CTC++
3. An interrupt occurs and the interrupt manager is started
4. The interrupt manager calls the 1st interrupt handler
5. Interrupt Handler 1 calls `unlock` in *linkdriver*, which signals the wrapper that it should resume the CTC++ RTAI task once all interrupt handlers are done
6. The interrupt manager calls the 2nd interrupt handler
7. The interrupt manager signals to the wrapper that all interrupt handlers are done and that now the CTC++ RTAI task can be resumed
8. The wrapper signals to RTAI that the CTC++ RTAI task can be resumed.
9. Execution continues in the *IdleTask*, which then immediately yields, scheduling to the highest priority thread

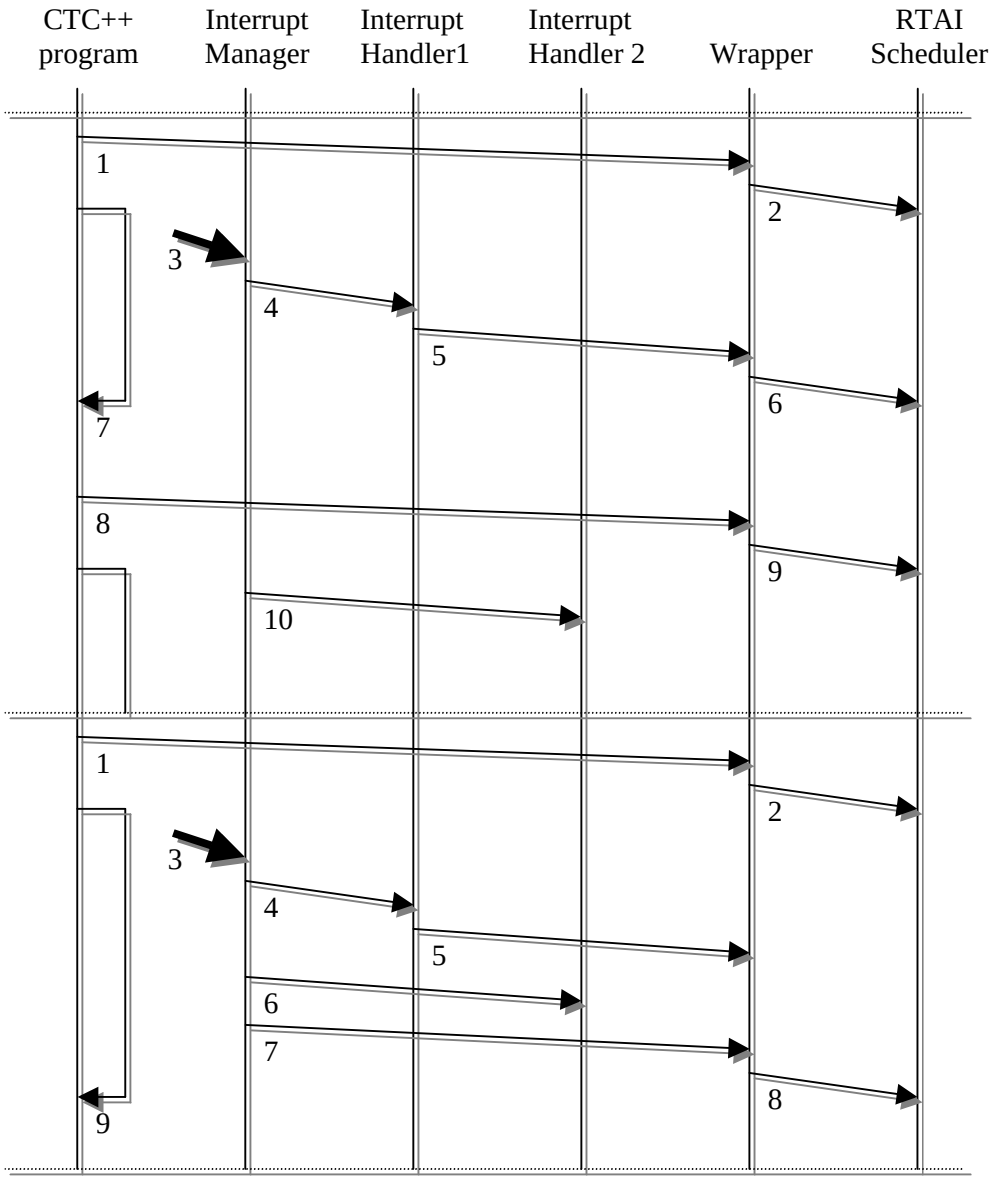


Figure 12. Sequence diagram for incorrect (top) and correct (bottom) behavior for unblocking from an interrupt routine

2.5.3 Real time safe functions

As mentioned in section 2.4, many functions are not available in Linux kernel space. There are functions inside the kernel that can be used as replacement in non real time kernel modules. Examples of this are `printk`, which is a replacement for `printf` and `kmalloc`, which is a replacement for `malloc`. However, these should not be used in real time programs, because they are not written for real time behavior. For some of these functions, there are real time replacements provided by RTAI. These will be discussed in this section.

Two replacements have to do with memory management. The first is `rt_malloc`. This function is a real time variant of the function `kmalloc`, which is a kernel replacement for `malloc`. Normally `kmalloc` has to do some memory management, which can slow down the request considerably. The RTAI memory management solves this by allocating a chunk of memory when the Linux kernel is running and dividing this up when `rt_malloc` calls are made. This way the memory allocation is really fast. When the total amount of allocated memory is above a certain threshold, the RTAI memory management module allocates a new chunk of memory. It does this in Linux kernel time, so it doesn't interfere with the real time tasks. This means that if the Linux task gets no processor time, the RTAI memory management module can run out of memory. The size of the chunk that is allocated can be changed inside the RTAI source code. The second memory related replacement is `rt_free`, which is

the real time equivalent of the C function `free`. It must be used to free memory that was allocated using `rt_malloc`.

As said in section 2.4, the `rtai_cpp.o` module provides `new` and `delete` functions for C++ kernel modules. In these functions the `rt_malloc` and `rt_free` functions are used, so `new` and `delete` are also real time.

Another real-time safe replacement provided by RTAI is `rt_printk`. It replaces `printk`, which is a kernel replacement for `printf`. A lot of functionality of `printf` is included `rt_printk`, with the exclusion of floating point printing. The text that is printed using `rt_printk` is first placed in a buffer and when the Linux kernel is scheduled in, it starts to print to the screen. The first consequence of this is that when the Linux kernel does not get any processor time, nothing will be printed to the screen. Secondly, the amount of text that can be printed while the CTC++ program is running is limited, because of the buffer that is used to store the text until the Linux kernel gets processor time. The size of this buffer can be adjusted in the RTAI source code.

2.5.4 Interrupt mechanism

Although CTC++ can be seen as an OS, because it has its own scheduler, for some architectures it still depends on an OS that is already running. One the functionality that is used from the OS is the interrupt mechanism.

If an interrupt handler is installed in a CTC++ DOS program, there is the possibility to chain the new interrupt handler. This means that an interrupt handler that is already installed by another program (or driver) will also be executed together with the CTC++ interrupt handler. The interrupt handling routine for CTC++ for RTAI also has a possibility to chain interrupts, but the meaning is different. In CTC++ for RTAI, it determines whether the received interrupt should also be forwarded to the Linux kernel. When an interrupt is forwarded, the Linux kernel executes its interrupt handler for that interrupt, when the Linux task is scheduled in.

Figure 13, Figure 14 and Figure 15 depict a timeline what the processor is executing when an interrupt occurs.

Figure 13 shows a timeline when an interrupt is only handled by Linux. When the interrupt occurs, the RTAI kernel disables the interrupt and resumes execution of real time tasks. When the real time tasks are suspended, the Linux task is scheduled in. This then starts to handle the interrupt after which the interrupt is re-enabled by the RTAI kernel. One can see that the minimum time between two interrupts is at least the time it takes for the RTAI task, which is here the CTC++ program, to get suspended or terminated. This should be no problem, because the interrupts are not used in the real time program.

Figure 14 shows the situation when an interrupt is only handled by CTC++ program. When an interrupt occurs, the RTAI kernel again disables the interrupt and immediately starts the execution of the real time interrupt handler. When the handler is finished, the interrupt is re-enabled by the RTAI kernel. The minimum time between two interrupts is the smallest in this case, because the interrupt is immediately enabled after the CTC++ interrupt handler.

Figure 15 shows the situation when an interrupt is shared between a CTC++ handled IO card and a Linux handled IO card. In this case, the CTC++ program should *chain* the interrupt. If an interrupt occurs, the RTAI kernel disables the interrupt and starts to execute the real time interrupt handler. When this is finished, the running real time tasks are resumed. When these are all finished or suspended, the Linux kernel is scheduled in, which starts to handle the interrupt. When the Linux interrupt handlers are finished, the RTAI kernel re-enables the interrupt.

There is a downside to chaining interrupts in CTC++ for RTAI. The interrupt is disabled until the Linux kernel has handled the interrupt. However the Linux kernel can only handle the interrupt when it is scheduled in, which happens when all RTAI tasks are ended or suspended. This means that a chained interrupt can only occur once between the running of the Linux task. Therefore it is strongly advised to avoid interrupt sharing between a CTC++ handled IO card and a Linux handled IO card.

When an interrupt is not shared between IO cards or only used by IO cards that are handled by a CTC++ for RTAI program, the interrupt doesn’t have to be chained, but can be claimed. This ensures

the maximum possible interrupt rate because interrupts are enabled as soon as all interrupt services are called.

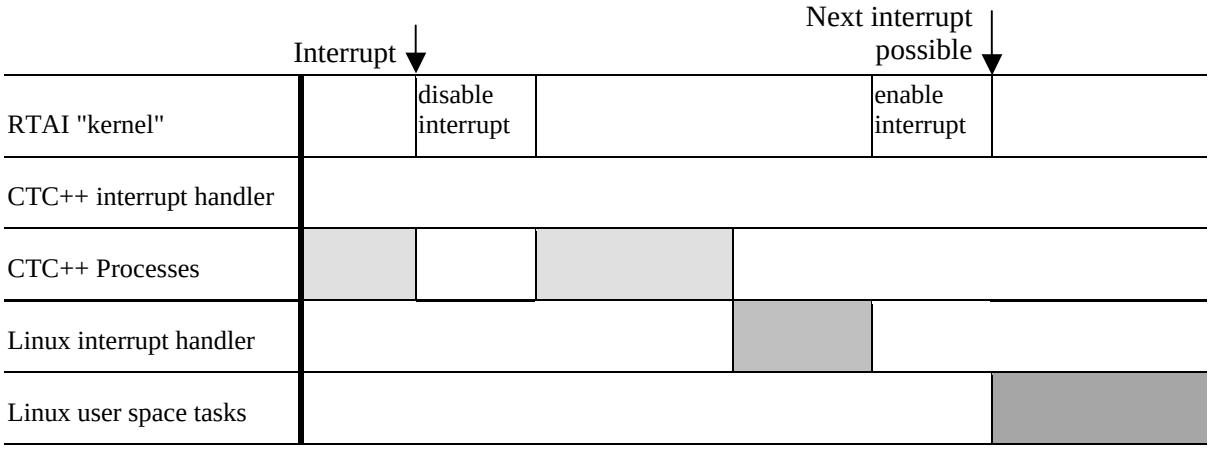


Figure 13. Timeline when an interrupt is only handled by Linux

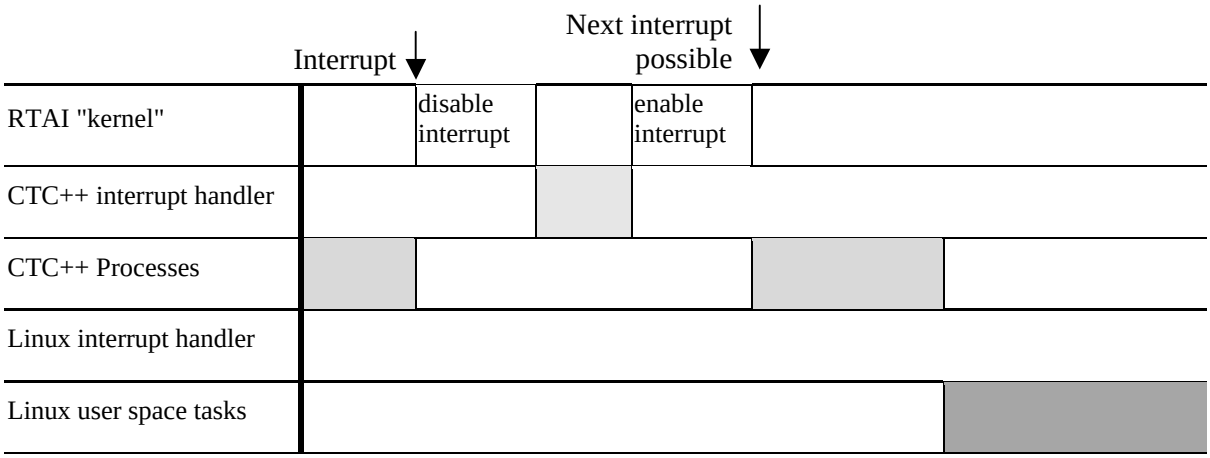


Figure 14. Timeline when an interrupt is only handled by CTC++

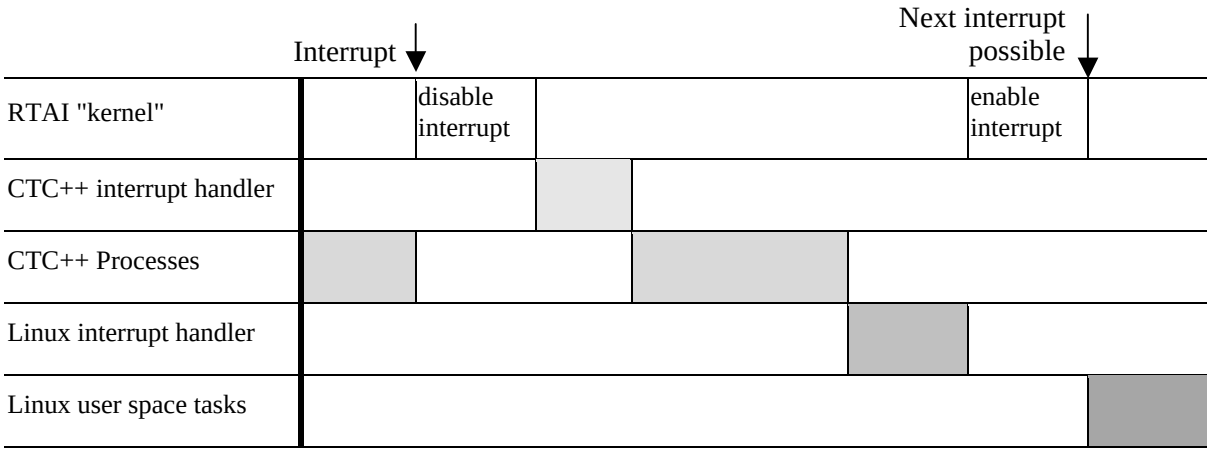


Figure 15. Timeline when an interrupt is handled by both CTC++ and Linux

2.5.5 Timer (updating Linux time)

As said in section 2.5.4 CTC++ runs on top of another OS for some architectures. Therefore, it needs to take care not to interfere with the OS that is already running. Because of this, an addition had to be made to the `Timer64` object in the CTC++ for RTAI library.

When Linux boots it sets up the PC's timer to generate timer interrupts at a rate of 100 Hz. Linux uses this, among other things, to allow round robin scheduling between tasks and to keep track of the system time. When the interrupt would no longer be received by Linux, the dynamic behavior of Linux changes drastically, because it can only schedule to another user space task in certain kernel functions, for example semaphore functions. Therefore, the RTAI for CTC++ library needs to forward the timer interrupts to the Linux kernel. The frequency of the forwarding should be close to 100 Hz.

To achieve this, the CTC++ for RTAI library creates a task when a new `Timer64` object is created. This task runs at a rate of 100 Hz the only thing it does is forward one timer interrupt to the Linux kernel every time it runs. When the `Timer64` object is destructed, it restores the Linux interrupt handler and the 100 Hz timing, so that everything is back to normal.

2.5.6 Implementation of the changes

The changes to the CTC++ library that are mentioned in the previous sections have been implemented. During implementation, no problems were encountered. The changes can mostly be found by searching for `#ifdef RTAI` or `#ifndef RTAI` commands in the source code of the library.

2.6 Porting example

As a CT demonstrator setup for the Control Engineering department the JIWY is often used. More on JIWY will be discussed in chapter 4. This chapter will discuss the porting of the existing CTC++ for DOS program to a CTC++ for RTAI program.

The largest change had to be made to the drivers for the NI6024E interface card that is being used. The problem was the (PCI) initialization part of the driver. The card responds to memory access at certain addresses. These addresses can not be directly accessed under DOS nor Linux. To solve this, the DOS driver remaps the card in such a way that the card responds to an address range that the DOS driver can access. It does so in a very low level way, by directly accessing registers on the card.

This method is not allowed under Linux, as Linux maintains all the PCI addresses. To be able to access the card the Linux kernel provides functions to reserve a piece of memory that, when accessed, actually points to the address the interface card listens to. The include files that contain the prototypes for these functions are however Linux kernel include files and thus cannot be used in the C++ driver for the IO card. This was solved by putting the PCI initialization in a separate C file, which can include the Linux kernel include files.

As already mentioned in section 2.4, there are functions that are not available in the kernel or for which no prototype is available. Some of these functions are also used in the JIWY program. This was solved by making custom header files, which redefine the functions that are not available to empty statements or to equivalents that can be used.

Another thing changed to the program was the addition of the possibility to use a `FifoChannel` instead of an `AnalogJoystick` channel. The `AnalogJoystick` channel returns the position of an analog joystick connected to the controlling computer. This position is used as the reference for the controllers of JIWY (See Figure 16).

A `FifoChannel` is a channel that uses RTAI fifo's to make communication possible between Linux user space and a RTAI CTC++ program that runs in the kernel. By using a `FifoChannel`, the JIWY setup can be controlled by a program running in Linux user space. This program can for example take data from a network connection and send it to the JIWY RTAI CTC++ software running in the kernel. This would allow JIWY to be controlled over the network (Ros, 2004) (See Figure 17).

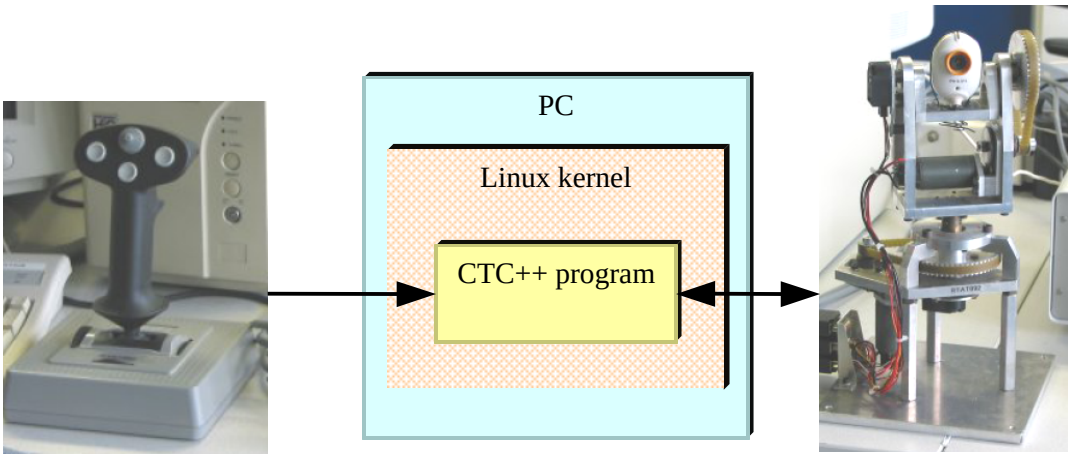


Figure 16. Diagram of JIWY controlled by a joystick

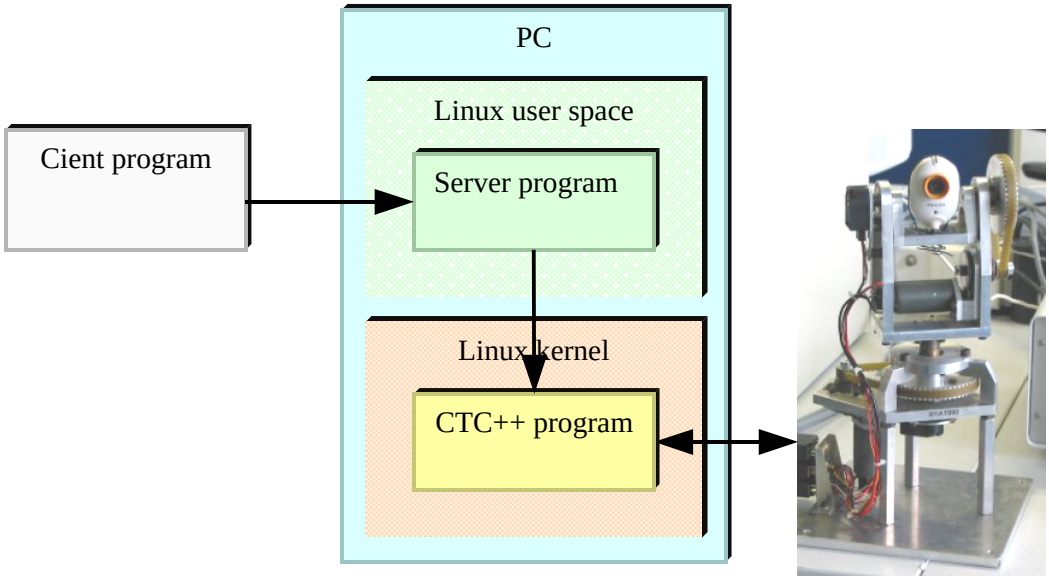


Figure 17. Diagram of JIWY controlled over a network connection

2.7 Summary

In this chapter, CTC++ for RTAI has been discussed. A small introduction into RTAI and kernel modules is given where the problems with writing kernel modules using C++ are discussed. These problems were also encountered when porting the JIWY program to CTC++ for RTAI.

There are two large problems when using CTC++ for RTAI programs:

- Not being able to use all standard functions, because they are in dynamic libraries.
- Not being able to include kernel header files in C++ files.

The necessary changes to the CTC++ library that were needed to make it work correctly under RTAI are discussed. Changes to the *Idletask*, to the unblocking of external channels, to the interrupt mechanism and to the `Timer64` object were needed to make CTC++ cooperate in the right way with RTAI and Linux. The implementation of the changes was straightforward.

As a porting example, an existing CTC++ program for DOS was ported to a RTAI version. As an extra option, the possibility was added to allow JIWY to communicate with Linux user space, allowing it to be controlled over a network connection.

3 Exception handling in CT

3.1 Outline

This chapter will discuss the exception handling mechanism that has been added to the CTC++ library. Section 3.2 will give a short discussion on what exceptions are, what the demands on an exception handling mechanism are and how exceptions are handled in programming languages. Section 3.3 will give a description of the conceptual design of the exception handling mechanism. This will not only be on the CTC++ library, but more on CT in general. How this concept was implemented in the CTC++ library is discussed in section 3.4. In section 3.5, an evaluation will give be given of the implementation and in conclusions will be presented in section 3.6.

Examples of using the implemented exception handling mechanism are the subject of chapter 4.

3.2 Exceptions and exception handling

3.2.1 Exceptions, raising and handling exceptions

An exception is an indication (signal) of the inability to perform a certain operation form the operational domain of a system (Cristian, 1995).

The notion of an exception is often coupled to occurrence of an error in the system design or its environment. Common examples in computing practice are:

- “Division by zero” (inability to perform a mathematical operation)
- “Memory page fault” (inability to perform a memory operation)
- “File does not exist” (inability to perform a file operation)

Exceptions can be signaled by hardware (“Memory page fault”) and by software (“File does not exist”). Division by zero can be signaled by both hardware and software. A program can check if the denominator is zero and if this is the case raise an exception or it can attempt the division in the hardware without checking the denominator’s value. In that case, the hardware may raise an exception when this value is zero. In section 4.4.2 disabling a cable on a robotic set-up represents an exception to normal environmental conditions.

Another way to look at exceptions builds upon the concept of *operational state space* of a system, what will be here referred to as *operating range* (of the values of the system’s variables). Every system (process, program, etc.) has a certain valid operating range within which it operates correctly. For a processor a certain range in ambient temperature is a component of its operating range. For (mathematical) division of real numbers, the valid operating range for denominator values spans all real numbers except zero. The operating range consists of both the input to the process and the internal state of the process. In the case study of chapter 4, the position of a motor axes (internal state of the motor) outside an allowed position range may cause overflow of underflow of the position encoder supplying the controller with feedback information (input to the computational process).

An exception can be seen as an indication of a violation of the valid operating range of a process. By observing the occurrence of an exception caused with exceeding the operation range in some of its subsystems (processes, components, etc.) a system effectively extends the operation range of its functionality, provided that the subsystem is able to encounter and signal the exception. For example, if a processor cannot sense the ambient temperature, it cannot raise an exception when it is out of bounds. This can lead to an error within the system and, in this case, damage to the processor because of a high temperature. This indicates two necessities for using exceptions: an exception occurrence should be observable and there must be a possibility to signal the exception. Beside these two necessities, there must be a possibility to react to the signaled exception.

Combining the two views, it can be said that an exception occurs when an operation cannot be performed because the system state violates the operational range of the system. Note that the operational range of the system can be time dependent, for example if an action has to be performed before a certain deadline.

Raising (or *throwing*) an exception by a process represent signaling of the inability to perform the operation. Receiving (*catching*) an exception and a proper *handling* is, in principle, responsibility of the same or another process, called *exception handler*. Catching an exception may not follow the throwing immediately – the mechanism of *propagating* an exception may get activated.

When a process is accompanied (guarded) by a separate exception handler, the combination of the two forms a new process. The valid operating range of this new process is larger then the valid operating range of the original (unguarded) process. This does not say anything about the correct behavior of the new process. Whether the behavior is correct or not is up to the system designer.

The functionality of original, unguarded process is called the *error-free operation* (Burns and Wellings, 2001). The functionality of the exception handler is called *exceptional operation*.

3.2.2 Demands on an exception handling mechanism

An exception handling mechanism is the complete chain from raising an exception, propagating and handling the exception. In the literature (Buhr and Mok, 2000; Burns and Wellings, 2001), several general demands are imposed on exception handling mechanisms.

1. The mechanism should be simple to understand and use.
   If this rule is violated, the exception handling mechanism will not be used or not be used to its full potential.
2. A clear separation between the exceptional operation and the error-free operation code.
   This is desirable because one can focus independently on the normal operation and the exceptional operation. In a program, the code is more maintainable because it is clearer what a certain piece of code does;
3. Information about occurrence of an exception must contain all the necessary data for a proper handling.
   There is little use in knowing that an exception was thrown, without knowledge on the context of the exceptional occurrence.
4. The overhead of the exception handling mechanism should stay to a minimum for the error-free operation. In other words, overhead should only occur when an exception is thrown.
5. The mechanism should allow uniform treatment of exceptions detection both by the environment (processor) and by the program.
6. The mechanism should prevent an incomplete operation from continuing, because there is no use in continuing if the correctness of its output cannot be guaranteed.

There are more specific demands for an exception handling mechanism in a concurrent environment:

7. Throwing an exception in one process should start handling in that process and all others that communicate with that one.
8. Hierarchy of exceptions.
   In a concurrent environment, it is possible that two or more exceptions, coming from different sources, arrive at an exception handler at the same time. The exception handler should be able to prioritize the exceptions and handle them accordingly.

Demands on an exception handling mechanism, more specific for CT are:

9. It should comply with the CSP based, process oriented design methodology.
10. It should be expressible in machine-readable CSP (CSPm).
    Strength of a CSP design is that it is formally checkable by a model checker, for example FDR. Therefore, new things that are added to the CT library should be formally described in CSPm.

3.2.3 Examples of exception handling mechanisms

Handling exceptions has to do with multiple hardware and software levels, like the processor, OS, programming language and the structuring of programs. A form of exception handling in hardware is hardware redundancy. When a piece of hardware is broken (the exception) another piece of hardware takes over (the exception handling).

Some exception handling mechanisms that are used in software are discussed below.

Exception handling in C++/Java

In single threaded, sequential C++ and Java programs, exception handling is structured using `try/throw/catch` statements. This also closely resembles the exception handling mechanism of Ada.

The `try` statement embodies a guarded error-free operation. Raising an exception is implemented by throwing an arbitrary object using the `throw` statement. Once an exception is raised, the exceptional operation, which is embodied by the `catch` statement, receives and handles the exception.

Positive things about this kind of exception handling are:

- It gives a clear separation of error-free code and exceptional operation code
- The exception can contain any information that is encapsulated by the thrown object
- When an exception is raised, it propagates to the closest appropriate exception handler, along the call chain

A serious downside is a lack in semantics when used in multithreaded applications. Namely, the models for synchronizing concurrent exceptional operations are not provided by the languages themselves.

Exception handling in C

In C programs, error conditions (exceptions) are handled using function return values with `goto`, non-local jumps using `setjmp` and `longjmp` and OS supported constructs. C does not define any exception-handling facilities within the language. Such an omission clearly limits the usefulness of the language in the structured programming of reliable system (Burns and Wellings, 2001).

Return values & goto

Return values of functions is the most widespread form of error handling in C. They can be combined with `goto` statements or `if/else` statements.

The largest drawback to this technique is that the return value has to be checked every time. If a function returns an error value, the invoker should check for the error code and further signal the error if it is not able to handle the error. In this chain of error return codes, every part must check the return value. In order to make the error propagate to a place it can be handled every part of the code has to check the return values. Certainly, the non-automatic propagation of the error is a large drawback.

Another large drawback to this form of error handling is that the error handling code intertwines with the error-free operation code. This limits the readability and maintainability of the code. In addition, requirement three is not fulfilled since error codes can contain only a limited amount of information about the error causing the exception.

Non-local jumps

Non-local jumps use the functions `setjmp` and `longjmp`. With the `setjmp` function a point can be defined to which the program can later jump back to using the `longjmp` function.

This mechanism looks a bit like the `try`/`throw`/`catch` mechanism of C++. As will be shown in section 3.4.1, a "`try`/`throw`/`catch`"-like mechanism can be made using the functions `setjmp` and `longjmp`. This means that it inherits some of the advantages of the `try`/`throw`/`catch` mechanism like separation between normal code and exception handling code.

The `setjmp/longjmp` technique itself does not provide a coherent way of using it as exception handling mechanism, failing at least the first demands of section 3.2.2. This can only be provided by programming structures that use this mechanism.

OS support for exception handling

Signals

Besides the use of the mechanisms above, provided by the programming language, signals can be used in an environment that supports it. Signals can be seen as software interrupts. The functionality of code that is allowed in a signal handler is in general very limited so most of the time the signal handler can only set a variable. This variable is then checked in the error-free operation. This gives the same problem as function return values; the variable should be checked at certain places. If somewhere such a check is omitted, the exception handling stops functioning properly.

Exception handling in multithreaded environments

For multithreaded applications, the programming language constructs are used (so `try/catch`, function return calls and so on) together with inter process communication like signals, semaphores and message boxes and functions to suspend and resume tasks. Some of the requirements stated in section 3.2.2 can be met using these constructs, but they are not enforced. It largely depends on how the programmer uses the provided constructs.

3.3 Conceptual design of exception handling in CT

3.3.1 Main concept

As stated in section 3.2.3, the programming languages that implement CT libraries, C, C++ and Java, do not provide adequate exception handling mechanisms (for concurrent environments) on their own. Therefore, a concurrency-adequate exception handling mechanism on the level of the CTC++ library is designed.

The main concept of the exception handling design in CT was to make it resemble the C++ and Java `try`/`throw`/`catch` mechanism. This is a concept that many programmers are familiar with and it complies with all of the general demands listed in section 3.2.2 except demand 5.

That the mechanism has to resemble the `try`/`throw`/`catch` mechanism does not mean that `try`/`throw`/`catch` statements must be available in the language. Section 3.4.3 shows the feasibility of implementing the mechanism in CTC and same concept could probably be introduced into new derivatives of Occam (Welch and Wood, 1996).

The process termination model

Because the `try`/`throw`/`catch` mechanism of C++ was made for a sequential program, it only works in one thread of control. The reason for this is the stack unwinding that takes place when an exception is being raised. In a sequential program, the complete program runs on one stack and the stack unwinding can never go beyond the top of that stack. If the top of the stack is reached, the program unconditionally terminates.

This means that when this mechanism is used in a multithreaded environment, that CT is, exceptions must be caught somewhere in the same thread that raised the exception. Otherwise, if the exception were not caught, the stack unwinding would go beyond the top of the stack, which would result in unpredictable behavior, usually termination of the program.

`try`/`throw`/`catch` is an exception handling mechanism using the termination model. This means that a task upon raising an exception is terminated and an exception handler is then executed. When this is transformed to the CT concept, it could be seen as an abnormal process termination and a transfer of control flow to the exception handler. A CT construct reacts to termination events of child processes. When extending CT with the concept of abnormal terminations, the constructs react differently to normal and abnormal termination events.

The exception propagation model

From the moment an exception is raised, it propagates through processes and constructs until an exception handler is found. How an exception propagates depends on the constructs it propagates through. This behavior for the different constructs will be discussed further on in this chapter.

3.3.2 Raising exceptions and exception types

One of the demands mentioned in section 3.2.2 was that there should be enough information for the handler to properly determine how to handle the exception that was raised. In C++ and Java, an exception handler catches objects of specific classes like `int`, `String` or some user-defined class; thus any type of class can be thrown and caught. Within the proposed exception handling mechanism for CT only `ExceptionSets` are being caught and thrown. The `ExceptionSet` is a class that can contain multiple `Exception` instances in a linked list. The reason that `Exceptions` themselves are not caught, but `ExceptionSets`, is that the parallel construct collects exceptions thrown by child processes and throws them higher up. The parallel construct will be further discussed in section 3.3.6 and section 4.4.4.

An exception is always of a type that is derived from the `Exception` class. This `Exception` class contains methods to determine the exact type of the exception in the `ExceptionSet`. This way, the exception handler can distinguish different exception occurrences. The handler can thus handle all the exception types that it knows how to handle and if the `ExceptionSet` contains any more unhandled exceptions, it can throw the remaining `ExceptionSet`. There will be an example of this behavior in section 3.4.8

3.3.3 The EXCeption construct

In the proposed exception-handling concept, a new construct is needed. The new construct is named the *EXCeption* construct. The GML symbol of that new construct is  . This construct catches an `ExceptionSet` that is thrown and directs it to the associated exception handler.

The *EXCeption* construct is associated with a process and an exception handler (See Figure 18). The process encapsulates the error-free operation. This process is run first. When it terminates normally, the *EXCeption* construct terminates normally as well. When the normal process terminates abnormally by throwing an `ExceptionSet`, the *EXCeption* construct executes the exception handler. If this exception handler terminates normally, the exception construct terminates normally. If however, the exception handler terminates abnormally by throwing an `ExceptionSet` the exception construct terminates abnormally.

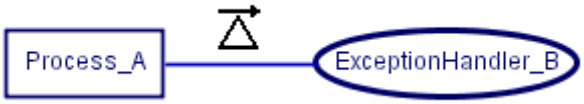
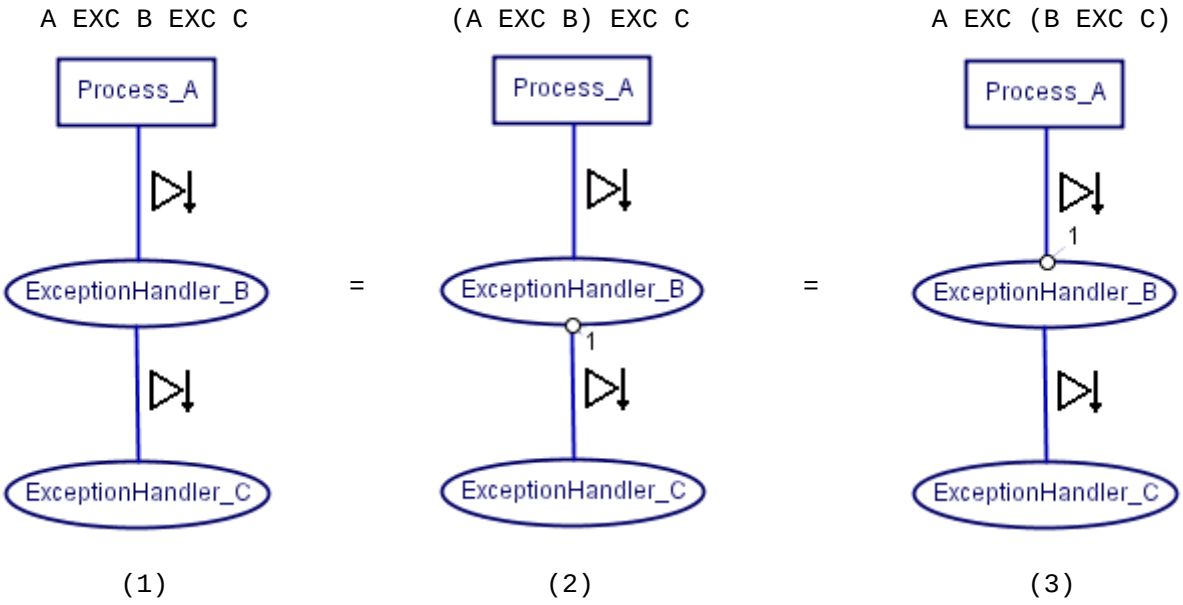


Figure 18. GML model of a process that is guarded by an exception handler

The *EXCeption* constructs can be cascaded so that exceptions that are not handled by the first exception handler can be handled by other exception handlers:



All three of the above options capture the concept of partial exception handling. That means that ExceptionHandler_B may handle some exceptions from the exception set and propagates the remaining exceptions by throwing them higher up. These can then be handled by ExceptionHandler_C.

However, due to the restriction that the *EXCeption* construct can only be associated with a process and an exception handler, option two is the model of implementation for partial exception handling. The parenthesizing on option two indicates the composition of Process_A and ExceptionHandler_B into an *EXCeption* construct, which is again a process. Thus, it can be composed with ExceptionHandler_C in a higher *EXCeption* construct.

Option one cannot be implemented because an *EXCeption* construct can only be associated with one process and one Exception handler. Option three cannot be implemented because here the *EXCeption* construct between ExceptionHandler_B and ExceptionHandler_C would be the exception handler for the upper *EXCeption* construct, which is not allowed because an *EXCeption* construct is a process and not an exception handler.

3.3.4 The Sequential construct

The *Sequential* construct arranges the execution of the child processes in sequence, as in the case of C++ of Java code. The behavior when an exception occurs resembles the behavior of C++ or Java code. The process that is being executed terminates abnormally by raising an exception and consequently, the sequential construct terminates abnormally without running any more processes. Since the genuine behavior of this construct in presence of exceptions complies with the desired (exceptional) flow of control, supporting exception handling on the level of the CT libraries requires no change to the sequential construct.

3.3.5 The (Pri)Alternative construct

The alternative construct makes a choice to run a certain process based on the guards that are guarding channels. A guard observes if the process on the other side of the channel wants to communicate. If a guard, for whatever reason, cannot make this observation, it can signal this to the alternative. Because the alternative cannot make a competent choice, it terminates abnormally by throwing an `ExceptionSet` that contains all the exceptions given by the guards.

If all guards are able to observe if the other side wants to communicate, but no one wants to, the alternative waits for one guard to signal a communication possibility. During this waiting, the guards can still signal a problem to the alternative, which then terminates abnormally.

If one or more of the guards signal that the other side wants to communicate, the alternative chooses one of the guards and executes its associated process. If this process terminates abnormally, the alternative will also terminate abnormally.

Thus, the exception enabled alternative execution consists of three stages:

1. Initial checking of the guards. The guards can signal problems to the alternative. If a problem was found, the alternative terminates abnormally.
2. Waiting for one or more guards to become ready or signal a problem. If a problem was found, the alternative terminates abnormally.
3. Running the associated process. If the process terminates abnormally, the alternative will terminate abnormally.

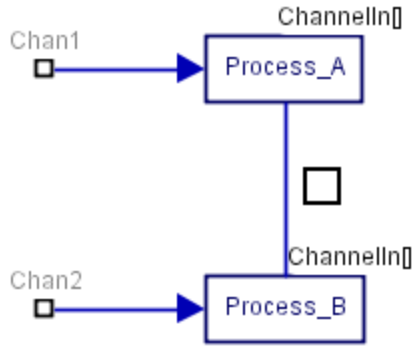


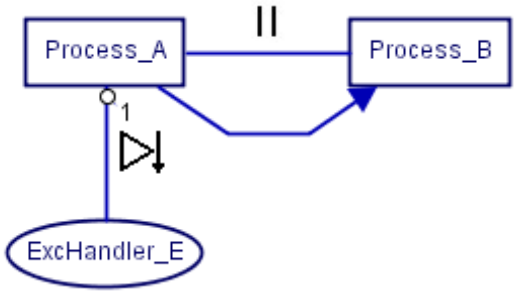
Figure 19. Two processes in an Aternative construct

If, for example, in Figure 19 `Chan1` is an external communication channel, but a cable break is detected, the guard can signal this to the alternative, which will then terminate abnormally. If the guard is able to observe a communication attempt from the other side, `Process_A` will start. If in the execution of `Process_A` an `ExceptionSet` is thrown, it will propagate through the alternative construct.

3.3.6 The (Pri)Parallel construct

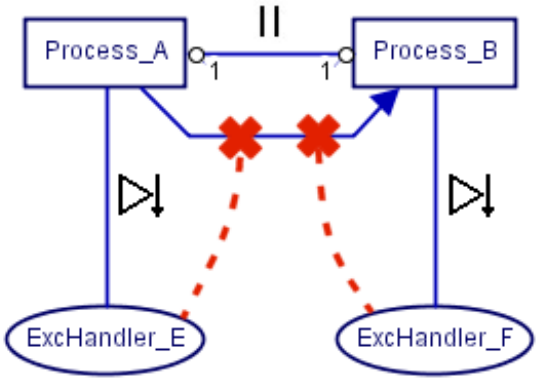
The behavior of the parallel in an error-free operation is that it terminates, when all its child processes have terminated. This behavior stays the same under abnormal termination of any of its child processes. The parallel collects `ExceptionSets` that are thrown by child processes and places their exceptions in one `ExceptionSet`. If this `ExceptionSet` is empty when all child processes have terminated, so there were no exceptions raised by the child processes, the parallel terminates normally. If the `ExceptionSet` of the parallel is not empty when all child processes are terminated, the parallel terminates abnormally by raising this `ExceptionSet`.

This behavior of the parallel can introduce a deadlock condition. Consider the following structure:



If an exception is raised in A, A will terminate abnormally. If it was to communicate with B at some point and B is waiting for this to happen, B will not terminate. It will be blocked, waiting for the communication. Therefore, the parallel construct will not terminate and E will not execute. Therefore, if two processes communicate with each other and they are running in parallel and they can terminate abnormally, each one of them should have an exception handler, which would make sure the other process also terminates.

This problem in the proposed exception handling mechanism is addressed by the channel rejection technique. Now looking back at the example, when A en B could communicate with each other and for example A would be able to terminate abnormally, the structure should be changed to:



In this situation, exception handler E not only handles the exception that occurred in A, but also rejects the channel between A and B, annotated by the bold crosses. This way, the exception handler of A makes sure that B also terminates (although abnormally). If B terminates because of the channel rejection, exception handler F filters out the exception thrown because of the channel rejection.

When exception handler F also rejects the channel between A and B, it doesn't matter any more which of the two processes terminated because of an exception. Due to the channel rejection, the other process will eventually also terminate.

More examples on this will be given in section 4.4.4.

3.4 Implementation in CT

3.4.1 C++ try/throw/catch

As mentioned in section 3.3.1, the concept of exception handling in CT looks a lot like `try`/`throw`/`catch` from C++ or Java. In C++, a piece of code using exception handling is shown in Listing 1

```
try {
//Some guarded code
…
throw(SomeType());
…
} catch (SomeType localInstance) {
//Exception handler
…
  if (!completely_handled)
    throw;
…
}
```

Listing 1. Exception handling in C++

3.4.2 CTC++ TRY/THROW/CATCH/ENDTRY macros

Instead of using `try`/`throw`/`catch` in CTC++ programs directly, macros should be used that contain `try`/`throw`/`catch`. The macros are given below:

```
//inside ExceptionSet.h
#define TRY try
#define THROW(a) throw (a)
#define CATCH(a) catch (ExceptionSet *a)
#define ENDTRY
```

Listing 2. TRY/THROW/CATCH/ENDTRY macros in CTC++

The use of macros was done to make the implementation more platform and language independent between version of the CT library like CTC and CTC++ under RTAI. For CTC and CTC++ under RTAI the implementation using `try`/`throw`/`catch` cannot be used. In C, these functions do not exist and under CTC++ under RTAI, these functions cannot be used because of they are in a dynamic library, which cannot be used in the kernel (See section 2.4).

3.4.3 CTC TRY/THROW/CATCH/ENDTRY macros

For CTC the `setjmp` and `longjmp` functions could be used. The `setjmp` function saves the current processor context into a structure. When the program returns from a `setjmp` that stores the current processor context, the return code is zero. Using the `longjmp` function the program can jump back to a previous saved point. When a `longjmp` is issued, the program will return from the `setjmp` function but now with a return code that is not zero. This way a distinction can be made between the call to `setjmp` and a jump from `longjmp`.

Using `setjmp` / `longjmp`, the TRY macro in CTC would become:

```
#define TRY {\
  state_buf local_buf;                                        \
  local_buf.prev = currentThread->handler;                    \
  current = &local_buf;                                       \
  local_buf.item_thrown = setjmp(local_buf.jmp_buf);          \
  if (!(local_buf.item_thrown))

  //The “else”-branch is in CATCH
```

Listing 3. The TRY macro in CTC

One can see here that every time the `TRY` macro is executed, a new `state_buf` structure is placed on the stack. This structure is filled with a pointer to the structure from the previous `TRY` call. This is needed for a hierarchical exception handling mechanism. Every thread contains a handler pointer that points to the latest added structure. In the `TRY`, this handler pointer is updated to point to the new `state_buf`.

The saving of the processor context causes some overhead, which is the largest downside to this implementation of exception handling in CTC. The time for saving the context on a 350 MHz Pentium II is 0.12 µs (using GCC 3.3.3 for DOS). For a modest program using a low number of `TRY` macros, the overhead will be small, but for more complex CTC programs with heavily nested exception handlers, the overhead will become significant.

The result value from the `setjmp` call is also stored in the new `state_buf`. If it is the first call to `setjmp` the value will be zero. The `if` statement will then execute the error-free code. If a `longjmp` was made, the value returned by the `setjmp` will be the value that was thrown. By storing this value, it can be used in the CATCH statement to give the exception handler the item that was thrown.

The CATCH macro in CTC becomes:

```
#define CATCH(ExceptionSet_Name)                      \
  else {                  \
    currentThread->handler = local_buf.prev;          \
    ExceptionSet *ExceptionSet_Name =                 \
      (ExceptionSet *) local_buf.item_thown;
```

Listing 4. The CATCH macro in CTC

The `else` in the `CATCH` macro belongs to the `if` in the `TRY` macro. This means that the code after the `CATCH` statement is executed when the program jumps back to the `if` statement because of a `longjmp`. The first thing it does is resetting the handler pointer to the previous `TRY` block. This way when a `THROW` is executed again, it jumps back to the previous saved processor state.

After this, the thrown value is retrieved from the structure. This value is appointed to a pointer to an `ExceptionSet`. The name of this pointer is determined by the name given as parameter to the `CATCH` macro.

The ENDTRY macro becomes:

```
#define ENDTRY                                    \
  } currentThread->handler = local_buf.prev; }
```

Listing 5. The ENDTRY macro in CTC

The ENDTRY macro restores the current pointer to the previous processor context. This restores everything that the TRY macro changed. There is no equivalent for ENDTRY in C++ so for the CTC++ implementation this empty.

Using longjmp, the THROW macro in CTC becomes:

```
#define THROW(ExceptionSet_Ptr) {                                    \
   longjmp(currentThread->handler->jmp_buf, ExceptionSet_Ptr); }
```

Listing 6. The THROW macro in CTC

The THROW macro uses the value from the handler pointer to determine the latest saved processor context for this thread and feeds this together with the value to be thrown into longjmp. It will not return from this call to the point after the longjmp. Instead, it will return from the setjmp that saved the used processor context.

The macros given above are a complete solution for CTC.

3.4.4 CTC++ / RTAI exception handling mechanism

The TRY/THROW/CATCH/ENDTRY macros as presented in section 3.4.2 cannot be used for CTC++ under RTAI because try/throw/catch are not available. Without changes, the CTC macros also cannot be used, because setjmp / longjmp are also not available. Setjmp and longjmp are not very difficult functions, so they could be written rather easily. When this is done, exception handling could be implemented using the same macros as the CTC solution. This implementation has one problem however when used with C++. Consider the example given in Listing 7.

```
class SomeClass
{
  private:
  char *buffer;

  public:

  SomeClass() {
    buffer = malloc(a_lot);    //Alocate a lot of memory
  }

  ~SomeClass() {
    free(buffer);                          //Release the allocated memory
  }
}
```

```
void SomeFunction() {
  SomeClass someClassInstance; //Allocate a_lot of memory

  …
  //Code that could execute a THROW
  …

  //Here someClassInstance is destructed and the memory released
}
```

Listing 7. Problematic code when using setjmp/longjmp in C++

In this code, every instance of `SomeClass` allocated a certain amount of memory. This is for instance done when `SomeFunction` is entered. When `SomeFunction` is left, `someClassInstance` is destructed and the memory is given back to the system. This also happens when `SomeFunction` is left because of a C++ `throw` command. That is because `throw` unwinds the stack, which means that it will destroy any objects that are on the stack. For classes with a destructor this includes calling the destructor.

Since `throw` is not available under RTAI, the C++ macros cannot be used. The CTC macros that can be used instead, do not unwind the stack. The `THROW` macro just jumps back to the position of the `setjmp` in the TRY macro and resets the registers on the processor. In the above example, this would mean that `someClassInstance` is not correctly destroyed and the memory it allocated is not released. This in turn would result in a serious memory leakage problem.

3.4.5 Exception

One of the demands of exception handling is that is should be possible to precisely determine what the source of the exception is. This could be done by the possibility to determine the class of the exception that was thrown. For every source of exception, a class would be derived from a more global type, with the Exception class at the top. An example would be an `EncoderOverflowException` that is derived from an `EncoderException`, which in turn is derived from `Exception`.

If this is used, there must be a possibility to determine the type of the class that was thrown. In C++ this is normally done using dynamic casting, which requires RTTI (Run Time Type Interface). Because RTTI is neither available under RTAI and in C, a type checking method was devised for the CT library.

For the type checking method, several options were thought of. The options are first explained and then the pros and cons are mentioned in a table. Example code for all the options can be found in Appendix B.

1. Array in base class

In the first solution, the `Exception` class contains an array with pointers to strings containing names and a counter that contains the derivation level of a certain instance. When an instance of a derived class is instantiated, the constructors of all the subclasses are called in the order of inheritance beginning with the `Exception` class. Each constructor should register itself with its name to the Exception base class, which then stores a pointer to the name and increases the derivation level.

When a type has to be checked of a certain instance, the base class compares the given name with the name of that particular instance and all its lower derived names.

2. Linked list of ExceptionType variables

In this solution, every exception derived from `Exception` should contain an instance of an `ExceptionType` class, which is initialized upon construction of the derived `Exception` with the name of the derived exception and a pointer to the newly formed exception. The `ExceptionType`

registers itself with the `BaseException` class, which is a base for every `Exception`. This way a linked list of `ExceptionTypes` is formed.

When a type has to be checked of a certain instance, the linked list is followed in reversed order. In each step, the name given to the `ExceptionTypes` is compared to the given name to compare it with.

3. Linked list of static ExceptionType variables

This solution looks a lot like the previous mentioned solution, with the difference that the `ExceptionType` variable is a static variable. This enables the type check function to do a pointer comparison to check the type.

4. Virtual functions

In this solution, every derived exception should implement its own virtual function to check the type. If the type is not correct it can direct the type checking to the type it is derived from. This goes on until the base class `Exception` or until the type is correct.

Solution	
1. Array in Base class	+ Only one function call in the constructor – A limited level of derivation Comparison of strings (slow) Derived exception classes should call SetType("..name..")
2. Linked list of ExceptionTypes	+ Looks very clean in the code C++ can use operator overloading for type checking – Comparison of strings (slow) Derived exception classes should contain an ExceptionType variable and initialize this using its constructor.
3. Linked list of static ExceptionTypes	+ Comparison of pointers (fast) Nicer call to check type – Derived exception classes should have a static variable declaration in class declaration and in the class implementation and fill a pointer in base class in the constructor.
4. Virtual functions	+ A nice C++ way. – Derived exception classes should implement all virtual functions comparison of strings (slow) Virtual functions are not available in C. A workaround could probably be made by using function pointers.

From these options, option 2 was chosen for the final implementation, because of the small language dependency and the clean code it produces. The comparison of strings only occurs in the exception path so it imposes no run-time overhead during the normal operation. The second downside to this implementation becomes less of a problem by having code generation for user defined exceptions in the gCSP tool.

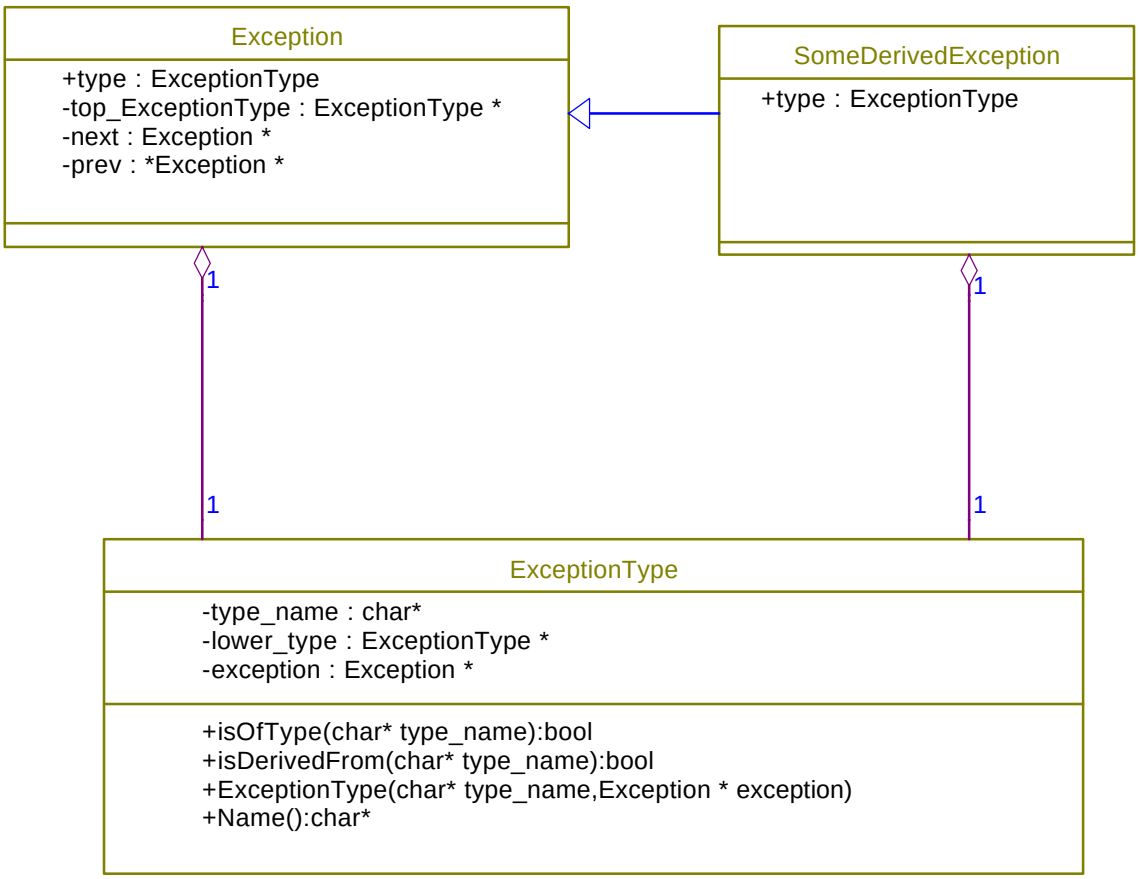


Figure 20. UML diagram of the Exception class, a user defined exception and the ExceptionType class

A UML diagram of the `Exception` class is given in Figure 20. This also shows the `ExceptionType` class. Every exception or derived exception should contain a variable of this type. By calling the `isOfType() method,` one can determine if the exception is exactly of the type that is passed as a string. Using the `isDerivedFrom()` method, one can check if the exception is derived from the exception that is passed as a string and thus is a more specific type then the one passed.

The `ExceptionType` class uses the `exception` pointer to find the exception it belongs to. Then it uses the `top_ExceptionType` pointer to find the last registered `ExceptionType`. It then compares the given string to the string stored in `type_name` and returns true if they are the same. If they are not the same, the `isOfType()` method will return false. The `isDerivedFrom()` method repeats this process with the `ExceptionType` of the parent class (pointed to by the `lower_type` variable). It does so until either the strings match or the `Exception` class is reached and thus a parent class cannot be found.

In CTC++, operator overloading is used such that one can also use == and >=, the former calling `isOfType()` and the latter calling `isDerivedFrom()`.

As mentioned above, the `Exception` class contains an `ExceptionType` variable called `type`. The string given to this variable is “Exception”. Therefore, if one calls the method `isDerivedFrom()` passing it the string “Exception”, the result will always be true, because every exception is derived from Exception.

The `Exception` class also contains two pointers called `prev` and `next`. They are use by the `ExceptionSet` class (See below) and they point to the next and previous exception in the exception set.

```
Class SomeDerivedException : virtual public Exception
{
public:
  ExceptionType type;
  SomeDerivedException() : type(“SomeDerivedException”, this) {
  }

};
```

Listing 8. Example code of a class declaration of a derived exception

When one wants to write a new derived exception, that exception should also contain a variable of the type `ExceptionType` that is initialized in the constructor. Listing 8 shows the class declaration of the `SomeDerivedException` of Figure 20. Although this is not used in this example, a user-defined exception can of course have extra arguments in its constructor, this way providing more information on the source of the exception.

3.4.6 ExceptionSet

An `ExceptionSet` is a collection of zero or more `Exception` objects. `Exception` objects can be added and removed from the set. In Figure 21, a UML diagram is given of the `ExceptionSet` class.

The user can get the first exception of the set by calling the `First()` method. This will return the contents of `first`, which is a pointer to the first exception of the set. At this moment, the `current` pointer is set to this exception. Using the `Remove()` method, the exception pointed to by `current` (the “current exception”) is removed from the set and a pointer to the removed exception is returned. The `current` pointer is updated to point to the next exception. If there is no next exception, the pointer will be null.

The exception that `current` points to can always be retrieved by calling the `Current()` method.

Using the `isHandled()` method, the exception set is signaled that the “current exception” is handled. The exception is removed from the set and deleted. The `isHandled()` method returns the next exception in the set. If an exception cannot be handled, the next exception can be retrieved by calling the `Next()` method. This will update the `current` pointer and return the next exception in the set without removing or deleting the old one.

An Exception can be added to the set by calling the `Add()` method or giving it in the constructor. It is also possible to add the contents of an `ExceptionSet` to another `ExceptionSet`. All the exception are removed from the former and added to the latter. The empty exception set is not deleted.

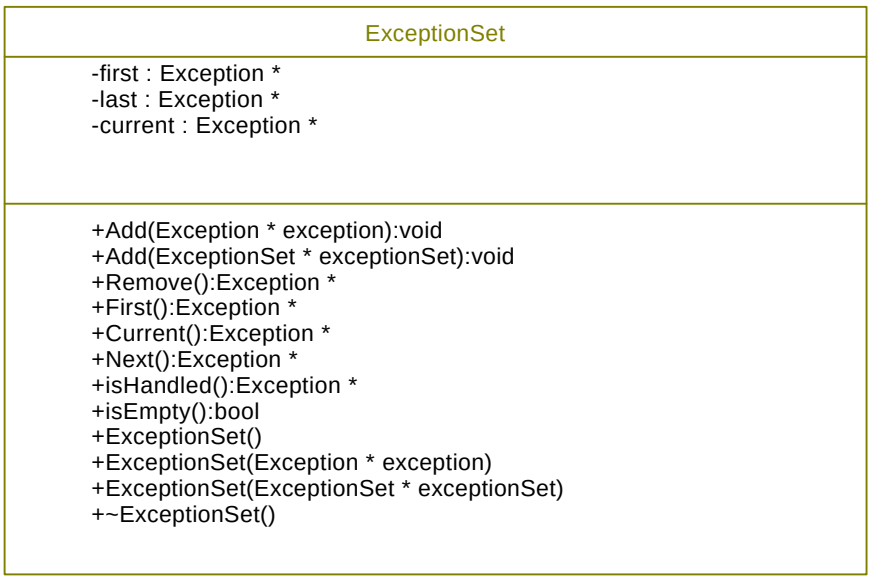


Figure 21. UML diagram of the ExceptionSet class

3.4.7 ExceptionConstruct

The `ExceptionConstruct` class is the implementation of the *EXCeption* construct. The constructor of the `ExceptionConstruct` requires a pointer to a `Process` and a pointer to an `ExceptionHandler`. The implementation of the `run()` method of the `ExceptionConstruct` is given in Listing 9.

```
void ExceptionConstruct::run() {
  TRY {
    process->run();
  } CATCH (exceptionSet) {
    exceptionhandler->run(exceptionSet);
  } ENDTRY;
}
```

Listing 9. Implementation of the ExceptionConstruct::run method

As can be seen the implementation is straightforward. This implementation can also be used in a user written process. This way a separate `ExceptionConstruct` is not necessary, saving memory usage and runtime overhead.

3.4.8 ExceptionHandler

The `ExceptionHandler` class is something that looks a lot like a process, but it is **not** derived from Process. It contains a `virtual run(ExceptionSet *)` method. This routine is called by an `ExceptionConstruct` as the exception handler.

The reason for not making it a normal `Process` is that it needs to know somehow about the `ExceptionSet` that was thrown. If an exception handler is specific for certain exception, it should iterate through the received `ExceptionSet`, handle the exceptions that are supported and then throws the remaining `ExceptionSet` higher up. The exception handling mechanism does not check for which type the handler has been written. It just looks for the first exception handler in the propagation path.

```
void SomeHandler::run(ExceptionSet *eSet) {
  Exception *exc = eSet->First();
  while (exc) {
    if (exc->type.isDerivedFrom("ExceptionName")) {
      //Do things here that are specific for this type of exception.
      //...
      exc->isHandled();
    } if (exc->type.isDerivedFrom(“SomeOtherException”)) {
      //...
      exc->isHandled();
    } else {
      //Place code here for when the exception type cannot be handled.
      //...
      exc = eSet->Next();
    }

  }

  //Do general things here
  //...

  //Check if the remaining set is empty
  if (!eSet->isEmpty()) {
    //And THROW if it is not empty
    THROW(eSet);
  } else {
    //delete the set when it is empty
    delete eSet;
  }
}
```

Listing 10. Template for an exception handler

Listing 10 shows a code template for an exception handler. This exception handler iterates through the given exception set in a `while` loop. Using the `isDerivedFrom()` method, it checks if the exception is of a type that it can handle. When the exception is handled, it is removed from the exception set and deleted by calling the `isHandled()` method, which returns the next exception of the set. If the exception cannot be handled, the Next method is called, to get the next exception in the set.

When the handler has iterated through the set, code is executed that is not specific for the kind of exception. After execution of this code, the handler checks if the set is empty and deletes the empty set if this is the case. If the set is not empty, it means that not all exceptions have been properly handled, and the set is `THROWN`.

3.4.9 (Pri)Parallel construct

The *(Pri)Parallel* construct was changed in such a way that it collects `ExceptionSets` that are thrown by child processes. If all child processes have terminated and one of them threw an `ExceptionSet`, the *Parallel* construct will also throw an `ExceptionSet`, which contains all the exceptions from the child processes.

Because child processes are on different stacks, the `ExceptionSets` that are thrown by them can only be caught on their own stack. Therefore, there must be some code before the actual run method of the child process. This was already necessary, because one cannot determine the address of a non-static method of a class. Such an address is necessary for the scheduler because it has to know where the execution of a thread should start. For this purpose, the `ProcessFix__run` function was written.

For the exception handling mechanism, the `ProcessFix__run` method was changed in such a way that it catches `ExceptionSets` that are thrown by the child process it starts. If an `ExceptionSet` is caught, the set is added to the `ExceptionSet` of the *Parallel* construct that contains the process.

For performance reasons, the parallel contains no `ExceptionSet` when it starts. When the first `ExceptionSet` from a child processes is caught, the *Parallel* construct takes over that `ExceptionSet` as its own. This way, it saves memory and processing time during the exceptional operation.

3.4.10 (Pri)Alternative construct

The *(Pri)Alternative* construct in CT, as analyzed in section 3.3.5, can be seen as having 3 phases. To recall: the first phase is that it checks all the guards, the second phase is that it waits for guards to become ready and the third phase is selecting and running the selected process.

The first phase is always executed. During this phase, the `enableAlting()` method of each *Guard* is called. This in turn calls the `setGuard()` method of the channel it is associated with. In this function, the channel can add an `Exception` to the `ExceptionSet` of the alternative if the channel is not functioning correctly. When the *Alternative* construct has checked all the *Guards*, it checks if its `ExceptionSet` is still empty. If this is not the case, it throws the `ExceptionSet`, which is a collection of `Exceptions` generated by guards/channels.

If the `ExceptionSet` of the *Alternative* construct is empty, it waits until one *Guard* becomes ready. This is done using a semaphore, which is released by the channel when it becomes ready. During this period, the channel can also add an `Exception` to the `ExceptionSet` of the *Alternative* construct and wake up the *Alternative* construct. When the *Alternative* construct wakes up, it checks again if its `ExceptionSet` is empty. If this is not the case, it throws the `ExceptionSet` higher up.

When the `ExceptionSet` is still empty, it depends on the invocation method of the *Alternative* construct if it calls the `run()` method of the process associated with the guard. If the `run()` method of the *Alternative* construct was invoked, it calls the `run()` method associated with the *Guard*. If an `ExceptionSet` is thrown during the execution of this process, it propagates through the *Alternative* construct.

Just like the in the *Parallel* construct, the *Alternative* construct does not contain an `ExceptionSet` of itself. Instead, it takes the first `ExceptioinSet` as its own.

Figure 22 shows three sequence diagrams of possible behavior of the *Alternative* construct when an exception occurs. Example 1 shows the behavior when an exception is signaled during the checking of the guards :

1. The *Alternative* construct calls the `EnableAlting` method of Guard1
2. Guard1 calls the `SetGuard` method of channel1
3. Channel1 adds an `Exception` to the `ExceptionSet` of the *Alternative* construct
4. The *Alternative* construct calls the `EnableAlting` method of Guard2
5. Guard2 calls the `SetGuard` method of channel2
6. The *Alternative* construct throws the `ExceptionSet`

Example 2 shows the behavior when an exception is signaled while the *Alternative* construct waits for a guard to signal a communication attempt:

1. The *Alternative* construct calls the `EnableAlting` method of Guard1
2. Guard1 calls the `SetGuard` method of channel1
3. The *Alternative* construct calls the `EnableAlting` method of Guard2
4. Guard2 calls the `SetGuard` method of channel2
5. The *Alternative* construct waits until it is woken up by a channel
6. Channel1 adds an `Exception` to the `ExceptionSet` of the *Alternative* construct and wakes up the *Alternative* construct

7. The *Alternative* construct throws its `ExceptionSet`

Example 3 shows the behavior when an exception is thrown by a chosen process:

1. The *Alternative* construct calls the `EnableAlting` method of Guard1
2. Guard1 calls the `SetGuard` method of channel1
3. The *Alternative* construct calls the `EnableAlting` method of Guard2
4. Guard2 calls the `SetGuard` method of channel2
5. The *Alternative* construct waits until it is woken up by a channel
6. Channel1 signals a communication event to the *Alternative* construct and wakes up the *Alternative* construct
7. The *Alternative* construct calls the `run` method of Process1
8. Process1 throws an `ExceptionSet`

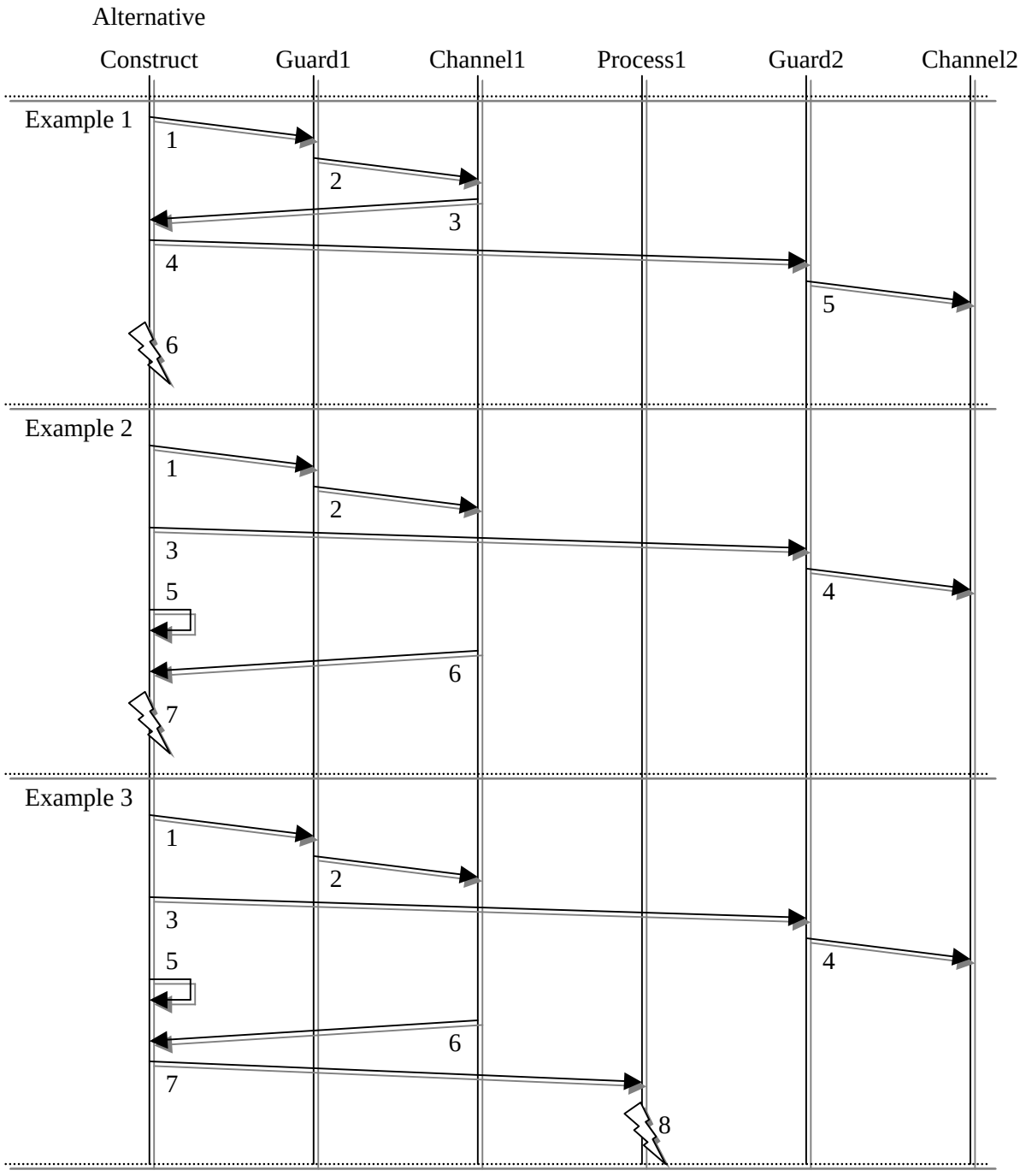


Figure 22. Sequence diagram of the behavior of the *(Pri)Alternative* construct

3.5 Evaluation

The presented exception-handling concept does have some downsides.

The first has to do with the memory allocation. This problem is however not specific for the CT library, but common for all systems.

```
//Some guarded code in our real time process
…
  THROW(new ExceptionSet(new Exception));
…
```

Listing 11. Example code for memory allocation in a real time program

If we look at the code above, it has the problem that memory allocation is done in the real time part. To be able to guarantee that the parent process still meets it deadline, there be a bound on the time it takes to allocate memory. Furthermore, the allocation and de-allocation of the `Exceptions` and `ExceptionsSets` can cause memory fragmentation. This in turn could lead to memory allocation problems, when one wants to allocate a large piece of memory.

One possibility to solve these problems is by allocating a new `ExceptionSet` and `Exception` in the initialization stage of the object that throws the `ExceptionSet`. Either the `ExceptionSet` should be used only once or the exception handler should not delete the `ExceptionSet` and the `Exception`. In the latter case, the number of `ExceptionSets` that the object allocates should be equal to the number of invokers it has, so that it can throw a unique `ExceptionSet` to each invoker. This solution is however not usable when the number of users of an object is unknown, like in an “any to any” channel.

A second downside to this concept is that there is no way of letting the compiler check which exceptions are thrown and if a process is allowed to do so (like the protection in Java and the possible protection in C++). How a certain process reacts to certain exceptions should be mentioned in the specification of the Process. The specification should mention what kinds of exceptions are handled by that process, meaning they are not thrown higher up. It can be assumed that every other exception is thrown higher up by the process.

With respect to property five in section 3.2.2, the mechanism presented in this thesis can only handle software exceptions. When there is a desire to handle hardware exception, these should be mapped to software exceptions by the OS in combination with CT.

3.6 Conclusions

The CT library is enriched with an exception handling mechanism. It is based on unsuccessful termination of processes and looks like the C++/Java `try/catch`. This enhancement contributes substantially on applicability of CSP/CT paradigm in building reliable systems, as it provides the main instrument for fault-tolerance.

The common drawbacks attached to this sort of fault-tolerance provisions are identified and possible solutions are analyzed.

Chapter 4 demonstrates exploitation of the developed mechanism.

4 Exception handling demonstrators

4.1 Introduction and Outline

In chapter 3 an exception handling mechanism for CT has been presented. This chapter will give some examples of how this mechanism can be used. All of the examples are also translated into CTC++ programs.

The examples that will be given in this chapter are:

1. Division by zero.
   This will show the behavior when an exception is thrown from a process.
2. JIWY watchdog cable.
   This will show the mechanism of throwing an exception from an external channel
3. "No zero" channel.
   This will show the mechanism of throwing an exception from an internal channel. This will also discuss the mechanism of channel rejection.
4. Examples of exception handling in an ensemble of parallel processes:
   a. Division by zero with two processes
      This shows the use of channel rejection the concurrent environment
   b. Soft end stops in JIWY
      This will show how the rejection mechanism can be used in real setup.
   c. Comstime with channel rejection
      This will give an example of how channel rejection can be used to terminate a program.

Section 0 will first describe the JIWY setup as it was at the beginning of this research. During this research extra hardware was added to the JIWY setup. For this system, new software had to be written. Both the hardware and the software will be discussed in section 4.3. After this, the examples mentioned above are discussed in section 4.4. Each example starts with a presentation of the mechanism. After this, a concrete execution is given.

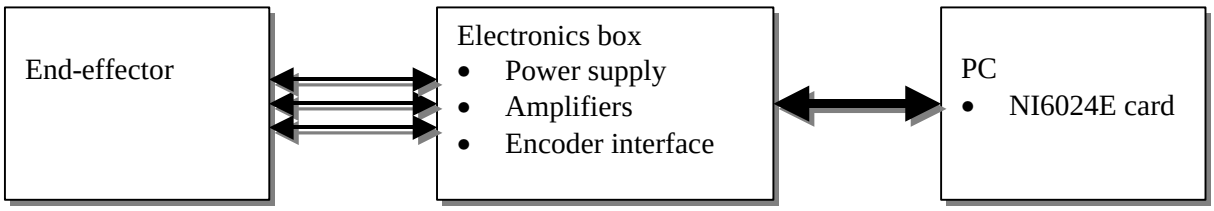


Figure 23. Schematic view of the JIWY setup

4.2 Demonstrator setup

Jiwy is a two DOF (degree of freedom) robotic end-effecter developed at the Control Engineering Lab. The setup is described in (Broenink *et al.*, 2002) and (Jovanovic, Dusko *et al.*, 2002).

4.2.1 Hardware

The JIWY setup consists of 2 parts (See Figure 23). The first part is the mechanical part. It consists of two axes, each having a motor with gearbox, a pulley and an encoder connected to the end-effecter.

The second part is a box that contains a power supply, current amplifiers and some electronics for encoder interfacing. This box is connected to a NI6024E interface card in a PC by means of a cable that contains the low power steering signals, encoder signals and extra, unused signals.

From the power box, three cables run to the mechanical part of the setup consisting of the end-effecter. Two of these cables contain the motor steering and the encoder signals. The third cable can be used for extra signals and will in the future be used for end stop switches. Switches would be placed near the mechanical end stops of the axes, this way providing feedback on when a mechanical end stop is reached.

The three cables all have a spare wire that is not used for control of the setup. Instead, it can be used as a watchdog cable. The idea is that when this cable is interrupted, a signal is given to the processor controlling the setup. The circuitry and software for this feature will be discussed in section 4.3.

4.2.2 Software

Figure 24 shows the model of the original JIWY software. It contains two parallel branches, with no communication between them. Each branch contains five processes that are in sequence.

The RotateLeft processes let the axis turn to one side by maintaining a constant velocity. If the velocity is zero while the motor has a steering value, the RotateLeft processes assume that the axes have hit their mechanical end stops. At this point, the encoder value is saved. This encoder value is the mechanical limit on the left side. The Back processes that are executed next are position controllers that make the axes go back to the initial position. The following RotateRight processes do the same as the RotateLeft processes but to the opposite direction. They store the encoder values for the mechanical limits on the right side. The next processes in the sequential constructs are the Jiwy processes. These are also position controllers. The desired positions are read two axes of a joystick. The center position of the joystick corresponds to the center position of JIWY.

The encoder values of the center position of JIWY are determined on basis of the encoder values of the mechanical limits determined by the RotateLeft and RotateRight processes.

The Jiwy processes terminate upon reading that the stop button on the joystick has been pressed. The sequential constructs then start to execute the Home processes. These are also position controllers that set JIWY back to its center position. The Home processes will stop when the center position has been reached.

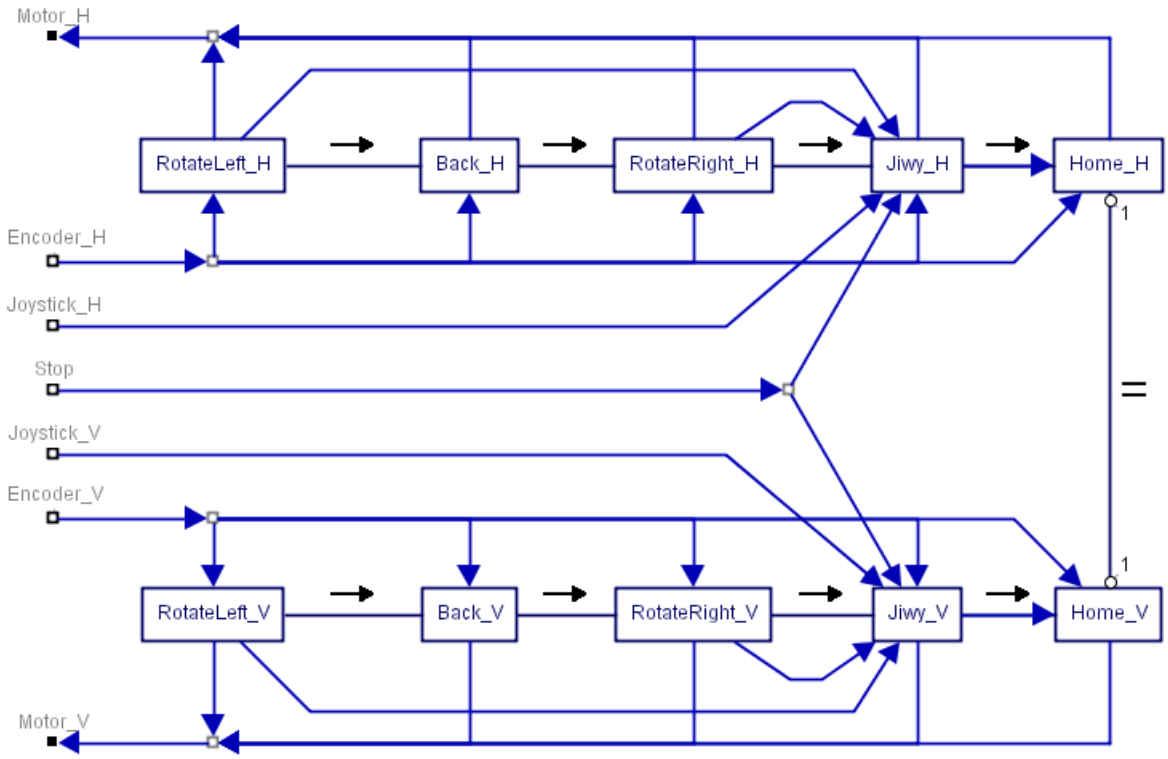


Figure 24. GML model of (old) JIWY software

4.3 Digital Inputs

As mentioned in the previous section, provisions had been made for using watchdog cables and end stop switches. They were however not yet implemented. Part of this assignment was to implement the watchdog cables. They could then be used as examples for the exception handling mechanism.

4.3.1 Hardware demands

The current control of the setup is done using a PC that contains a NI6024E interface card. This card contains sixteen analog inputs, two analog outputs, two counters and eight digital inputs. Of these eight digital inputs, two are already used by the optical encoder interface. This means that there are six digital inputs left.

When the two desired extensions, namely the watchdog cables and the end stop switches, would be incorporated, seven digital inputs are needed at least: three for the watchdog cables and four for the end stops (two switches for both axes). More digital inputs would be welcome, because then extra switches, like safety switches, could be attached in the future. Because only six inputs were available and at least seven were needed, extra digital inputs had to be added.

Because the expected rate of change of the inputs is very low, polling would, in total, give more overhead then interrupt driven behavior. Having the inputs interrupt driven also makes the demonstrator better in the sense, that CT responsiveness to external events can be clearly shown.

Since the NI6024E card has too few digital inputs and is not capable of generating interrupts that indicate a change to the digital inputs, additional interfacing hardware is designed. For this extra hardware, there were a few options:

A new interface card

This interface card would be connected to the PCI bus and would typically contain 24 to 48 IO pins. It would support Interrupt on Change, which means that when an input changes, an interrupt is generated.

Extra circuit connected to parallel port

The extra circuitry that would be added would provide the Interrupt on Change functionality. The interrupt output of the circuitry would be connected to the interrupt pin of the parallel port. The parallel port has enough inputs to read three watchdog signals and four end switch signals. However, multiplexing the signals could be an option, this way providing more then eight inputs for future expansions.

Extra circuit connected to NI6024E card

In this case, like in the previous, the extra circuitry would provide the Interrupt on Change functionality. The interrupt output of the circuitry would be connected to a GATE input of the NI6024E card. When the counters on the NI6024E card are used as encoder interface, the GATE signals are not used and can be used to generate an interrupt to the processor. Because there are only six usable digital inputs left on the NI6024E card, the end switch signals and the watchdog cable signals would have to be multiplexed. The output of the multiplexer would be connected to the digital IO pins of the NI6024E card.

Solution		
1. New interface card	+	Lots of digital inputs
	–	New drivers had to be written
		Extra cable connection from the JIWY box to the PC
		Expensive
2. Extra circuit connected to the parallel port	+	Exactly meets the design requirements
	–	Design time of the circuitry
		Extra cable connection from the JIWY box to the PC
3. Extra circuit connected to the NI6024E card	+	Exactly meets the design requirements
	–	Design time of the circuitry
		Still a limited number of inputs (although 12 should be enough)

It was decided to implement option 3, the extra circuitry connected to the NI6024E card. The cost of this solution would be low, it probably cost only a little bit more time and there are no extra cables needed between the controlling computer and the JIWY setup.

4.3.2 Debouncing

When a switch is being closed or opened or a cable is being connected or disconnected, the contact is never made or released instantly. Because of the contact bouncing, the software can think that the input changed many times, very rapidly. This is undesirable and can cause misinterpretation in the software.

For a 3-pole switch, this can be circumvented by using the schematic given in Figure 25. The problem with 3 pole switches comes mostly from the transition period between breaking contact with one side and making contact with the other. During this period, the signal has no clear defined state. In given circuit, the capacitor stores the state of the signal during these transition periods, until definitive contact has been made. This provides an adequate method to prevent bouncing.

Besides the end switches, the watchdog connection also had to be debounced. This type of connection can be seen has two states (connected or not) in contrast to the three states of a 3-pole switch. For the watchdog, it was desirable to have a quick detection of the disconnection event. When the cable would be reconnected, the event detection mechanism could be slower.

For debouncing the watchdog cables, the schematic of Figure 26 has been made. Shown is only one cell, for debouncing a single switch or cable. Six of these cells were added to the circuitry so more 2-pole switches could be added in the future.

The FET in combination with the resistor connected to the gate inverts the signal. This was necessary to have a fast detection of disconnection events. The FET, the capacitor and the resistor connected to them form a delay stage. When the FET conducts current (so when the switch is opened), the capacitor quickly discharges, this way giving a sharp negative edge on the signal line. If the switch is closed, the FET stops conducting current and the capacitor begins to charge. When the threshold of the input is reached, a new signal level is detected. If during this charging, the switch opens again (because of contact bouncing) the capacitor discharges rapidly again. It is not until a steady state of the switch is reached, that the signal level reaches the input threshold.

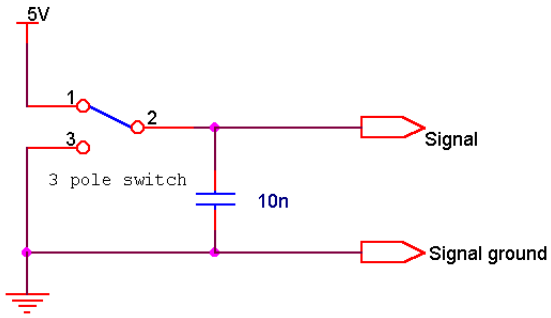


Figure 25. Schematic for debouncing a 3-pole switch

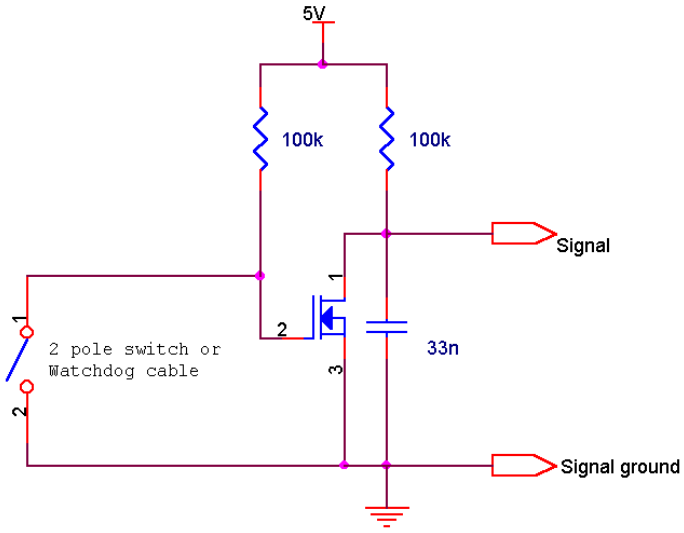


Figure 26. Schematic for debouncing a 2-pole switch

4.3.3 Print buildup

The extra circuitry was built up using some analog electronics for the debouncing of the watchdog cables and an Altera EPLD for the digital part. The complete schematic of the print is given in Appendix C, together with the AHDL code that is in the EPLD.

Figure 27 gives an overview of the parts on the print and what is in the EPLD. For the inputs from the watchdog cables, the print contains the debouncing circuitry given in section 4.3.2. The capacitors for debouncing the end switches can be mounted in the switches themselves, so they are not placed in the print.

The twelve debounced signals go into an Altera EPLD. This first synchronizes the inputs to a clock the circuit receives from the NI6024E card. This sampling also removes glitches in the input signal that have a width equal to or less then the clock width.

The values that come from the sampling/filtering part are constantly compared to the value that was last read by the program. If these differ, the interrupt output is set high. This rising flank can trigger an interrupt on the NI6024E card. When a read action is signaled, the EPLD updates its internal registers that store the last read value, this way resetting the interrupt output.

The twelve inputs to the print are multiplexed into two groups of six inputs. Which of the two groups is read is determined by the same signal as the one that signals the read operation. This means that always the six end switch inputs are read first and then the six watchdog cable signals.

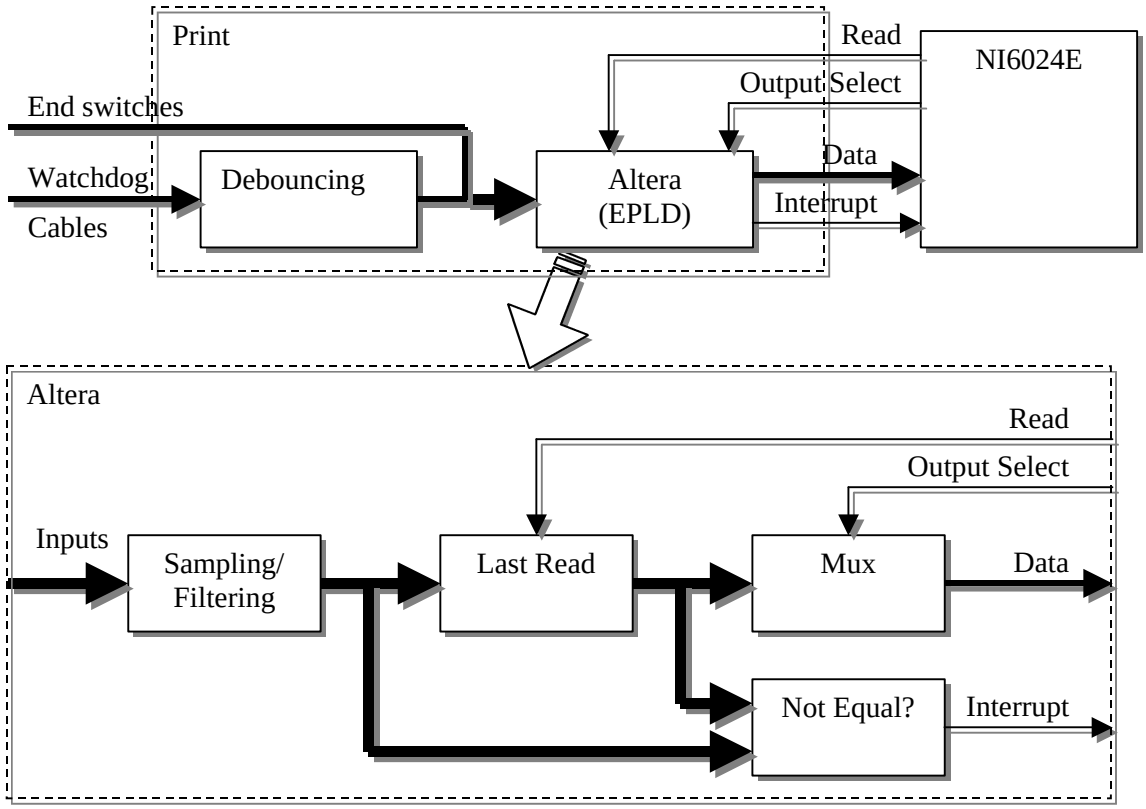


Figure 27. Contents of JIWY print

4.3.4 Software changes

Digital input device driver

The design of the extra digital input circuit is such that data should be read only once for every interrupt, otherwise interrupts can get lost, i.e. input changes can go undetected. Consequently, it must not be allowed that multiple CT channels read the digital inputs independently.

Updating multiple channels can be done in two ways. The first solution is a single channel that reads all the digital inputs at once. This channel is connected to a process that distributes the separate bits to the processes that need them. The downside to this solution is that the individual values can only be distributed to processes and not to other channels. Therefore, for instance the encoder channel cannot check the status of the watchdog cable and raise an exception if the cable is broken.

The second solution, which is used here, is that a digital input device driver does the reading from the hardware and distributes the desired values to individual registered channels. Channels can provide read values to processes or use them for other purposes, for example error detection. In this way, the encoder channel can check the status of the watchdog cable.

Linkdrivers

During a previous assignment, (link)drivers for the NI6024E card were written (Stephan, 2002). Because of the new options of exception generation when a watchdog cable is broken, some extensions had to be added.

First, some changes have been made to the NI6024E drivers. An interrupt manager for the card was written, which distributes an incoming interrupt to registered routines, depending on the source of the interrupt. The timer/counter driver was changed, such that one can choose if a flank on the Gate pin triggers an interrupt.

Second, the linkdrivers for the encoders have been changed. During a read of the old linkdriver the reading task is blocked. The reading task is only unblocked by a timer interrupt. In the interrupt-routine, the encoder value is read and the blocked task gets unblocked. If however the watchdog cable

has been broken, it can be assumed that the encoder signals have been corrupted, which means the encoder value is incorrect. In this case, an exception should be thrown.

Listing 12 shows the code of the new linkdriver. In the new linkdriver, the reading task can also be unblocked by an interrupt from the NI6024E card. The interrupt is passed through the NI6024E interrupt manager, then through the digital input device driver to the channel. If the new linkdriver observes that the watchdog cable is broken, it sets the `cables_problem` variable and unblocks the reading task. The current state of the watchdog cable is always stored in the `current_status` variable. When the suspended task is resumed, it checks if an error occurred by looking at the `cable_problem` variable. If an error occurred, the current state of the watchdog cable is read. If the status is ok, the `cable_problem` variable is reset until a new problem arises. Finally, a new `ExceptionSet` containing a `WatchdogException` is thrown if a problem occurred.

More explanation on using interrupts to detect error situations is given in section 4.4.2.

```
void NewLinkdriver::read(int *data) {
 //Check if there has been a cable break, or that the cable is still broken
 if (cable_problem) {
  if (current_status == CABLE_OK)
   cable_problem = false;
  THROW(new ExceptionSet(new WatchdogException(encoder_nr)));
 }

 //Read a encoder value. This will block the process until the encoder is
 //read in a timer interrupt or a cable interrupt signals a problem
 OldLinkdriver::read(data);

 //Check again if there has been a cable break
 //It might have changed while it was blocked during the read
 if (cable_problem) {
  if (current_status == CABLE_OK)
   cable_problem = false;
  THROW(new ExceptionSet(new WatchdogException(encoder_nr)));
 }
}
```

Listing 12. Example code of the changed encoder linkdriver

4.4 Examples

4.4.1 Raising exceptions in a process

Mechanism

Figure 28 shows a basic GML model of a process that is guarded by an exception handler. When an exception is thrown in Process1, the *EXCeption* construct catches it and redirects the exception to ExcHandler1 by calling its `run(ExceptionSet *eSet)` method. This can then handle the exception in the exception set. Listing 13 shows example code of how an exception is thrown.

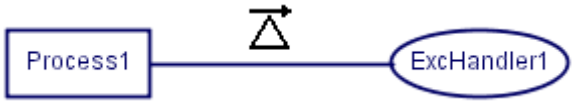


Figure 28. Basic model of throwing exception from a process

```
#include “csp/lang/include/ExceptionSet.h”

…

void Process1::run() {
…
  if (someErrorCondition) {
    THROW(new ExceptionSet(new SomeExceptionType(arguments)));
  }
…
}
```

Listing 13. Example code of throwing an ExceptionSet from a process

Example 1: Division by zero

Figure 29 shows an example of the mechanism described above. The Calculate process reads values from the keyboard. In this case, it does so using C++ functions, but it could also do so from a keyboard channel. After a value is read, it tries to divide a constant by the value that has been read. Before performing the division, it checks if the value that has been read is not 0, which would result in a division by zero exception from the processor. Because processor generated exceptions cannot be caught by the CTC++ library, the program should do this check itself. If this value is 0, a `DivisionByZeroException` object is allocated and added to a new `ExceptionSet` object. A pointer to this `ExceptionSet` is then thrown. The pointer is caught by the *EXCeption* construct. This calls the `run(ExceptionSet *)` method of CalculateHandler, which then prints an error to the screen and terminates.

By not placing the error handling code inside the Calculate, but let Calculate throw an exception, the response to an error situation can be changed more easily, because the exceptional operation is isolated in the CalculateHandler process. Beside this, the Calculate process gets more general and therefore more reusable.

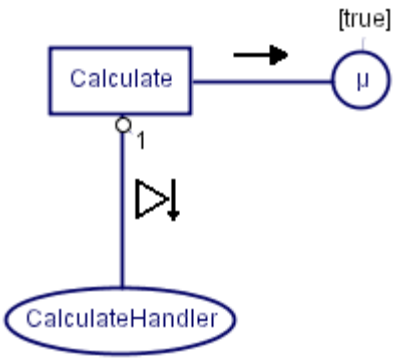


Figure 29. Model of division by zero example

4.4.2 Raising exceptions in an external channel

Mechanism

Discussed above is the mechanism for raising exceptions in a process. Instead of throwing an `ExceptionSet` in the code of the process, it can also be thrown in the code of a channel, both internal and external. This example will discuss throwing from an external channel. Throwing from an internal channel will be discussed in section 4.4.3.

When an exception is thrown from an external channel, the `THROW` is executed in the code of the channel (the read or write method). This code is however executed in the context of a certain process.

If an exception is thrown inside this code, the exception will be caught in the context of the calling process. Effectively, the mechanism of throwing, propagating and handling an exception is the same regardless if an `ExceptionSet` is thrown from a process or a channel. This keeps the mechanism uniform while it lets the user choose to guard the code at the most appropriate place. It may depend on the application or the user's taste to place guarding code in processes or channels or both. This allows a separation of error detection concerns: in complex fault-tolerant distributed software all anticipated errors can be divided into those arising from data communication (detected by channels) and those arising from data processing (detected by processes).

In Figure 30, the read method of the external channel `input` is executed in the context of Process1. This means that if an exception is thrown in the code of `input`, the exception is caught by the `ExceptionConstruct`, which in turn will call the `run(ExceptionSet *)` method of ExcHandler1.

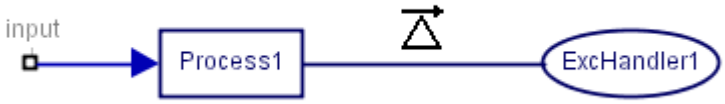


Figure 30. Basic model of throwing an exception in an external channel

When one wants to throw exceptions inside an external channel, one must make sure that this is not done inside an interrupt routine. Because an interrupt routine is **not** executed in the context of the acting (reading/writing) process, one should not throw an exception inside the interrupt routine. The best solution is to store the fault condition in the channel, wake up blocked processes, and then have a check on fault conditions inside the read or write method. Listing 14 gives example C++ code for an external input channel that uses interrupts.

When interrupts are not used, a simple check at the beginning of the read or write method on possible error conditions is sufficient. See Listing 15 for an example of an external output channel that does not use interrupts.

Example 2: Single JIWY axis with watchdog cable

As an example of throwing from an external channel, the JIWY setup is used. The old program was used with a few changes. The first is that the new encoder channels are used, that check for continuity on the watchdog cable. If there has been a moment of discontinuity, the new encoder channel throws an exception.

The second change was the addition of an `ExceptionHandler` for every axis. This exception handler checks for exceptions that are thrown because of discontinuity in the watchdog cable. If such an exception was thrown, the exception handler sets the output to the motor to zero, prints a message to the screen and waits for a stop or restart signal from the joystick. If it receives a stop signal, the exception handler stops, this way stopping the program for this axis. Because the other axis also receives the stop signal, the complete program stops.

If the exception handler receives a restart signal, it reads from the encoder and catches any exception that is thrown. If no exceptions are thrown, it means that the cable shows continuity again and the code for this axis restarts from the alignment phase. If an exception was thrown during this read, a message is printed that tells that the cable is still not working. The exception then again waits for a stop or restart signal. This goes on until a stop is received or a restart is received and the cable is in order.

```
void InterruptChannel::read(double *data) {
  checkForErrorAndThrow();

  checkForDataAndBlock(double *data);

  checkForErrorAndThrow();
}

void InterruptChannel::checkForErrorAndThrow() {
  if (problem) {
    if (currentStatus == STATUS_OK) {
      problem = false;
    }
    THROW(new ExceptionSet(new SomeChannelException()));
  }
}

void InterruptChannel::checkForDataAndBlock(double *data) {
  if (!dataAvailable) {
    lock();
  }
  if (dataAvailable) {
    *data = *tempData;
    dataAvailable = false;
  }
}

void InterruptChannel::dataInterruptHandler() {
  //This interrupt routine is called when new data has arrived
  dataAvailable = true;
  tempData = readDataFromHardware();
  unlock();
}

void InterruptChannel::errorInterruptHandler() {
  //This interrupt routine is called when the error condition has changed
  currentStatus = readStatusFromHardware();
  if (currentStatus != STATUS_OK) {
    problem = true;
  }
  unlock();
}
```

Listing 14. Throwing in an external channel that does use interrupts

```
void NonInterruptChannel::read(int *data) {
  if (readStatusFromHardware() != STATUS_OK) {
    THROW(new ExceptionSet(new SomeChannelException()));
  }

  *data = readDataFromHardware();
}
```

Listing 15. Throwing in an external channel that does not use interrupts

Figure 31 shows the model of one JIWY axis. In the demonstrators, two of these axes run in parallel, each controlling one degree of freedom. There is no communication between the axes, so when an exception is thrown in one of the axis, the other axis continues normally. Section 4.4.4 will discuss solutions for concurrent exception handling, in which the other axis also terminates.

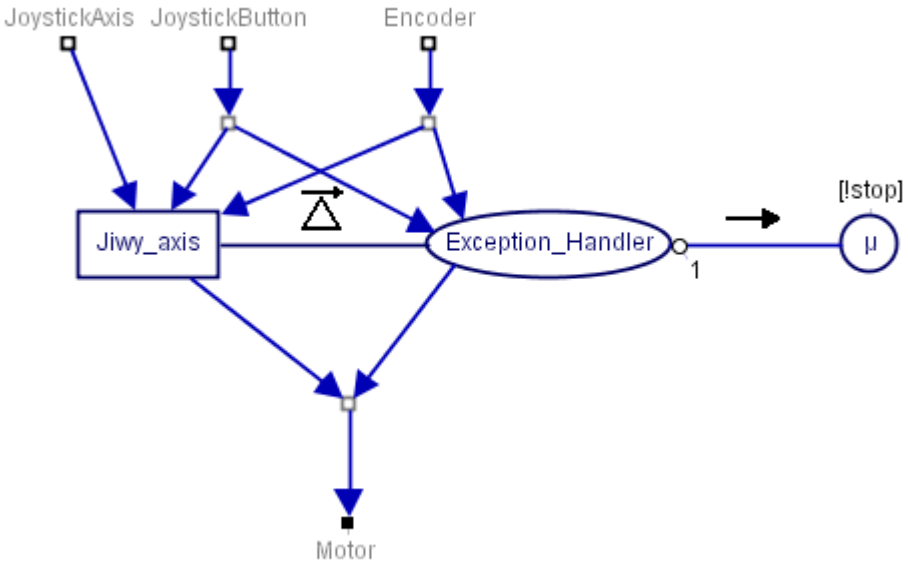


Figure 31. Model of one JIWY axis with exception handling for the watchdog cable

4.4.3 Raising exceptions in an internal channel (to two sides of a channel)

Mechanism

For exception throwing in an internal channel, the mechanism is the same as when throwing an exception in a process or an external channel. This means that when an exception is thrown in the code of the channel, it is caught in the context of the process that is executing that code.

If processes are blocked on code in the channel and the error condition arises, the blocked processes should be unblocked. When they resume their execution of the code of the channel, a check should occur for the error condition and an `ExceptionSet` should be thrown.

Figure 32 shows a basic model of exception handling when the exception is thrown on an internal channel. Suppose the channel in the figure throws exceptions to two sides. Because eventually in the context of both Process1 and Process2 an exception is thrown, they both terminate. For Process1, ExcHandler1 is executed and for Process2 ExcHandler2 is executed. When both exception handlers have terminated, the *Parallel* construct also terminates.

If the internal channel would only throw an exception to one side, only that side will terminate and the other side will block on the communication event. This is again the problem described in section 3.3.6. A solution to this problem will be given in section 4.4.4.

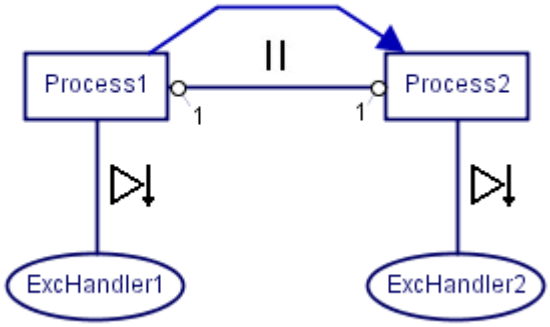


Figure 32. Basic model of throwing exceptions on internal channels

Example 3: NoZeroChannel

Figure 33 shows a model that is derived from the model in Figure 32 while the functionality is equivalent to example 1. In this example, there are two processes, ReadfromKeyboard and Calculate, which together do the same as the Calculate process of example 1. The two processes communicate through a channel. ReadFromKeyboard reads a value from the keyboard (using C++ functions) and sends that value to the Calculate process. This process tries to divide a constant by the received value and prints the output to the screen.

Because division by zero is illegal, a channel was adapted to throw an exception to both sides of the channel when a zero is written to the channel. Therefore, when a writer tries to write zero to the channel, the channel stores this value and then throws an `ExceptionSet` that contains a `ZeroWrittenException` to the writer. If a reader was blocked on the channel, it gets unblocked. When the execution of the reader continues (inside the read method), an `ExceptionSet` containing a `ZeroWrittenException` is also thrown to the readers' side. Both ReadFromKeyboard and Calculate terminate because of the `ExceptionSets` that have been thrown and their exception handlers are executed. They both print a message on the screen and terminate.

gives the code of the read and the write method of the channel from this example. The structure for the read method looks a lot like the code of the read method of Listing 14. This is because from a reader's point of view, a write action is an asynchronous event, just like an interrupt.

The structure of the write method is a bit different, because the error condition is caused in the write method. If both sides could cause an error condition, the code of the write method would more closely resemble the code of the read method.

Channel rejection

When a channel is rejected, any action on this channel like reading and writing, but also guarding, will result in throwing an exception (except rejecting the channel again). In CTC++ this is an exception of type `ChannelRejectedException`. If an action already started but got blocked, the blocked process is woken up and an exception is thrown in the context of that process.

The concept of channel rejection is a very useful tool to terminate running processes. This can be useful for termination of the complete program or parts of the program.

The code in the ChannelBase.cpp file of the CTC++ library is also a good example of how exceptions can be thrown to two sides of a channel.

```
virtual void read(char *data) {
  //Check for a possible error condition
  if (zeroWritten == true) {
    THROW(new ExceptionSet(new ZeroWrittenException()));
  }

  //Wait for a writer to come along!
  waitingreaders++;
  Channel<char>::read(data);
  waitingreaders--;

  //Check for a possible error condition
  if (zeroWritten == true) {
    THROW(new ExceptionSet(new ZeroWrittenException()));
  }
}

virtual void write(char *data) {
  // Check for a possible error condition
  if (*data == 0) {
    zeroWritten = true;
    //Check if there is a readers waiting!
    if (waitingreaders > 0) {
      //And wake it up if there is by finishing the communication
      Channel<char>::write(data);
    }
    THROW(new ExceptionSet(new ZeroWrittenException()));
  } else {
    zeroWritten = false;
  }

  //Everything is OK so wait for a reader to come along
  Channel<char>::write(data);
}
```

Listing 16. Example code for throwing in an internal channel

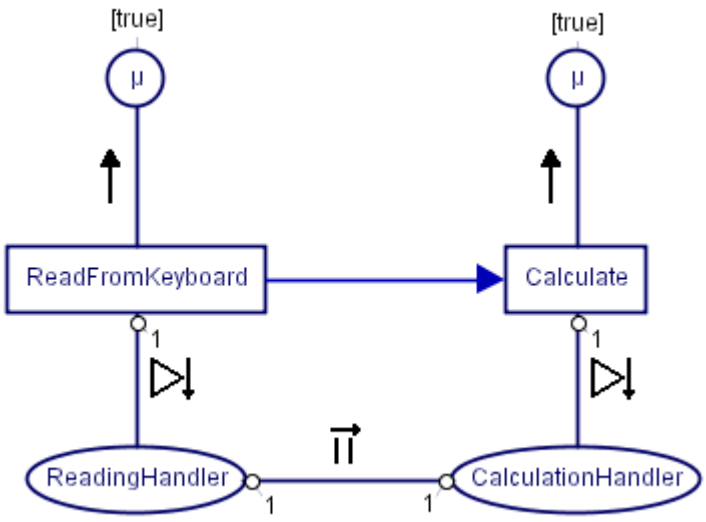


Figure 33. Example of throwing exceptions on internal channels

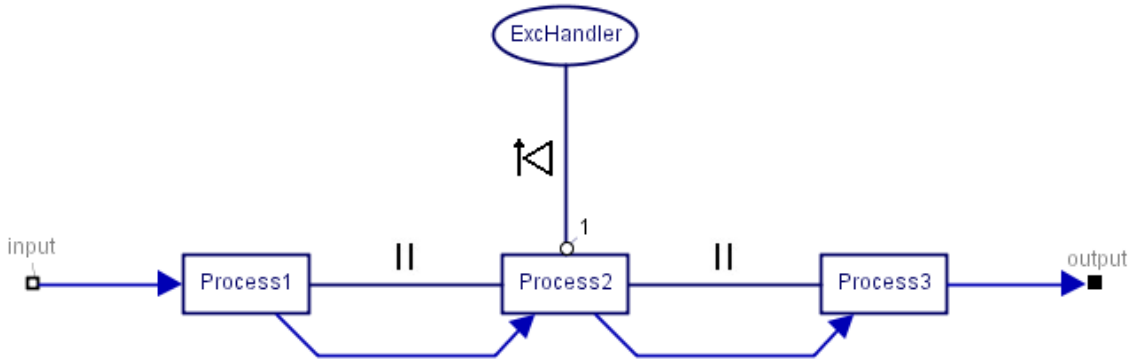


Figure 34. Basic model of raising exception in a parallel composition

4.4.4 Raising exception in parallel compositions

Problem and solutions

The basic problem is already discussed briefly in section 3.3.6. Another example is given in Figure 34. If an exception is thrown from Process1 (which consequently gets terminated), the thrown exception is not handled until the parallel that contains Process1 terminates. This only happens when Process2 and Process3 also have terminated. If these do not terminate, but instead Process3 waits for communication with Process2 and Process2 waits for communication with Process1, the exception remains unhandled and the program deadlocks.

If termination of Process2 and Process3 is not necessary for a proper handling of the exception, a possibility is to associate the exception handler with Process1 (See Figure 35). The combination of Process1 and ExcHandler1 should then also restart, else there is still the possibility that Process2 and Process3 get blocked and thus cannot terminate.

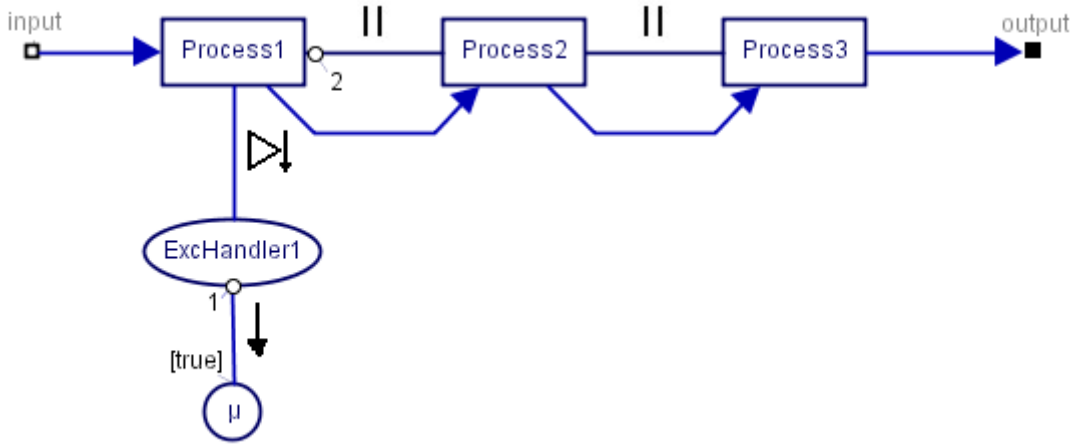


Figure 35. Moving the exception handler

If termination of Process2 and Process3 is required, a more elaborate scheme is needed. In this case, Process1 needs a separate exception handler. This exception handler has to signal Process2 and Process3 to stop somehow. After this, it should throw the received exception again so it can be

handled by the higher exception handler ExcHandler, when Process2 and Process3 are finished (See Figure 36).

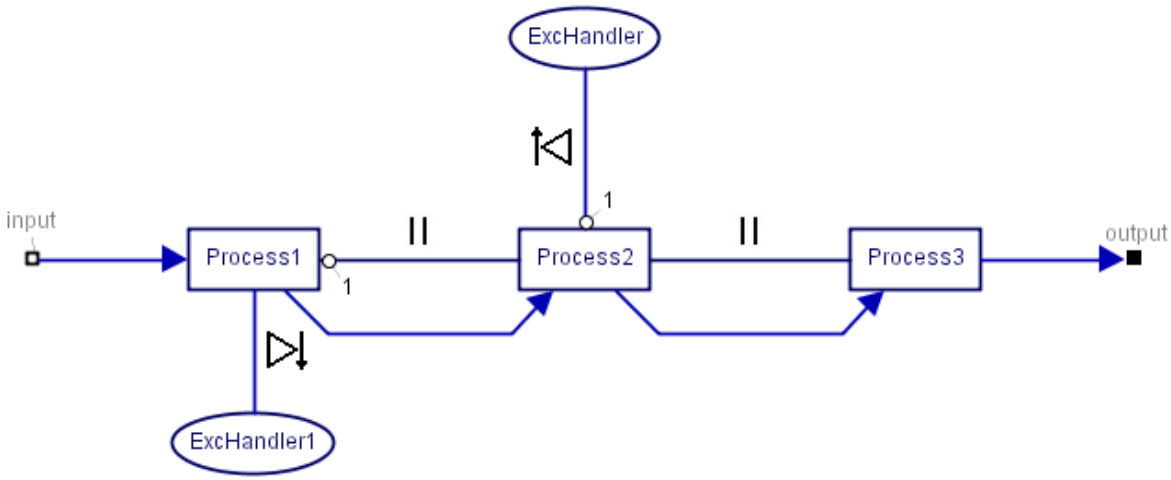


Figure 36. Process1 with ExcHandler1

As already mentioned, ExcHandler1 should stop Process2 and Process3. It has two options to do so: using channel communication (a stop channel) and using channel rejection.

Stop channel

One solution is to use channels to Process2 and Process3 that signal them to stop. This can become hard to reason about when Process2 and Process3 also have a channel from other processes that tells them to stop. When not properly designed, there is a risk of deadlock. This occurs when Process2 or Process3 have already terminated because of a signal from a fourth process. In this case, ExcHandler1 cannot complete the communication with Process2 and Process3 and thus will not terminate.

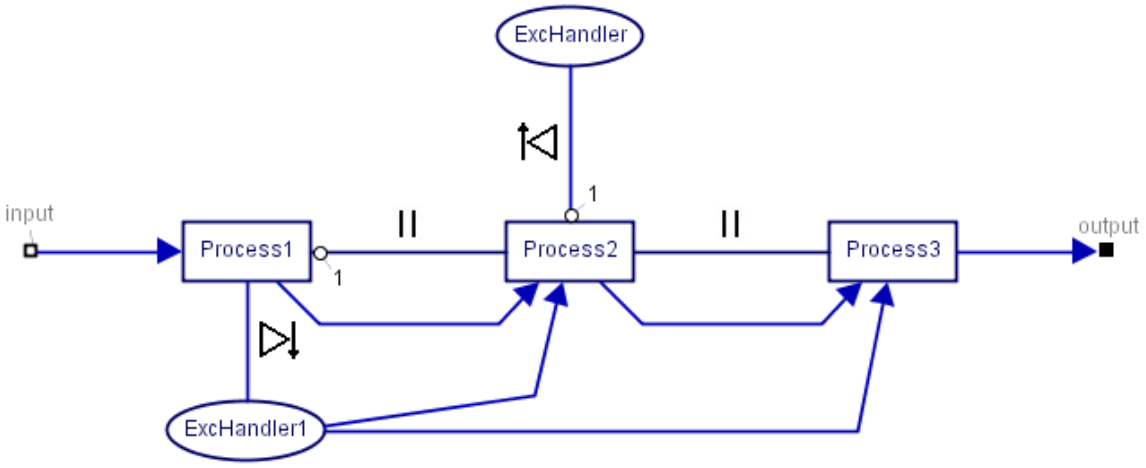


Figure 37. Stopping processes using channels

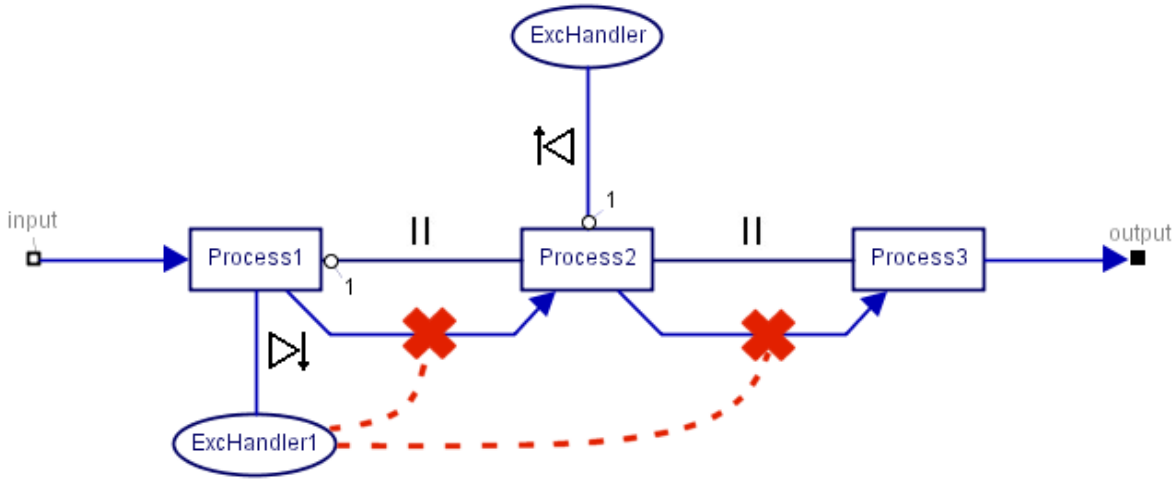


Figure 38. Stopping processes using channel rejection

Channel rejection

Another solution is to have ExcHandler1 reject the channel between Process1 and Process2 and the channel between Process 2 and Process3. See Figure 38 for a GML model of this solution, where rejections are annotated with bold crosses.

The result will be that as soon as Process2 and Process3 try to perform an action on the channel, they will both terminate. At this point all the processes are terminated, and the parallel construct will throw its own `ExceptionSet`. This `ExceptionSet` contains all the exceptions from the `ExceptionSets` that were thrown by its child processes. This `ExceptionSet` will be caught by the *EXCeption* construct which then calls ExcHandler. ExcHandler can now filter out the exceptions that were thrown because of the rejection of the channels and handle the exceptions that remain.

The model in Figure 38 is a simple model that suffices when exception can only be raised in Process1. A more general solution is that Process1, Process2 and Process3 all have an exception handler that rejects the channels the process communicates through. Figure 39 shows a GML model of this situation.

When this solution is used, it does not matter where an exception is thrown. Every process will terminate because all channels will get rejected. Because the exception handlers of the processes can filter out the exceptions caused by channel rejection, only the exception that matters arrives at ExcHandler. If exceptions are thrown in two processes at the same time, they both will be collected by the parallel and they both propagate to ExcHandler, which can then handle both exceptions. A downside to this general solution is that more try/catches are executed, which takes execution time.

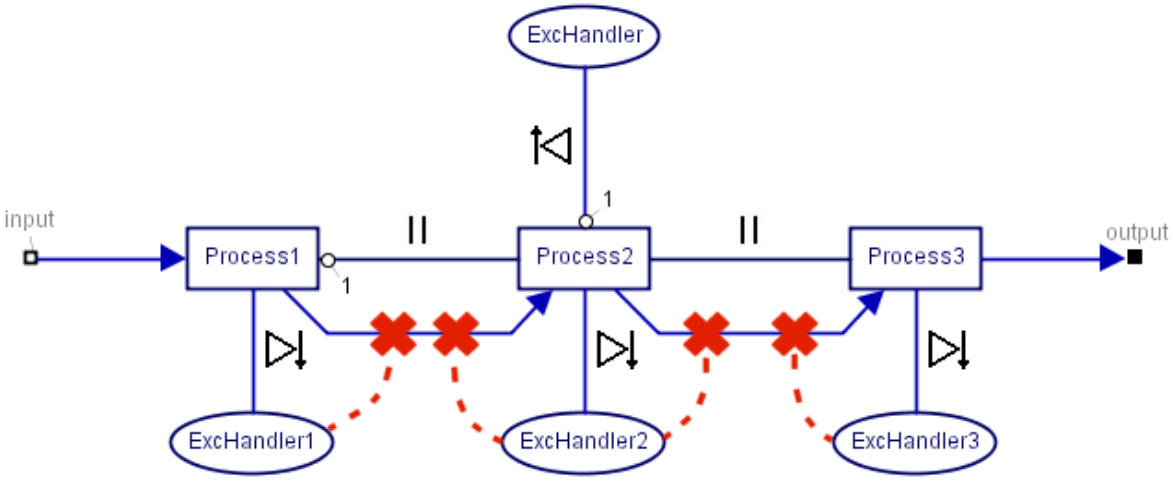


Figure 39. Stopping processes using channel rejecting

Example 4a: Division by Zero with two processes and channel rejection

This example is a variation to the example 1 and example 3.

In this example, the channel between ReadKeyboard and Calculate is a normal channel, so not one that is specifically written. If a zero was written to the channel by ReadKeyboard, the Calculate process reads this from the channel. When it tries to make the division, it anticipates division by zero and it throws a `DivisionByZeroException`. This is caught by the exception handler CalculationHandler. Upon receiving a `DivisionByZeroException`, CalculationHandler rejects the channel between ReadKeyboard and Calculate and then terminates. After rejecting the channel, the ReadKeyboard process will also terminate. The exception handler ReadingHandler checks the exceptions and removes any `ChannelRejectedExceptions`. If any exceptions remain, it throws the set higher up. This will however not occur in this example.

At this point, both processes have terminated and thus the parallel terminates.

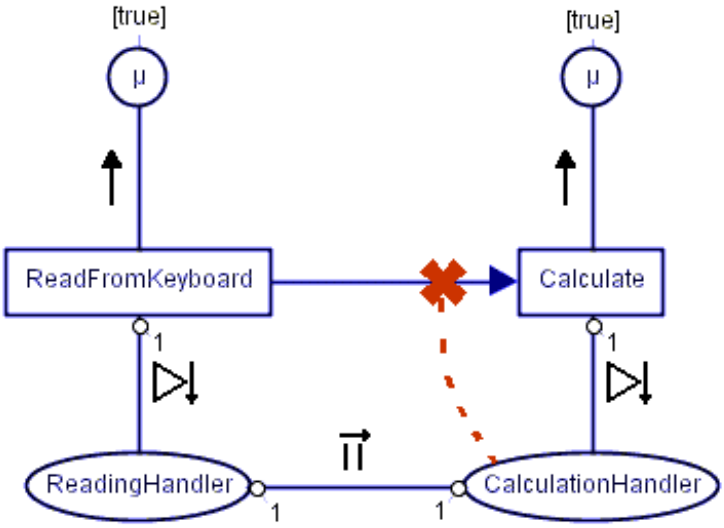


Figure 40. Division by zero example with channel rejection

Example 4b: JIWY axis with soft end stops and program termination

Mechanical systems usually have limited movement ranges. In the case of JIWY, there are mechanical end stops that prevent the rotational JIWY joints from turning too far and break the cables connected

to the motor or encoder. When JIWY hits the end stop, it is a very abrupt stop, which could eventually lead to damage to the setup. A way to prevent this is by programming software end stops. A software end stop takes action when JIWY gets too close to a hardware end stop and prevent the setup from hitting the hardware end stop.

Although more gentle solutions can be used to prevent the hitting of the end stop, it is specified in this example that reaching a software end stop is such a serious event that the complete program should terminate. An example of a more gentle solution is the addition of an extra process between the output of the controllers and the real output that would prevent the setup from hitting the mechanical end stop.

Because in this example, the complete program has to terminate, it means that not only the axis that hits the end stop should terminate, but also the axis that is functioning properly. Therefore, it is not an example as in section 4.4.2, in which the axes are completely uncoupled, but a problem of exception handling in a parallel composition.

Figure 41, Figure 42 and Figure 43 give the GML model of the new JIWY software used in this example.

The motor steering values from the controllers go via channels to the Safety process. The Safety process reads the calculated steering values from these channels, processes these values and then sends them to the (external) motor channels. An example of this processing can be a limitation of the steering values to prevent damage to the motors. Exception handling could be used here as well.

In order to guard the system against exceeding the allowed rotational ranges, processes jiwy_horizontal and jiwy_vertical are changed to incorporate `THROW` statements. Every time new encoder values are read, they are compared to the virtual positions of the software end stops. If an axis gets to close to the mechanical end stop, i.e. exceeds the software end stop, a `SoftEndStopException` is thrown.

When an exception is thrown, it is caught by the appropriate handler. The handler rejects the steering channel that runs to the Safety process and throws the received `ExceptionSet` higher up. Then Safety process gets terminated because of the rejected channel and SafetyHandler is executed. SafetyHandler rejects the two steering channels coming into the Safety process, thereby making sure that the other axis will also terminate.

The exception handler of the other axis will filter out the `ChannelRejectedException` and if its `ExceptionSet` is not empty throw it higher up.

At this moment, the three child processes of the parallel construct have terminated and the parallel construct will throw the collection of exceptions, which will eventually reach the GlobalHandler.

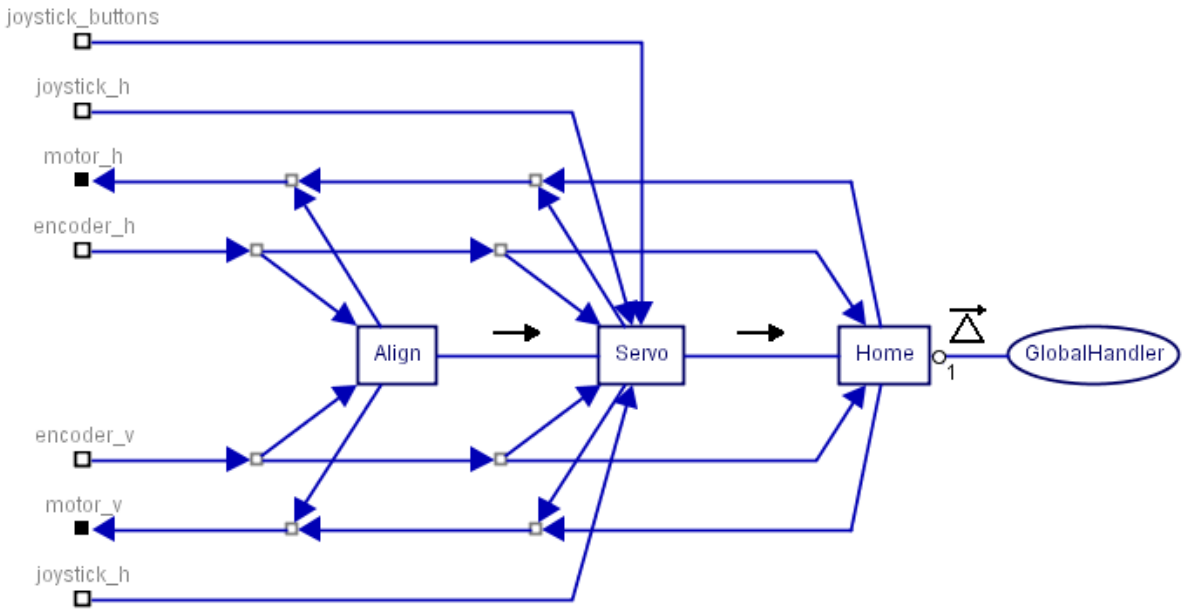


Figure 41. Top level model of JIWY software

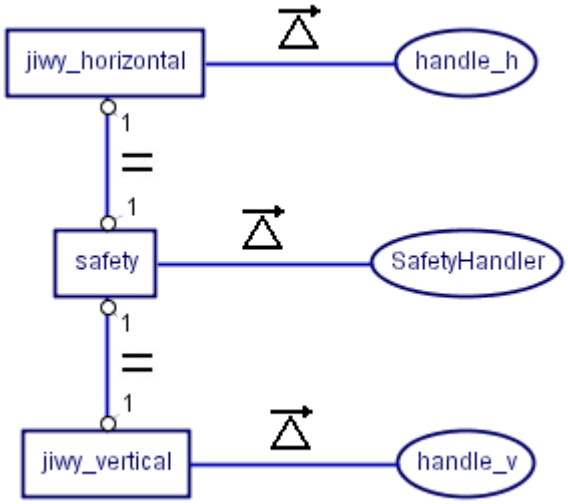


Figure 42. Compositional model of the Servo process

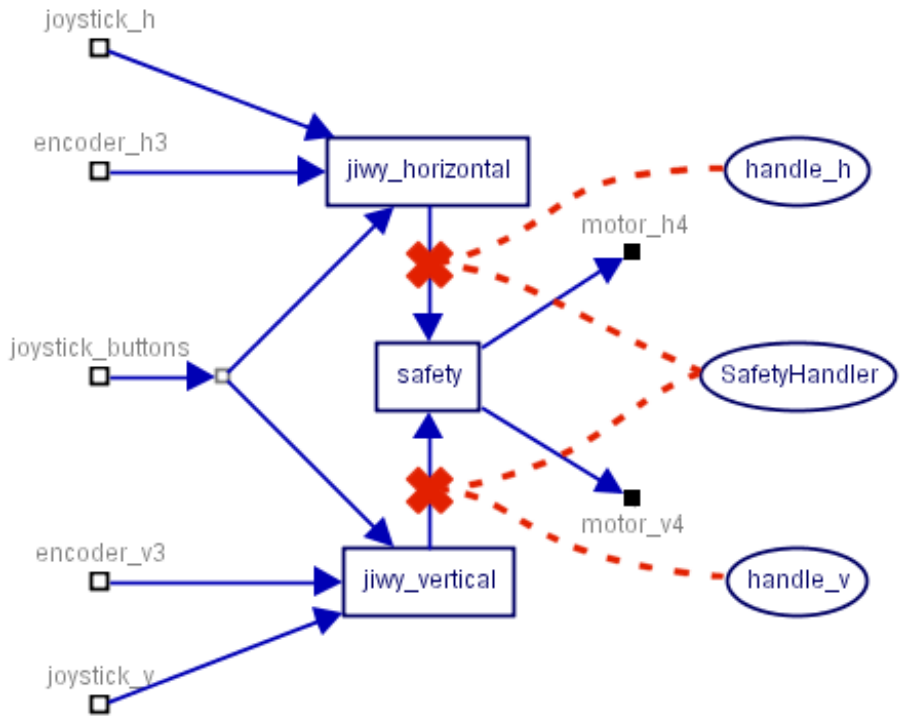


Figure 43. Communications model of the Servo process

Example 4c: comstime with channel rejection to terminate

Comstime is a program that is often used to benchmark languages or libraries that are based on CSP concepts. Figure 44 shows the standard GML model of the program.

The Prefix process starts with writing a zero on its output channel. The Delta process sends the values it receives from the Prefix process to the TimeMeasurement and Increment processes. The Increment process increments the value it receives by one and sends the incremented value to the Prefix process, which after outputting a first zero just sends out what it receives.

The TimeMeasurement process determines the time it takes to go round one loop of the program. It does so by measuring the time between the reading of the 1st and for example the 10.000st value.

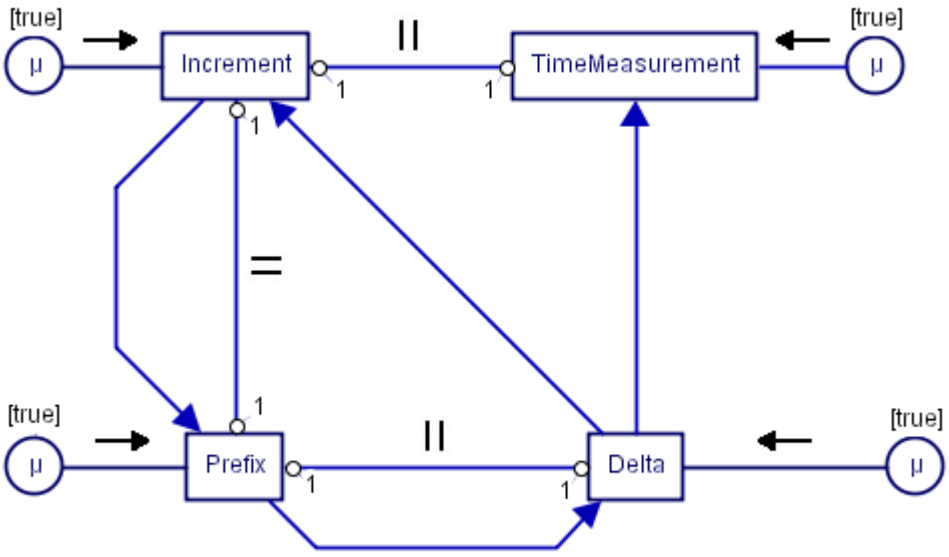


Figure 44. Comstime model (without termination)

The comstime program in Figure 44 never terminates. This behavior is generally not desired. This comstime program could under DOS and Linux be stopped by pressing CTRL-C, but in many other programs this is not an option. Therefore, a change has to be made to make the complete program stop.

A μ primitive repetition process with a `true` guard indicates an infinite loop. In order to stop the program, the infinite loops should be changed, such that they all stop at the same time. This can be done with conditional repetition where the guards are either checking a global variable (instead of being constant `true`) or local counters with the same limit for all loops. One should take care that there are no processes blocked on channels at the moment a loop stops. Otherwise, the other loops will not reach their stopping condition, what is equivalent to deadlock. That makes this way of synchronizing repetitive loops hard to reason about.

Using exception handling together with channel rejection gives an alternative. Figure 45 and Figure 46 show the compositional model and communication model for this solution.

In this solution, the TimeMeasurement determines when the program should terminate. It does so by rejecting the channel to the Delta process before terminating. Because of the channel rejection, Delta will terminate by throwing an exception. If the exception handler of the Delta process also rejects all the channels the Delta process communicates with, the Prefix and the Increment processes will also terminate. When that happens, all processes are terminated and the program terminates.

Although only the Delta really needs an exception handler to propagate the terminating sequence of channel rejections, in this solution every process is given an exception handler. These exception handlers reject the channels connected to the processes they guard. Which channels should be rejected is given to the constructor of the exception handler. There is no direct way the exception handler can determine this by itself.

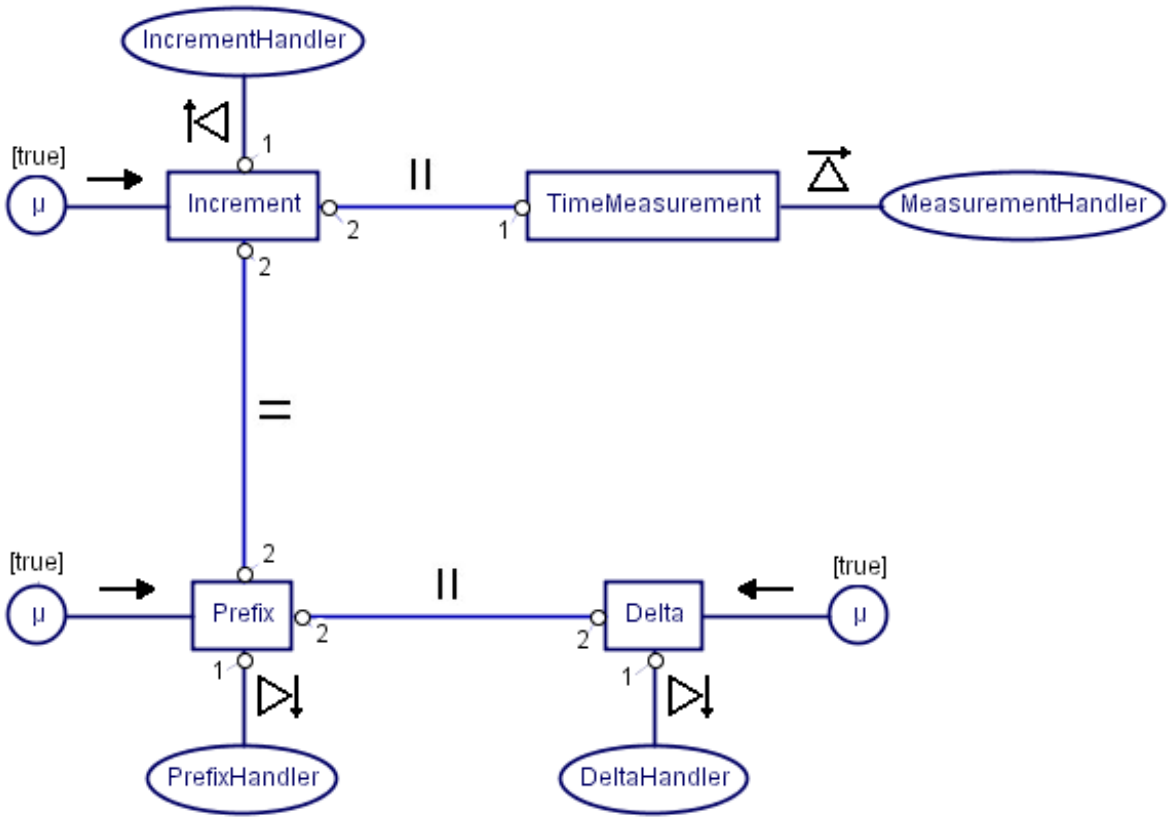


Figure 45. Compositional model of comstime with termination using channel rejection

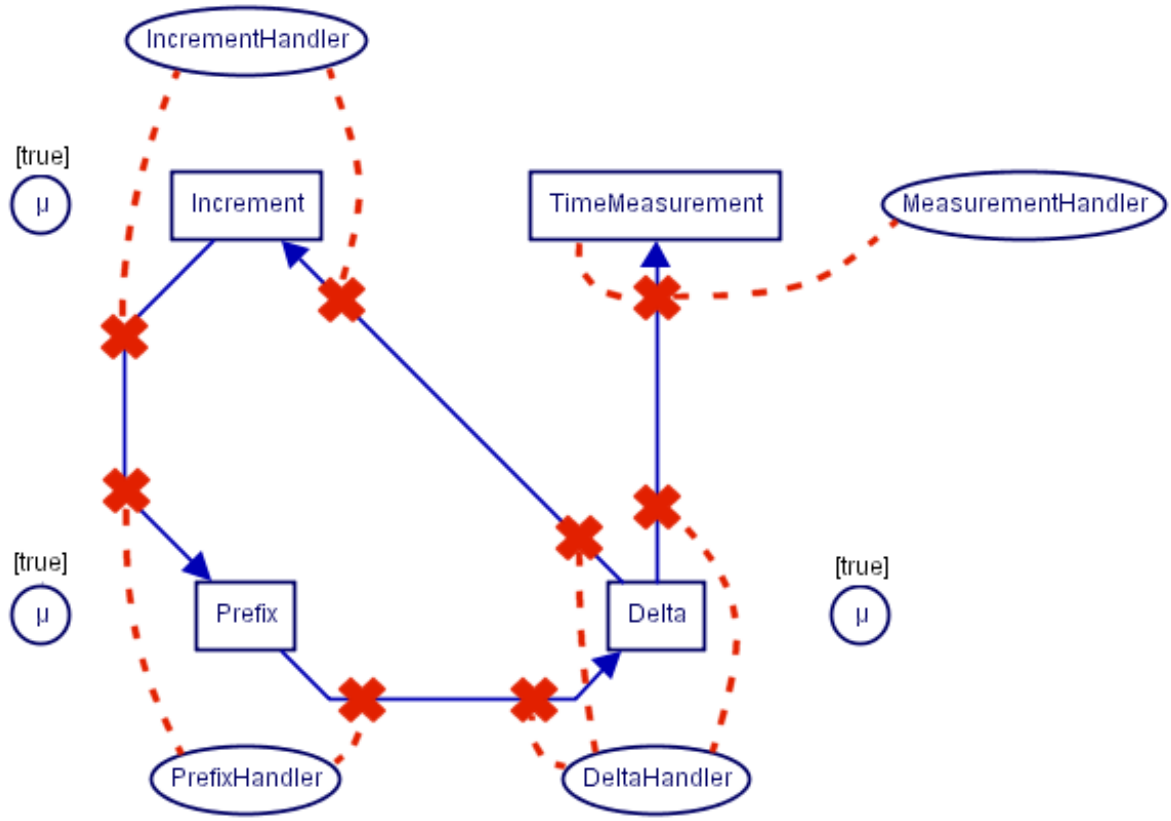


Figure 46. Communication model of comstime with termination using channel rejection

```
ChannelRejectingHandler::ChannelRejectingHandler(ChannelBase **channels,
int size) {
  for (unsigned int i = 0; i < size; i++) {
    this->channels[i] = channels[i];
  }
}

void ChannelRejectingHandler::run(ExceptionSet *eSet) {
  Exception *exc = eSet->First();
  while (exc) {
    if (exc->type >= "ChannelRejectedException") {
      //Do things here that are specific for this type
      //of exception.
      exc = eSet->isHandled();
    } else {
      //Place code here for when the exception type
      //is unknown.
      exc = eSet->Next();
    }

    //Do general things here
    for (unsigned int i = 0; i < nrOfChannels; i++) {
      channels[i]->reject();
    }

    //Check if the remaining set is empty
    if (!eSet->isEmpty()) {
      THROW(eSet);
    } else {
      delete eSet;
    }
  }
}
```

Listing 17. Constructor and run method of the comstime exception handlers

Listing 17 shows the run method of the exception handlers. They do not handle specific types of exceptions, but only reject all the given channels. This way, all the processes will eventually terminate, either normally or because of a channel rejection.

5 Conclusions and recommendations

5.1 Conclusions

CTC++ under RTAI

- It is now possible to compile programs using the CT library that are able to run under RTAI, a real time Linux variant. The are 2 problems when doing this:
    - Functions that are in a shared library cannot be used. This means many functions that C/C++ programmers are accustomed to like `printf` or `malloc` cannot be used. The Linux kernel and RTAI do provide some alternatives.
    - It is impossible to include Linux kernel header files. This can be solved by copying the variable or function prototypes into a header file that can be included in C++.
- It is possible to port existing CTC++ for DOS based programs to a CTC++ for RTAI based version. As an example, a program that controls JIWY was ported from its DOS version to a RTAI version. For this, changes had to be made to the interface card library. The RTAI version of the program shows better behavior then the DOS version if it is run in a DOS box under windows.

Exception handling in CT

- CTC++ is enriched with an exception handling mechanism, the main function used for fault tolerance. Together with channel rejection, it offers a powerful tool for termination of parts of the program.
- The *EXCeption* construct is supported by the GML tool (as can be seen in all the figures given in this report).
- User can define its own type of exception. The type of the class can be determined without using C++ RTTI.
- Although an implementation is possible under RTAI, it can lead to memory leaks.

5.2 Recommendations

Making it possible for CTC++ programs to run under LXRT.

LXRT is an extension to RTAI that makes it possible to run real time programs in user space. This means that dynamic libraries can be used, although when a System call is made into the Linux kernel (not the RTAI kernel), the program jumps back from hard real time (RTAI) to soft real time (Linux with raised priority). One problem when using LXRT would be IO interfacing, especially interrupt servicing.

Investigate the possibility for immediate termination of the (Pri)Parallel construct

This would solve the problem of the behaviour in a concurrent environment. The number of exception handlers, that are needed to reject all the channels, could be greatly reduced. The problem with implementing this, is that when it process is waiting for an event (channel communication for example), there should be a method to let the process retreat from the event in an orderly fashion. In a normal channel, there is a counter that counts the number of waiting readers. If a reading process that is blocked should terminate because of an exception in a parallel process, the counter in the channel should be decreased with one.

Find a CSPm expression for the exception handling mechanism

When the implemented exception handling mechanism can be expressed in CSPm, design using this mechanism can be checked for correctness. Until then, a very large pro for CSP based designs, the possible checking for correct behaviour, is not possible for design using the exception handling mechanism.

Use the exception handling mechanism on a more extensive mechanical setup

For a large mechanical set, a safety scheme was developed (Eglence, 2003). It would be useful to try to implement that safety scheme using the CT library and the developed exception handling mechanism.

Have more support for exception handling in gCSP.

Two features would be very helpful if they would be incorporated into the gCSP tool:

- Generating frameworks for user-defined exceptions. This way, the user-defined exception is always properly derived and it will always contain the necessary `ExceptionType` variable.
- Option to specify the type of exceptions an exception handler can handle. The tool can then generate code in accordance with the exception handler template given in section 3.4.8.

Appendix A Compiling a program with CTC++ for RTAI

Below, a bash script is given that can be used to compile CTC++ programs to run under RTAI.

```
#!/bin/bash
#this script assumes that there is a tmp directory

#fill in the variables below
MODULE_NAME=
CTC_DIRECTORY=
CTCPP_DIRECTORY=
RTAI_INCLUDE_DIRECTORY=
RTAI_CPP_DIRECTORY=
LINUX_INCLUDE_DIRECTORY=

rm -f $MODULE_NAME
cd tmp
rm -f *.o

#fill in the necessary options below. Recommendation: -Wall -O2
OPTIONS=''

#these options are fixed!
OPTIONS=$OPTIONS’ –c’
OPTIONS=$OPTIONS' -fno-rtti -fomit-frame-pointer -fno-exceptions'
OPTIONS=$OPTIONS' -fno-common -fno-strict-aliasing -fno-strength-reduce'
OPTIONS=$OPTIONS' -DGPP -DCSP -DRTAI'
OPTIONS=$OPTIONS' -D__KERNEL__ -DMODULE'


INCLUDES=' –I’$LINUX_INCLUDE_DIRECTORY
INCLUDES=$INCLUDES' –I’$RTAI_INCLUDE_DIRECTORY
INCLUDES=$INCLUDES' –I’ RTAI_CPP_DIRECTORY
INCLUDES=$INCLUDES' -I'$CTCPP_DIRECTORY
INCLUDES=$INCLUDES' -I'$CTC_DIRECTORY

#fill in the paths for the include files
INCLUDES=$INCLUDES''

#fill in the files that need to be compiled
FILES=''
FILES=$FILES''

#uncomment the next line if static libraries other than the
#CT library should be used and fill in the library directory after the -L
#LIBRARY_DIR=’ –L’

#uncomment the next line if static libraries other than the
#CT library should be used and fill in the library name (without lib)
#after the -l
LIBRARY=’ -l’

g++ $OPTIONS $INCLUDES $FILES

ld -m elf_i386 -Ur -o ../$MODULE_NAME $RTAI_CPP_DIRECTORY/crtsbegin.o *.o
$RTAI_CPP_DIRECTORY/crtsend.o -L$CTCPP_DIRECTORY/lib -lcsp $LIBRARY_DIR
$LIBRARY

cd ..
```

Appendix B Code for the four options of type determination

Appendix B.1 Option 1

```
class Exception
{
  private:
    const char *type_names[100];
    int type_counter;

  protected:
    void SetType(const char * string) {
      if (type_counter < 100) {
        type_names[type_counter++] = string;
      }
    };

  public:
    Exception() {
      type_counter = 0;
      SetType("Exception");
    };

    bool isOfType(const char *string) const {
      return !((bool) strcmp(string,type_names[type_counter-1]));
    }

    bool isDerivedFrom(const char *string) const {
      int i;
      for (i = type_counter; i > 0; i--) {
        if (!(strcmp(string, type_names[i-1]))) {
          return true;
        }
      }
      return false;
    };

    const char * Type() const {
      return type_names[type_counter-1];
    }
};

class RejectedException : public Exception
{
  public:
    RejectedException() {
      SetType("RejectedException");
    }
};

class EncoderRejectedException : public RejectedException
{
  public:
    EncoderRejectedException() {
      SetType("EncoderRejectedException");
    }
};
```

Appendix B.2 Option 2

```
class ExceptionType;
class BaseException {
  private:
    ExceptionType *top_ExceptionType;
    friend ExceptionType;

  public:
    BaseException() {
      top_ExceptionType = 0;
    }
};

class ExceptionType {
  private:
    const char *type_name;
    ExceptionType *prev_type;
    BaseException *be;

  public:
    ExceptionType(const char *string, BaseException *be) {
      prev_type = be->top_ExceptionType;
      be->top_ExceptionType = this;
      this->be = be;
      type_name = string;
    }

    bool isOfType(const char *string) {
      return !((bool) strcmp(string,be->top_ExceptionType
        ->type_name));
    };

    bool isDerivedFrom(const char *string) {
      ExceptionType *exception_type = be->top_ExceptionType;
      while (exception_type) {
        if (!strcmp(string, exception_type->type_name))
          return true;
        exception_type = exception_type->prev_type;
      };
      return false;
    };
};

class Exception : public BaseException {
  public:
    ExceptionType type;
    Exception() : type("Exception",this) {
    }
};

class RejectedException : public Exception {
  public:
    ExceptionType type;
    RejectedException() : type("RejectedException", this) {
    }
};
```

```
class EncoderRejectedException : public RejectedException {
  public:
    ExceptionType type;
    EncoderRejectedException() : type("EncoderRejectedException",this) {
    }
};
```

Appendix B.3 Option 3

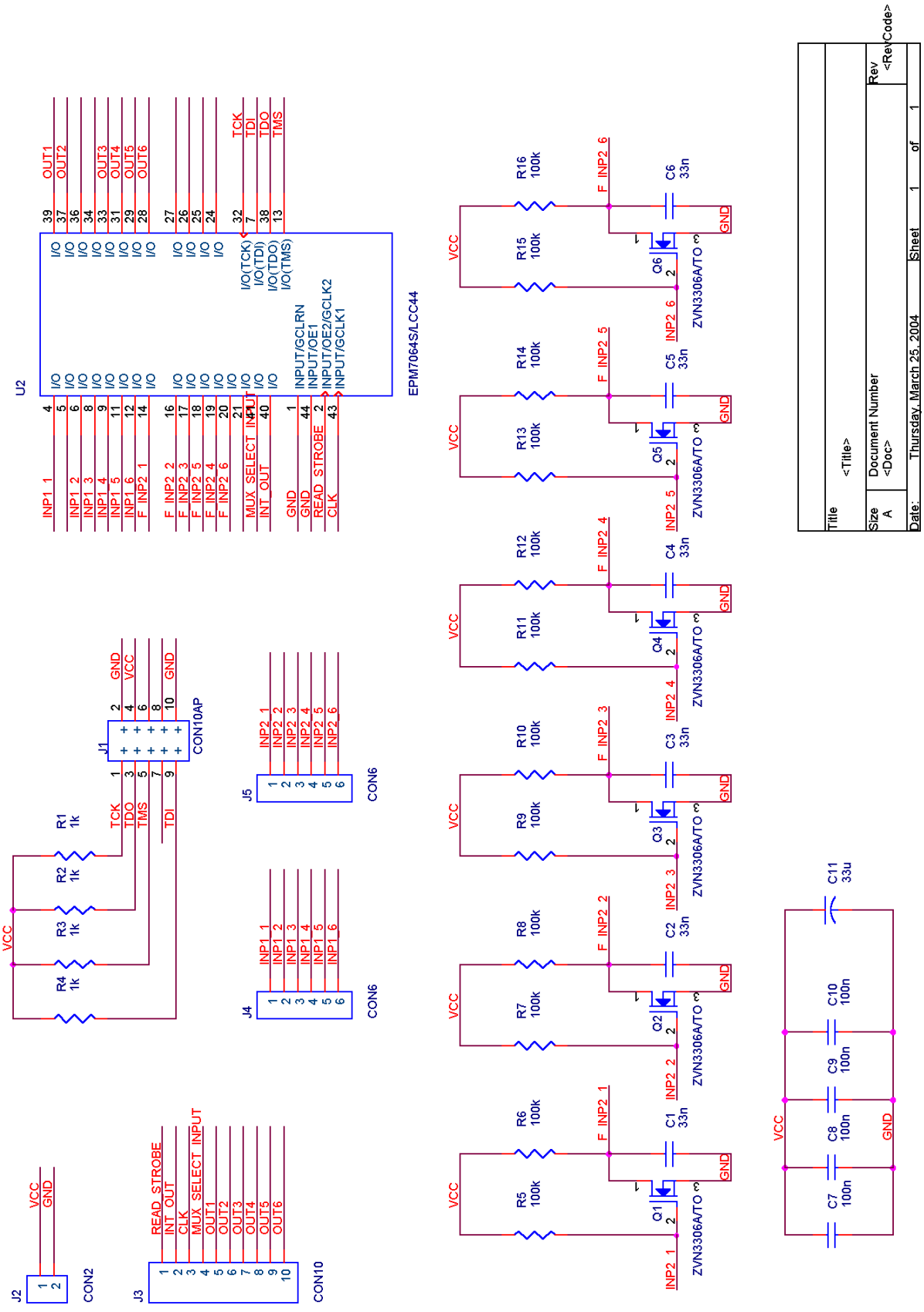
```
class ExceptionType {
  private:
    ExceptionType *prev;
    friend class Exception;

  public:
    ExceptionType() {
      prev = 0;
    };

    ExceptionType(ExceptionType &type) {
      prev = &type;
    }
};

class Exception
{
  protected:
    ExceptionType *my_type;

  public:
    static ExceptionType type;
    Exception() {
      my_type = &type;
    }

    bool isOfType(ExceptionType &type) {
      return (my_type == &type);
    }

    bool isDerivedFrom(ExceptionType &type) {
      ExceptionType *temp = my_type;
      while (temp) {
        if (temp == &type)
          return true;
        temp = temp->prev;
      }
      return false;
    }
};
ExceptionType Exception::type;

class RejectedException : public Exception {
  public:
    static ExceptionType type;
    RejectedException() {
      my_type = &type;
    }
};
ExceptionType RejectedException::type(Exception::type);

class EncoderRejectedException : public RejectedException {
  public:
    static ExceptionType type;
    EncoderRejectedException() {
      my_type = &type;
    }
};
ExceptionType EncoderRejectedException::type(RejectedException::type);
```

Appendix B.4 Option 4

```
class Exception {
  public:
    Exception() {
    }

    virtual bool isOfType(const char *string) {
      if (!strcmp(string, "Exception")) {
        return true;
      }
      return false;
    }
};

class RejectedException : public Exception {
  public:
    RejectedException() {
    }

    virtual bool isOfType(const char *string) {
      if (!strcmp(string,"RejectedException")) {
        return true;
      }
      return Exception::isOfType(string);
    }
};

class EncoderRejectedException : public RejectedException {
  public:
    EncoderRejectedException() {
    }

    virtual bool isOfType(const char *string) {
      if (!strcmp(string,"EncoderRejectedException")) {
        return true;
      }
      return RejectedException::isOfType(string);
    }
};
```

Appendix C Digital input print for JIWY

Appendix C.1 Schematic



Appendix C.2 AHDL code of Altera chip

```
TITLE "jiwy_digital_inputs";
INCLUDE "lpm_mux.inc";

PARAMETERS
(
  INPUTS_PER_GROUP = 6
);

SUBDESIGN jiwy_digital_inputs
(
  CLK                                                                  : INPUT =
GND;
  INP[2..1][INPUTS_PER_GROUP..1]                 : INPUT = GND;
  READ_STROBE                                                                  :
INPUT = VCC;
  MUX_SELECT_INPUT                                                             :
INPUT = GND;
  OUT[INPUTS_PER_GROUP..1]                                        : OUTPUT;
  INT_OUT                                                              : OUTPUT;
)

VARIABLE
  INPUT_FFS[6..1][INPUTS_PER_GROUP..1]                             : DFF;
  READ_LATCH[2..1][INPUTS_PER_GROUP..1]                            : DFF;
  OUTPUT_MUX
       : lpm_mux WITH (LPM_WIDTH=INPUTS_PER_GROUP, LPM_SIZE=2,
LPM_WIDTHS=1);
  FILTERED_INPUTS[2..1][INPUTS_PER_GROUP..1]           : NODE;

BEGIN

  INPUT_FFS[][].CLK = CLK;
  INPUT_FFS[6..5][].D = INP[][];
  INPUT_FFS[4..3][].D = INPUT_FFS[6..5][].Q;
  INPUT_FFS[2..1][].D = INPUT_FFS[4..3][].Q;

  FOR GROUP IN 1 TO 2 GENERATE
    FOR INPUT_NR IN 1 TO INPUTS_PER_GROUP GENERATE
      TABLE
        INPUT_FFS[GROUP+0][INPUT_NR], INPUT_FFS[GROUP+2][INPUT_NR],
INPUT_FFS[GROUP+4][INPUT_NR]  =>  FILTERED_INPUTS[GROUP][INPUT_NR];
        0                                                    , 0
                  , 0
           => 0;
        0                                                    , 0
                  , 1
           => 0;
        0                                                    , 1
                  , 0
           => 0;
        0                                                    , 1
                  , 1
           => 1;
        1                                                    , 0
                  , 0
           => 0;
        1                                                    , 0
                  , 1
           => 1;
        1                                                    , 1
                  , 0
           => 1;
```

```
      1                                                    , 1
                  , 1
          => 1;
     END TABLE;
   END GENERATE;
  END GENERATE;


  READ_LATCH[][].D = FILTERED_INPUTS[][];
  READ_LATCH[][].CLK = !READ_STROBE;

  OUTPUT_MUX.data[][] = READ_LATCH[][].Q;
  OUTPUT_MUX.sel[] = MUX_SELECT_INPUT;

  OUT[] = OUTPUT_MUX.result[];

  IF !READ_STROBE THEN
    INT_OUT = GND;
  ELSE
    INT_OUT = !(FILTERED_INPUTS[][]==READ_LATCH[][].Q);
  END IF;

END;
```

Appendix C.3 Usage

The following piece of code is an example of how to use and initialize the encoder channels and the object needed for the digital input print.

```
//allocate a new DAQSTC object
  DAQSTC *daqstc = new DAQSTC();
  ICM *icm;
  ISHandler *ishandler;
  DAC *dacX, *dacY;
  GPC *encoderX, *encoderY;
  int irqnr;

  unsigned char old_mask, new_mask;
  unsigned int pic_port, irq_offset;

//Initialize the DAQ-STC. This will start the PCI-initialization
  daqstc->Initialise();
//Request the IRQ number of the card
  irqnr = daqstc->GetInterruptNr();
  if (irqnr == -1) {
    return(-1);
  }

//Determine the appropriate port and bits on the PIC chip
  if (irqnr < 8) {
    pic_port = 0x21;
    irq_offset = 0;
  } else {
    pic_port = 0xA1;
    irq_offset = 8;
  }

//--- Interrupt Control
//Allocate a new Interrupt Control Manager object
  icm = new ICM();
//Install the allocated ICM as interrupt handler
  ishandler = Processor__registerInterruptService(irqnr, icm, 1);

//--- Counters
  DigitalInputDeviceDriver *dig_dd = new DigitalInputDeviceDriver();
  encoderX = new NewEncoder(GPC::Counter0, *dig_dd);
  encoderY = new NewEncoder(GPC::Counter1, *dig_dd);

//Register the DigitalInputDeviceDriver as callback for the ICM
  icm->registerCallback(*dig_dd,ICM::G0_Gate,true);

//Allocate the encoder channels
  encoderX->Initialize(GPC::Counter0,GPC::RelativePositionSensing);
  encoderY->Initialize(GPC::Counter1,GPC::RelativePositionSensing);
  encoderX->Reset();
  encoderY->Reset();
  encoderX->SetMode(GPC::RelativePositionSensing);
  encoderY->SetMode(GPC::RelativePositionSensing);
  encoderX->Arm();
  encoderY->Arm();
```

```
//Enable the interrupt on the PIC chip
  old_mask = inportb(pic_port);
  new_mask = old_mask & ~(1 << (irqnr - irq_offset));
  printf("old_mask = %X, new_mask = %X\n",old_mask, new_mask);
  outportb(pic_port, new_mask);

//Enable the clock to the digital input print
  MISC::MISC_FOUT_Configure(MISC::SLOW, 16);
  MISC::MISC_FOUT_Enable();
//Setup the gate input to be an interrupt source
  GPC::DigInterruptSetup();
```

References

Broenink, J.F., D. Jovanovic and G.H. Hilderink (2002), Controlling a mechanic setup using Real-time Linux and CTC++, *Proc. Mechatronics 2002*, Enschede, S. Stramigioli (Ed.), pp. 1323-1331, ISBN: 90-365-17664.

Brown, N.C.C. and P.H. Welch (2003), An Introduction to the Kent C++CSP Library, *Proc. Communicating Process Architectures 2003*, 7-10 September 2003, Enschede, Netherlands, G. H. Hilderink (Ed.), pp. 139-156, ISBN: 1 58603 381 6.

Buhr, P.A. and W.Y.R. Mok (2000), Advanced Exception Handling Mechanisms, *IEEE trans. Software Eng.*, **26,** (9), pp. 820-836, ISSN:

Burns, A. and A. Wellings (2001), *Real-TIme Systems and Programming Languages*, Pearson Education, 0201729881.

Cristian, F. (1995), Exception Handling and Tolerance of Software Faults,*in: Software Fault Tolerance*, **3**, M. R. Lyu, John Wiley & Sons Ltd., Chichester, 81-107.

Eglence, M. (2003), *Design and realization of a safe control system for a parallel manipulator*, MSc thesis 010CE2003, Control Laboratory, University of TWente, Enschede.

FormalSystems (2004), *FDR2 Refinement checker for CSP models*, http://www.fsel.com.

Foundation, F.S. (1989), The GNU General Public License (GPL), pp. ISSN:

Hilderink, G.H. (2002), A graphical Specification Language for Modeling Concurrency based on CSP, *Proc. Communicating Process Architectures 2002*, 15-18 Sep 2002, Reading UK, P. W. James Pascoe, Roger Loader, Vaidy Sunderam (Ed.), pp. 255-284, ISBN: 1-58603-268-2.

Hilderink, G.H. and J.F. Broenink (2003), Sampling and timing a task for the environmental process, *Proc. Communicating Process Architectures 2003*, Enschede, Netherlands, G. H. Hilderink (Ed.), pp. 111-124, ISBN: 1 58603 381 6.

Hilderink, G.H., J.F. Broenink, W.A. Vervoort and A.W.P. Bakkers (1997), Communicating Java Threads, *Proc. Proc. WoTUG-20 on Parallel programming and Java*, Enschede, Netherlands, (Ed.), pp. 48-76, ISBN: 1383-7575.

Hilderink, G.H., D.S. Jovanovic and J.F. Broenink (2003), A multimodal robotic control law modelled and implemented with the CSP/CT framework, *Proc. Communicating Process Architectures 2003*, 7 - 10 September 2003, Enschede, Netherlands, G. H. Hilderink (Ed.), pp. 315-334, ISBN: 1 58603 381 6.

Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice Hall, 0-13-153271-5 (0-13-153289-8 PBK).

Jovanovic, D., G.H. Hilderink and J.F. Broenink (2002), A communicating Threads -CT- case study: JIWY, *Proc. Communicating Process Architectures 2002*, 15-18 Sep 2002, Reading UK, V. Sunderam (Ed.), pp. 311-320, ISBN: 1-58603-268-2.

Jovanovic, D.S., B. Orlic, G.K. Liet and J.F. Broenink (2004), gCSP: A Graphical Tool for Designing CSP Systems, *Proc. Communicating Process Architectures 2004*, 5-8 Sep 2004, Oxford, UK, M. Green (Ed.), pp. 233-251, ISBN:

LinuxOnline (1994), *The Linux Home Page at Linux Online*, http://www.linux.org.

Ros, M. (2004), *JIWY with a camera*, Individual Design Report, Control Laboratory, University of Twente, Enschede.

RTAI (2004), *DIAPM RTAI - RealTime Application Interface*, http://www.rtai.org.

Stephan, R.A. (2002), *Real-time Linux in Control Applications Area*, MSc thesis 016CE2002, Dept. Electrical Engineering, University of Twente, Enschede.

Welch, P.H. and D.C. Wood (1996), The Kent Retargettable occam Compiler, *Proc. Parallel Processing Developments, WoTUG-19*, Nottingham, United Kingdom, B. C. O'Neill (Ed.), pp. 143-166, ISBN: 9051992610.