National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

Synapse : A Real-Time Programming Language

Michel de Champlain

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

September 1989

© Michel de Champlain, 1989

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-51351-9

Canada

Abstract

Synapse : A Real-Time Programming Language

Michel de Champlain

This thesis defines the real-time programming language Synapse. The language main goal is to be small and expressive, so each of the primitives has simple and well-defined semantics. Small embedded real-time microprocessor systems and device drivers are its main application area. The concurrent programming model proposed by Synapse is asynchronous message passing via interrupts. Communication between tasks is effect by sending interrupt messages to handlers. This inter-task communication concept is in fact a natural extension of the hardware interrupt mechanism. Synapse makes both hardware and software interrupts uniformally available in high level language facilities. Synapse provides dynamic priority and time slice options, along with task identifiers, to allow the control of scheduling and dispatching decisions. It supports exception handlers, to handle exceptions caused by hardware or software. Synapse has the ability to specify and manipulate device registers for memory or port mapped devices to support both types of processor architectures.

To my wife Hélène
for her patience, help,
and, above all, love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose

Many real-time programs must satisfy strict timing demands. These demands should be more than just the correctness of concurrent programs. Traditionally, the approach to real-time programming has been to write a concurrent program. The purpose of concurrent programming is to support the sequencing of operations except the usage of synchronization statements [Glig83][Kung85]. By contrast, in real-time programming the programmer must be able to control the sequencing of operations explicitly[Lee84][Lee85].

This thesis introduces the real-time programming language Synapse. The language supports the construction of real-time applications that will satisfy strict timing constraints. Such applications require to respond to external interrupts within a fixed time deadline.

1.2 Main Goal

The main goal of Synapse is to bring high-level language programming to real-time applications. Synapse is small and expressive, so each of the primitives has a simple, well-defined semantics. This goal supports the following requirements in the language:

- General Goals in Programming Language Design
- Task Control Management
- Timing Constraint Specifications
- Hardware Environment Specifications

1.3 Organization of the Thesis

Chapter 2 introduces the goals in programming language design. Chapter 3 presents the requirements in language design for real-time programming. Chapter 4 summarizes the suitability of several existing languages by taking the results of recent surveys in requirements for real-time programming languages. Chapter 5 contains the complete Synapse language report. Chapter 6 describes some examples of application programs. Chapter 7 concludes with the experience, the state of the implementation, final remarks, and future directions. Appendix A is a summary of the grammar given throughout the report.

Chapter 2

Goals in Programming Language Design

This chapter presents several goals for programming language design. It states also some basic rules for making trade-offs among competing goals. Note that the presentation order does not reflected their relative importance. The language report, in chapter 5, explains how Synapse meets these goals.

2.1 Roots of Basic Rules, Methods and Principles

Several designers of successful programming language have made considerable efforts to define detailed rules, methods and principles to produce a better language [Ichb79, Brin82, Wirt85, Holt88]. These reports have led to one important fact to remember: "Language design is extremely complex and no single rule leads assuredly to good decisions" [Holt88]. The design of a language represents a trade-off among inevitably competing goals and reflects the skills, knowledge, and tastes of its designers [Pope77]. Language design is a product of human creativity. To serve some needs perceived, each creator conceives it in good faith with a good measure of common sense [Feld86]. Therefore, intel-

ligent debate and criticism are necessary to gain a clearer perspective on a programming tradition. This inheritance has become so natural that it inhibits the ability to search for new methods [Brin82]. Based on those principles, language designers have reached common agreement on a number of important motherhood goals for general purpose programming language design [Trem85], such as:

- Good Human Interface
- Simplicity
- Minimality
- Ease of Use
- Level of Abstraction
- Modularity and Separate Compilation
- Portability
- Expressiveness
- Orthogonality
- Error Handling
- Efficiency

2.2 Good Human Interface (Readability and Writability)

External documentation and comments cannot clarify all the complexity involved in a program. Notably since external documentation is too often out of date and incomplete. Also, programmers dislike writing excessive comments and avoid them. Given these facts, the most reliable form of documentation should be the program itself. A better understanding of a program should also occur whenever the syntax reflects the semantics. We must be able to read and write programs easily, but readability of programs is far more important for maintenance than their writability [Hoar73].

2.3 Simplicity

Simplicity is a major goal in language design: language constructs must be simple to understand in all situations. It is extremely difficult to decide to select or exclude all kinds of features "it might also be nice to have." Wirth states that language designers often fail in this respect: "I gladly admit that certain features of Ada that have no counter-parts in Modula-2 may be nice to have occasionally. Nevertheless, I question whether they are worth the price. Ada compilers are gigantic programs consisting of several hundred thousand lines of code, because our newest Modula compiler measures some five thousand lines only. I confess secretly that this Modula compiler is already at the limits of comprehensive complexity ... the real cost with those huge compilers is hidden in the unseen efforts of the innumerable programmers trying desperately to understand them and use them effectively." [Wirt85]

Always rated the benefit of a language feature against the added cost of its implementation avoids large and complex compilers.

2.4 Minimality

Minimality is the property of having the absolute minimum number of constructs with which a language can possibly survive (the language Edison is an example). There is a significant distinction between a minimal set of constructs and a minimal usable set [Trem85]. As an example, sequential and While constructs are enough to simulate all other constructs (even the If), but this minimal set is not really usable.

2.5 Ease of Use

All the constructs of a language must be easy to learn and natural to remember. As a sign of this, programmers familiar with the language should not consult their manuals constantly. The correct method of doing something will be clear if the language is simple and straightforward. Too many different ways of doing the same piece of code leaves the programmer struggling to decide which one is appropriate for his application. The programmer appreciates flexibility in a language when it remains easy to learn and use. However, not when the flexibility needlessly increases the language's complexity.

2.6 Level of Abstraction

Strong typing and type-checking are central concepts in modern programming languages (Pascal promotes them). Strong typing can improve the reliability and readability of programs. An often too simplistic approach regards the definition of a type as a set of values. A higher-level approach to data abstraction defines an abstract data-type (ADT) as structured data together with a set of operations. These operations are procedures, functions, or tasks. An ADT encapsulates and hides the representation of the data and the implementation of the operations. The user's view perceives an ADT as a "black box." Several languages use modules or packages for the specification of abstract data types as well as for the separate construction of program parts.

2.7 Modularity and Separate Compilation

A language must be modular, and hence provide control over the visibility of names. In large projects there is a need to protect programmers from the accidental misuse of implementation details. Because, this habit can lead to programs that are difficult to change because a change in one module may affect other modules. Encapsulation and scope are two facilities hiding implementation details from users of a given module. They extend the notion of information hiding, already at the core of the notion of abstract data-type. Encapsulation is a tool for structured program organization that groups logically related entities. Encapsulation constructs increase program security and simplify separate compilation. Scope rules specify which entities are visible in a given area of a program. Open scopes (like subprograms) specify constructs with all external entities "implicitly imported." Other constructs should explicitly import or export entities like Modula-2 modules. The package (Ada) and the module (Modula-2) are distinct classes of encapsulation constructs [Appe82]. Both specify a single instance of an environment by encapsulating a collection of declarations.

2.8 Portability

One of the original hopes for high-level languages (HLLs) was that they would be machine-independent. A language is machine-independent if a program compiles and runs correctly on machine X, and produces exactly the same output given the same input on machine Y.

2.9 Expressiveness

Language expressiveness should ease the formation of a self-explanatory statement of the programmer's intentions. It is also a measure of how naturally a program structure expresses a problem-solving strategy. It must also provide facilities for hiding the low level (machine-dependent) processing from other parts of the program. These parts can operate at higher levels of abstraction, as well as adequate modularization mechanisms. This improves the portability. Clever and quick programming tricks are directly in conflict with readability. Their usage must be discouraged since they are mostly never essential.

2.10 Orthogonality

Any composition of basic features without any restrictions or special cases is orthogonality. However, the need for simplicity should exclude a general orthogonality feature that introduces complexity and heaviness. The tradeoff between the two goals is hard to quantify and is a major decision in language design. It is not always clear which is better:

- to combine few simple concepts without restrictions.
  Example: Algol68 is 99% orthogonal but not simple.
- to combine few simple concepts with some restrictions.
  Example: Pascal is simple but not orthogonal.

2.11 Error Handling

In some languages, array subscripts outside of their range and dangling pointers are usually examples of errors undiscovered at compile time. Run-time routines support error detection by set up simple and inexpensive run-time checks for undetected errors during compilation. Debugging phase installs these routines and removed them when done.

2.12 Efficiency

In the past, many languages have had efficiency as a main design goal (implicitly or explicitly). Efficiency must come after reliability in language design. More computing sins were committed in the name of efficiency than for any other single reason, including blind stupidity [Wulf72]. The issue of efficiency is no longer measured exclusively by execution speed and space. Today, the "efficiency measure" is the productivity and the maintenance cost of software development rather than only the computing performance of the resulting products.

Chapter 3

Requirements in Real-Time Programming Language Design

This chapter presents the requirements for real-time programming language design, in single processor systems.

3.1 Introduction

Originally, real-time programming languages (RTPLs) base their design on sequential general-purpose high-level programming languages, such as Pascal. It evolved specifically for implementing system software components. Concurrent system software forced those original RTPLs to use the system calls available from the target operating system. Unfortunately, programs developed that way were not portable and were hard to maintain. Because, they contained hard-coded operating system calls. Over the past few years, the applications on computers have changed considerably. Multitasking systems are now plentiful, requiring new RTPL design goals for the needs of system programmers. Studies

in the way RTPLs are used (and misused) have led to several requirements [Glig83] [Appe85] [Klig86] [ARTE87] listed below:

- Task Control Management
- Timing Constraint Specifications
- Hardware Environment Specifications

Since RTPLs are high-level languages, RTPL design goals try also to include all the general goals in programming language design listed in the last section.

3.2 Task Control Management

Task control management consists of:

- control of scheduling,
- parameters for tasks,
- intertask communication,
- mutual exclusion among independent tasks, and
- support of critical sections.

Some real-time applications often use explicit control over the scheduling and dispatching to meet their demanding performance requirements. Another very desirable capability in many applications is an interface specification that provides periodic scheduling of tasks at a fixed frequency.

There are situations in real-time applications where a design approach requires the ability to adjust task priorities and time slices dynamically. For example, to drop the load when the system detects failures. A dynamic priority scheme along with task identifiers may allow the control of the scheduling and dispatching decisions.

Parameters for tasks are an excellent way of expressing parallel distribution of information to a collection of tasks. This feature makes a task unit code reusable when each of

which may require independent initialization.

Intertask communication requires that asynchronous tasks allow exchange of messages (information) or signals (synchronization only).

Mutual exclusion among tasks requires the serialization of their accesses to the same critical section of code.

Some applications must be sure that certain critical sections of code execute until completion without interruption by any other application task. This desirable control is a dangerous feature but justifiable [ARTE87] particularly if there is a timing constraint in the section. The following are the kinds of events that are not desired during a critical section:

- hardware interrupts (could have dangerous effects: interrupt latency),
- software interrupts (by other application tasks),
- preemption at the end of the time slice.

To avoid having to resort to assembly language, the language introduces the ability to control interrupts and preemption.

3.3 Timing Constraint Specifications

The need often arises to execute tasks with a very small variation between a requested time of execution and the actual time of execution. There should be a way to guarantee bounds on the time of execution. This variation can cause serious problems, particularly during overload situations.

In general, it is not possible for the compiler (or programmer) to guarantee that the task will execute within the specified response time. The best that can be done is to provide a mechanism that allows the programmer to:

- specify the desired response time, and
- specify strict upper bounds on the number of executions of all blocked tasks (for interrupts or events) and of all loop constructs.

Another crucial timing constraint is that each task must be schedulable within a certain time.

In fact this supports two sorts of tasks recognized in real-time applications: periodic and sporadic tasks [Lee85][Mok83]. A periodic task becomes ready at regular intervals and a sporadic task may become ready at any time.

3.4 Hardware Environment Specifications

RTPLs must provide constructs to access the hardware and make the software portable. A hardware environment specification must represent the following three important aspects: time, input/output interface and exceptions.

Time Access

Time includes real (absolute) time and elapsed (relative) time. Time-stamping needs the real time access. Any time measurements requires kinds of elapsed time access (time delay, time-out, etc.). RTPLs can provide both time facilities with Time and Delay statements.

I/O Interface Access

Hardware I/O interface requires access to the I/O registers.

Memory mapped processors address these registers as memory locations by using common load and store instructions. On the other hand, port mapped processors address these registers as port locations by requiring special I/O port instructions.

In either case, a "variable at" interface can represent a uniform access to an I/O register. This interface for memory and port mapped accesses will be through an absolute memory pointer [Holt83] or a library port function [Inte85] respectively. A RTPL must also provide an interface to interrupts. Interrupts are like instances of intertask communication between a driver task and a hardware device task.

Exceptions and Interrupt Control

Tasks operate continuously in the presence of possible run-time errors. Handling such errors in the traditional way is useless. A mechanism must handle these errors at both task and system levels. A task exception mechanism is essential to handle event(s) or error(s) caused by the execution of its local code. At the system level, an exception is like an asynchronous interrupt triggered by any detected failures coming from other system tasks. So, an exception handler is a software interrupt control.

Chapter 4

Requirements-Oriented Survey of Real-Time Programming Languages

The previous chapter has presented some important requirements for the design of a real-time programming language. These requirements determine the suitability of several existing Real-Time Programming Languages (RTPLs), Concurrent Programming Languages (CPLs), and some System Programming Langages (SPLs). Also, the results of other recent surveys [Abra82][Glig83][Appe85][Ghez87][ARTE87] take to account for the aspect of language design requirements. The survey done in this chapter serves as a starting point in identifying these requirements. The following languages have been surveyed:

- Ada
- CHILL
- Concurrent Pascal
- Edison
- Modula-2
- PEARL
- Turing Plus
- Real-Time Euclid

4.1 Ada

The Ada programming language was designed to meet many requirements presented in the Steelman document: packages, generic packages and procedures, overloaded function names, portable typing mechanisms, variant records, tasks, exception handling, separate compilation. Embedded computer systems are its main application area.

General Goals

The language has many facilities, a large description, and is rather complex. Some of the constructs are far from simple, and therefore are difficult to understand.

Task Control

Communication and synchronization among tasks is done by the Rendezvous mechanism. A Rendezvous is a form of remote procedure call. The language contains no facilities for the control of scheduling, task parameters, and critical sections. Ada provides no mechanism for asynchronous intertask communication.

Timing Constraints

The nondeterministic select statement cannot guarantee timing properties that involve upper bounds.

Hardware Environment

Ada provides several mechanisms for handling interrupts and hardware devices. However, the language provides no means of specifying timing constraints.

Official Definition

[ANSI83].

Important Papers

[DOD78] [Ichb79] [DOD80] [ARTE87] [Parr88] [Risi88] [Stan88].

4.2 CHILL (Ccitt HIgh Level Language)

CHILL is a language for parallel programming on single processor machines and distributed architectures. Several manufacturers of telecommunication switching systems use CHILL. Pascal is the foundation of the sequential part of CHILL. Modula and other sources inspired the constructs for parallel programming.

General Goals

CHILL is a rather complex language because of the CCITT "committee effect." It requires a large compiler and provides no separate compilation.

Task Control

Communication and synchronization among tasks is done by three mechanisms: regions, buffers and signals. Regions provide mutual exclusive access to shared variables declared within a region. And finally, signals are direct transmissions from one task to another without buffering. However, the language provides no means for controlling scheduling and critical sections.

Timing Constraints

None.

Hardware Environment

CHILL provides nine language-defined exceptions and several mechanisms for handling interrupts and hardware devices. However, it provides no mechanisms for accessing time and the I/O interfaces.

Official Definition

[CCIT80a].

Important Papers

[CCIT80b] [CCIT83] [Smed83].

4.3 Concurrent Pascal

Concurrent Pascal (CP) is an extension of a Pascal subset. Writing structured concurrent programs such as operating systems are its main application area.

General Goals

The language is far from simple and small. Processes, classes and monitors are the mechanisms that provide data abstraction. CP programs are difficult to read because their organization are in acyclic graph form. CP provides no separate compilation.

Task Control

Communication and synchronization among tasks is done by monitors.

Timing Constraints

None.

Hardware Environment

CP provides only an interface to memory-mapped device registers (based on its implementation on the PDP-11). Non-mapped architectures support is not part of the language definition. There is no mechanism for handling exceptions or interrupts and the language provides no time access.

Official Definition

[Brin75].

Important Papers

[Hoar74] [Howa76a] [Howa76b] [Glig83].

4.4 Edison

Edison was designed for implementing reliable real-time programs for multiprocessor systems with shared memory.

General Goals

Edison is based on Concurrent Pascal and Modula. Its design emphasizes simplicity by omitting the language constructs that are not strictly necessary. The expense of omitting these constructs decrease program readability (e.g. Pascal With statements) and program efficiency. The principal encapsulation construct is the module.

Task Control

The When statement supports task synchronization. The statement list of all When statements in a program are critical sections which are mutually exclusive.

Timing Constraints

None.

Hardware Environment

Edison provides only facilities for controlling peripheral devices. In fact, its implementation on the PDP-11 ignores interrupts completely (even at the machine level). Standard procedure calls (place, obtain, and sense) control the I/O interface.

Official Definition

[Brin82].

Important Papers

[Dubn88].

4.5 Modula-2

Modula-2 is a language for small computers. It was designed primarily for implementation on a conventional single processor and provides only a simple coroutine mechanism for quasi-parallel tasks.

General Goals

All primitives have simple semantics. The language provides separate compilation.

Task Control

Modula-2 has neither explicit mutual exclusion nor other synchronization constructs. The programmer must provide these constructs himself.

Timing Constraints

None.

Hardware Environment

Modula-2 does not provide mechanisms for exception handling, but does provide a mechanism for implementing device drivers.

Official Definition

[Wirt82].

Important Papers

[Spec72] [Ande86] [Hopp86] [Gree86] [Feld86].

4.6 PEARL

PEARL (Process and Experiment Automation Real-Time Language) has been designed for real-time systems but also for systems programming in general.

General Goals

PEARL is a considerably more complex language than either Modula-2 or Edison because some of its primitives are based on PL/I. Extensive documentation on PEARL exists but no formal definition. The language provides separate compilation.

Task Control

PEARL includes only priority scheduling facility.

Timing Constraints

PEARL includes timing facilities such as time, deadline and period-based scheduling.

Hardware Environment

PEARL provides an excellent mechanism to control device drivers and to handle priority interrupts.

Official Definition

[Mart78].

Important Papers

[Mart79].

4.7 Turing Plus

The Turing language evolves from Concurrent Euclid. It improves on it by adding convenient I/O, checking of uninitialized variables, and type-safe variant records. Turing also checked separate compilation, parametric procedures, and dynamic arrays.

Turing Plus is a general-purpose programming language that extends Turing to systems applications. These applications are operating systems, network controllers, basic device drivers, and embedded software. This extension was done by adding dynamic concurrent processes, monitors with immediate and deferred condition queues, inline assemb-

ly code, type conversion, convenient acces to underlying hardware, and exception handling.

General Goals

Turing Plus was designed with a striving for elegance. It increases utility and reduces complexity. Turing Plus has a concise and expressive syntax, and is easy to learn [Holt88]. The language provides separate compilation.

Task Control

Communication and synchronization among tasks is done by a special kind of module (called the monitor). However, the language provides no means of control of scheduling, task parameters, and critical sections.

Timing Constraints

None.

Hardware Environment

Turing Plus provides only an interface to memory-mapped device registers. There is no formal interrupt control (even if nothing prevents the alternation of the interrupt status register) since the kernel hides the mechanism. The language provides no time access.

Official Definition

[Holt85].

Important Papers

[Holt83a] [Holt88].

4.8 Real-Time Euclid

Real-Time Euclid (RTE) is a descendant of such languages as Pascal, Euclid, Concurrent Euclid, and Turing Plus. RTE has been designed to address issues of reliability and guaranteed schedulability in real-time systems. The philosophy of the language is that every exception detectable by the hardware or the software must have an exception handler clause associated with it. The language definition forces every construct to be time- and space-bounded.

General Goals

RTE (like CE[1]) is a considerably more complex language than either Turing Plus, Modula-2 or Edison. Recently, Holt stated in the design goals for Turing: "The desire to increase generality (in CE) was tempered (in Turing Plus) by the competing goals of language simplicity, efficiency and implementability ... more general features entail too much complexity or inefficiency to compensate for their potential benefits." [Holt88].

Task Control

Communication and synchronization among tasks is done by a special kind of module (called the monitor). However, the language provides no means to handle task parameters.

1 Concurrent Euclid [Holt83b]

Timing Constraints

RTE has time-bounded loops and activation information for schedulability.

Hardware Environment

RTE provides only an interface to memory mapped device registers. The language supports interrupt control and time access.

Official Definition

[Klig86].

Important Papers

[Pope77] [Holt83b].

The following tables 4.1, 4.2, 4.3, and 4.4 summarize the results of the survey. These tables represent the 4 main specific requirement aspects in the design of a RTPL (see section 3):

- General Goals in Programming Language
- Task Control
- Timing Constraints
- Hardware Environment

Table 4-1 General Goals in Programming Language Design Requirements

	Ada	CHILL	CP[1]	Edison	Mod2[2]	PEARL	TP[3]	RTE[4]
Simplicity	no	poor	poor	yes	yes	poor	yes	poor
Usability	no	poor	poor	poor	yes	poor	yes	poor
Abstraction	yes	yes	yes	yes	yes	yes	yes	yes
Separate Compilation	yes	no	no	no	yes	yes	yes	yes
Expressiveness	yes	poor	poor	yes	yes	poor	yes	yes

Table 4-2 Task Control Management Requirements

	Ada	CHILL	CP	Edison	Mod2	PEARL	TP	RTE
Control of Scheduling	no	no	no	no	possible	partial	no	yes
Task Parameters	no	yes	no	no	no	no	yes	no
Communication	yes	yes	yes	yes	yes	yes	yes	yes
Mutual Exclusion	yes	yes	yes	yes	possible	yes	yes	yes
Critical Sections	no	no	no	yes	no	no	yes	yes

1 Concurrent Pascal
2 Modula-2
3 Turing Plus
4 Real-Time Euclid

Table 4-3 Timing Constraint Requirements

	Ada	CHILL	CP	Edison	Mod2	PEARL	TP	RTE
In Scheduling	no	no	no	no	no	partial	no	yes
In Constructs	no	no	no	no	no	no	no	yes

Table 4-4 Hardware Environment Requirements

	Ada	CHILL	CP	Edison	Mod2	PEARL	TP	RTE
Interrupt Control	yes	yes	no	no	yes	yes	yes	yes
I/O Interface Access	yes	yes	partial	yes	yes	yes	partial	partial
Time Access	no	no	no	no	no	no	no	yes

For each table, a list of all the languages surveyed can be ordered from the most eligible (left) to the least eligible (right) by aspect of requirement:

Table 4-5 Eligible in General Goals in Programming Languages

Modula-2 $=^{1}$ Turing Plus $>^{2}$ Ada = Edison = Real-Time Euclid > PEARL > CHILL = CP

1 Almost the same eligibility than
2 More eligible than

Table 4-6 Eligible in Task Control Management

Turing Plus = Real-Time Euclid > CHILL = Edison = CP > Modula-2 = PEARL > Ada

Table 4-7 Eligible in Timing Constraints

Real-Time Euclid >>> PEARL > Other languages

Table 4-8 Eligible in Hardware Environment

Real-Time Euclid > Ada = CHILL = Modula-2 = PEARL > Turing Plus > Edison > CP

And finally, a list that considers the four aspects combined:

Table 4-9 Language Suitability[1] in the four areas as a RTL

Real-Time Euclid > Modula-2 = Turing Plus > PEARL = Edison = Ada = CHILL > CP

In conclusion, Real-Time Euclid, Modula-2, and Turing Plus arise among the best real-time programming languages.

1 Based on the following rates: yes = 1, possible (or almost) = .75, partial = .5, poor = .25, and no = 0

Chapter 5

The Synapse Language Report

5.1 Introduction

This report defines the real-time programming language Synapse[1]. The language main goal is to be small and expressive, so each of the primitives has a simple and well-defined semantics. Small embedded real-time microprocessor systems and device drivers are its main application area. The language reaches this goal by supporting the following requirements:

- General Goals in Programming Language Design
- Task Control Management
- Timing Constraint Specifications
- Hardware Environment Specifications

1 A synapse (pronounced "Sin-apse") is a signal transmission pathway interconnecting a collection of processing elements (called neurons or nerve cells) in a neural network in the human brain.

5.1.1 Design Goals

General Goals in Synapse

The following list mentions the main properties bestowed on Synapse. It is:

- understandable (good human interface),
- small (minimality),
- simple (simplicity),
- easy to learn and remember,
- provided with availability of data abstractions (level of abstraction),
- equipped with separate compilation (modularity), and
- built on well-defined language primitives (expressiveness).

Synapse is a modern block-structured language with a line-oriented approach and line indentation. Even if most modern languages are context-free, their users apply a line-oriented convention by placing their source text on separated lines to improve readability. This approach used by Synapse merely enforces what programmers do by adhering to a common programming style. Many languages, including natural languages, use expressively indentation and line separation. Synapse uses indentation and line separation to identify the task structure and the statement context. This enhances the readability of the code.

Simplicity is achieved by restricting goals, taking care of readability and expressiveness, and basing the language around a few well-defined, simple concepts.

A "#" character precede a Synapse's comment. The end of the line terminates it. This design choice prevents the common programming error of failing to close a comment (in Pascal, Modula-2 or C) thus causing code to be accidentally "commented out." This type of error is very hard to detect when reviewing code [Hoar73] [Trem85].

Encapsulation is a mechanism to structure a system by separating it into compilation units. Synapse compiles separately each task unit to address two issues:

- to enforce modularization.
- to reduce the time required for recompilation of units.

The segmentation of a problem is more important than any hiding of information [Gree86]. Most alarming problems in the industry is that too many applications are large [Feld86].

One of the main purposes of a task unit is to implement an abstract data-type that operates upon by a fixed set of operations executed by handlers.

Modula-2 [Wirt83], Praxis [Evan81], and Ada [Ichb79] already provide some facilities for encapsulation. These languages are even looser at allowing control of the scope of imported variables. The "lazy and easygoing import-all approach" is too permissive [Feld86]. Unfortunately, this approach is the one used by most programmers. A program maintenance problem occurs when an unqualified imported identifier conflicts with a local identifier or another unqualified imported identifier. For example, the Use statement in Ada unqualifies all identifiers to an entire compilation unit, a single subprogram, or a block.

The Synapse solution to the above problem is to force selective imports. This avoids massive importation into a task and forces the use of qualified identifiers. In reaction to many encapsulation problems, Synapse enforces some design rules by:

- Always having a qualified form for exported identifiers, consisting of an exported task name followed by a dot followed by an exported handler name;
- Requiring selective imports to restrict the massive importation of exported operations through the scope of a specific handler in a task [Ande86];
- Allowing export to accord public access of handlers;

- Restricting any exportation and importation of variables (as opposed to operations). Because, global variables are harmful and result in excessive coupling of program segments [Myer78]. Their usage (if allowed) should be only in case of desperate performance problems.

Synapse recognizes the need to avoid massive recompilation of units, and generates if necessary an updated task interface file.

Languages with higher concepts for concurrency, like Ada (task concept with rendezvous), Concurrent Pascal (monitor concept), and Edison (concurrent statement), specify very precisely the semantics of the supported communication method [Hopp86]. The Synapse language (task concept with interrupts) belongs to this class. It offers a compiler that can be sure the context where these concepts are used, and it promotes thereby a reliable language.

Synapse is also designed to favor more secure programs by ensuring that the constructs available to the programmer are without potential side-effects. This goal is achieved by eliminating or catching most of the trivial errors at compile time.

Pointer variables are also excluded from Synapse because at run time, it is generally difficult to know if they point to the right objects. Detection and analysis of runaway-pointer errors, therefore, is almost impossible to achieve. Like Edison [Brin82], we favor simplicity over endangering the security of a language by trying to get efficiency "the most overemphasized goal" in programming-langage development [Trem85]. Even good programmers make significant errors using pointers, so removing them looks like the best way to reduce errors [Feld86]. Synapse does not have operators that cause side effects inside expressions, such as the ++ and -- in the C language [Holt88]. Another improvement for operators is to restrict the number of operator-priority levels to achieve

readability and obviousness in their use. The C language [Kern78] has overloaded its operator-precedence structure. It uses fifteen different levels: this makes it quite confusing and proves extremely hard to remember.

Also, to prevent side-effects inside expressions, handlers are allowed to have only input parameters, as in Concurrent Euclid [Holt83].

Synapse allows the declaration of open-array parameters to write general-purpose handlers that can work with arrays of arbitrary size.

Task Control Requirements

The following are the foremost task control requirements retained for Synapse:

- control of scheduling,
- parameters for tasks,
- inter-task communication,
- mutual exclusion among independent tasks, and
- support of critical sections.

Some real-time applications to meet their demanding performance requirements often use explicit control over the scheduling and dispatching regime. Synapse specifies the time slice and the priority of tasks in the interface specification. This feature is a very desirable capability in many applications. There are also situations, in reaction to excessive loads, in which the design approach requires the ability to adjust task priorities dynamically. Synapse provides dynamic priority and time-slice options along with task identifiers to allow the control of the scheduling and dispatching decisions.

Synapse supports parameters for tasks, to express parallel distribution of information to them. This feature makes a task unit code reusable when each of which may require in-

dependent initialization.

A real-time application must respond to a series of external asynchronous hardware interrupts, which may occur at any time. On the software side, multitasking allows independent execution of program entities (tasks) that send to others via software interrupts. Synapse supports both hardware and software interrupts, as well as the handlers that serve them, using uniform language constructs. This notion is completely processor-independent; it brings a uniform degree of abstraction to interruption/handler, such that is seldom available in traditional mechanisms of high-level language or operating system.

The designers of Ada [Ichb79] have provided a link between tasks by defining the rendezvous mechanism. This mechanism is based on synchronous (blocking) message passing and has the restriction that an inter-task communication cannot be completed without the execution of a reply by the receiver task. Ada cannot easily provide asynchronous inter-task communications.

The synchronization requirements for tasks include:

- inter-task communication,
- mutual exclusion among independant tasks.

Inter-task communication requires that asynchronous tasks be allowed to exchange messages (information) or signals (synchronization only) with each other. Mutual exclusion among tasks requires that the access of asynchronous tasks to the same critical section of code be serialized.

Synapse regroups and supports the above requirements by assuring to an asynchronous task that interrupts another, an exclusive access during its execution to the critical section of code called the target handler.

The main goal of Synapse is to offer a more flexible task communication mechanism by sending and handling interrupts with message transfer. The concurrent programming model proposed by Synapse is that of asynchronous message passing via interrupts. Communication between tasks is effect by sending interrupt messages to handlers. This inter-task communication concept is in fact a natural extension of the hardware interrupt mechanism. Synapse makes both hardware and software interrupts uniformly available in high-level language facilities.

These facilities also solve the problem of handling exceptions in tasks by allowing an immediate interruption of the task receiving and serving the exception. Interrupt or exception handling allows the transfer of information (in the form of messages) between tasks via an optional parameter list. The flexibility of this mechanism also allows the choice of asynchronous or synchronous message passing, with the possibility of unidirectional or bidirectional information transfer, where and when it is required.

Some runtime applications are required to be sure that certain sections of code can be executed until completion, without interruption by other application tasks. This is especially true if there is a timing constraint on a section of code. Most of RTPLs cannot insure that a task will not be preempted by the executive[1] because of the interleaved execution of tasks due to the time-slicing. This is a dangerous feature, but it has been shown that real-time programmers will find a way (even through coding in assembly language) to get the equivalent effect of disabling preemption (or even worse: disabling interrupts) [ARTE87]. Synapse offers the ability to disable/enable preemption and/or interrupts without going

1 also called the kernel

through an assembly language.

Timing Constraint Requirements

The following are the foremost timing constraint requirements retained for Synapse:

The need often arises in real-time systems for task(s) to be executed with a very small variation between a requested time of execution and the actual time of execution. There should be a way to guarantee bounds on the time of execution. This variation can cause serious problems, particularly during overload situations.

In general, it is impossible for the compiler (or programmer) to guarantee that the task will execute within the specified response time. Synapse provides the constructs that allow the programmer to:

- specify the desired response time (if a task is waiting), and
- specify strict upper bounds on the number of executions of all loops.

These constructs assure that each task must be schedulable within a certain time limit; if not, an exception will be raised to inform the programmer at run-time.

Hardware Environment Requirements

RTPLs must provide constructs to access the hardware and make the hardware environment portable. A hardware environment specification must represent the following three important aspects: time, input/output interface and exceptions.

Time includes real (absolute) time and elapsed (relative) time. Time-stamping needs the real time access. Any time measurements requires kinds of elapsed time access (time

delay, time-out, etc.).

Hardware I/O interface requires access to the I/O registers. Memory mapped processors address these registers as memory locations by using common load and store instructions. On the other hand, port mapped processors address these registers as port locations by requiring special I/O port instructions. The ability to manipulate device registers includes the reading and the writing of registers. Synapse provides a construct to specify a memory-mapped or port-mapped device register. This feature allows it to support either type of processor architecture. In either case, a "variable at" interface can represent a uniform access to an I/O register. This interface for memory and port mapped accesses will be through an absolute memory pointer [Holt83] or a library port function [Inte85] respectively in a way which is completly transparent to programmer. Synapse also supports an interface to interrupts. Interrupts are like instances of intertask communication between a driver task and a hardware device task.

Tasks operate continuously in the presence of possible run-time errors. Handling such errors in the traditional way is useless. A mechanism must handle these errors at both task and system levels. A task exception mechanism is essential to handle event(s) or error(s) caused by the execution of its local code. At the system level, an exception is like an asynchronous interrupt triggered by any detected failures coming from other system tasks. So, an exception handler is a software interrupt control. Synapse supports exception handlers, whether exceptions are caused by hardware or by software.

5.1.2 Terminology and Basic Concepts

This section informally presents basic terms and concepts that differentiate Synapse from other real-time languages.

Real-Time Application and Task Units

A Synapse real-time application is composed of one or more task units, where each task unit is in a file that defines one task and must be compiled separately.

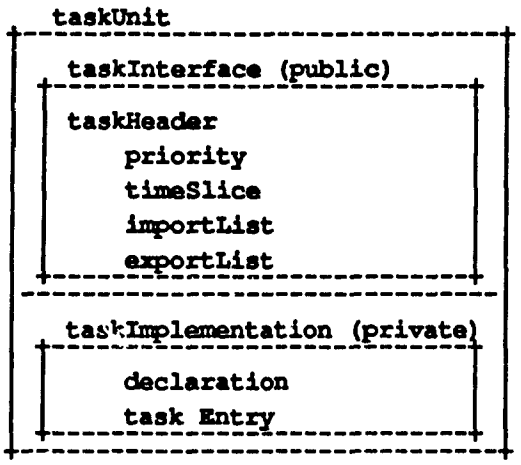

Figure 5-1 Two parts of a task unit

As Figure 5-1 shows, a task unit consists of two parts: the task interface and the task implementation. The task interface is the public (visible) part of the task unit. It contains the task header followed optionally by its priority, time slice, import list, and export list. The task implementation is the private (hidden) part of the task unit. The bodies of the handlers declared in the task interface reside here. These handlers are the only exportable entry points. Additional constants, variables, and devices can be declared and used within the implementation. These declarations must be followed by a task entry: this is

where the task begins its execution. Likewise, additional private handlers may exist in this section. However, all these objects are invisible to other task units (also called client tasks). A client task does not know that they exist and cannot reference or interrupt them. These hidden objects can only be used locally in the task unit. A modular real-time application in Synapse is achieved through information hiding, which enforces a rigorous separation between interface and implementation.

Symbolic Representation of a Task Unit

A task unit symbol is needed to represent complex multitasking real-time applications. Figure 5-2 shows a task symbol that emphasizes the only visible interface of a task unit: the task name and its handler entry points.

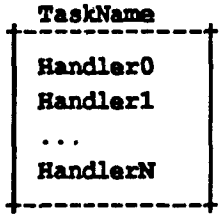

Figure 5-2 Symbolic Representation of a Task Unit

Inter-task Communication Model

The inter-task communication in Synapse is done through interrupts. This form of communication between tasks is called asynchronous transmission and reception of messages. The actions taken upon the arrival of a message are very similar in behavior to the handling of a hardware interrupt on a single processor. Sending interrupts is an intuitive

and very natural basic method for intertask communication in a real-time system.

Information (or synchronization) is transferred by sending and handling interrupts. Handlers are intended to deal with the reception of asynchronous interrupt messages. A handler is an interrupt service routine (ISR). It is declared solely in a task implementation that is dedicated to serve an interrupt. Once an interrupt call has been sent, the handler-Body is executed and assures a mutual exclusion in the body during its execution. There are two kinds of handlers: software and hardware. The software interrupt handler is a natural extension of the hardware interrupt mechanism. Only software handlers can receive parameters from interrupts. A handler behaves like a procedure and returns to the point of interrupt when the service is completed.

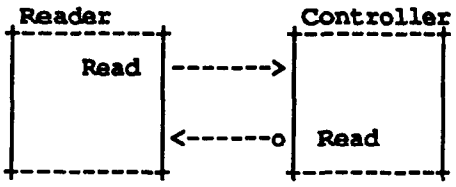

Figure 5-3 Inter-task Communication Diagram

Figure 5-3 shows two communicating tasks (Reader and Controller). The arrow indicates an asynchronous interrupt from the calling task to the interrupted task. A simple arrow represents an interrupt for synchronization only (no information in the message), since an arrow with a small circle represents an interrupt with information in the message. In this figure, the task Reader interrupts the task Controller to get a value that is sent back via an interrupt with a message containing the value.

Input and Output

Input and output statements have been implemented and introduced in the Synapse language for two reasons:

First, to test real-time applications easily, integer and string input-output (I/O) statements have been implemented. This basic support I/O is available to ease any software development. Obviously, they could be replaced (and/or ignored) if more powerful device drivers are written for I/O.

Secondly, these statements are extremely useful to show practical examples in this report. An I/O statement is a Get (input) or a Put (output) statement.

A Get statement reads getItem(s) from the keyboard. A getItem is a variable reference. The following input:

```
255 65535
```

is used in this example:

```
var aByte:byte
var aWord:word
get aByte, aWord
# aByte = 255, aWord = 65535
```

A Put statement writes putItem(s) to the screen. A putItem is an expression or a string constant:

```
put aByte
put 65535
put "Hello world!"
put "The answer ", 5, " is correct"
```

Separate Compilation

Synapse recognizes the need to be able to compile a program in distinct pieces and provides facilities for both bottom-up and top-down approaches. The Synapse language allows the compiler to fit smoothly into a operating system environment by having a direct file name correspondence between the task header and the disk operating system (DOS) file names.

This approach is of paramount importance when developing large systems with many programmers. It enables the programmers to produce and test modules concurrently in relative isolation from each other. Modularization enables the programmer to partition a software system into various tasks and regroups into a single task unit both data and the handlers needed. This serves to localize the scope of attention of the programmer and reinforces information hiding.

Separate compilation is reinforced in Synapse by the definition of a task unit: a file that MUST be compiled separately. This concept enhances the design of independent units with clean interfaces by refining the quantity and the type of communication between units.

This type of modularization reduces the time required for recompilation of units because the Synapse compiler takes care of generating an interface file with "`.int`" extension that contains the visible part of a task unit. Each time the programmer changes and recompiles a task unit, the compiler checks whether the changes have affected the interface file and updates it if necessary. All dependencies between tasks, which occur only through handlers, are specified by the import and export clauses.

The next section defines the separate compilation facility in Synapse. In what followings, the task Reader and the task Controller are separately compiled.

The reader.syn file contains:

```
task Reader in "reader.syn"
    from   Control import Read
    export Read
    var    value:word

    handler Read(v:word)
        value := v

    entry
        loop
            # ...
            interrupt Control.Read
                enable wait Read
            # ...
```

The controlr.syn file contains:

```
task Controller in "controlr.syn"
    from   Reader as caller import Read
    export Write, Read
    var    value:word

    handler Write(v:word)
        value := v
        put "writer#", caller, " writes ", value, '\n'

    handler Read
        put "reader#", caller, " reads  ", value, '\n'
        interrupt caller.Read(value)

    entry
        value := 5
        loop
            enable wait Read,  Write
            enable wait Write, Read
```

The interface of a task unit (visible part) is generated when it is compiled. The Synapse compiler can be invoked to translate the task unit `reader.syn` to a C source file (`reader.c`) and an interface module (`reader.int`) by this command:

```
sc reader.syn
```

The following invokes the translation of the task unit `controlr.syn`:

```
sc controlr.syn
```

The corresponding interface modules (`.int`) are generated as follow. Notice that the export list is replaced by all the corresponding handler headers exported.

The reader.int file contains:

```
task Reader in "reader.syn"
    from    Controller import Read
    handler Read(v:word)
```

The controlr.int file contains:

```
task Controller in "controlr.syn"
    from    Reader as caller import Read
    handler Write(v:word)
    handler Read
```

If a *make* utility program is available, a makefile can be constructed based on all the dependencies of the above files:

```
controlr.c: controlr.syn reader.int
        sc  controlr.syn

reader.c:  reader.syn  controlr.int
        sc reader.syn
```

With the above makefile, all compilations are reduced to a minimum because each interface module contains all the information needed to check for legimate references of the task unit, thus avoiding massive recompilations of units. It is also worth mentioning that dependencies should not be among ".syn" files but rather among ".int" files. A modification of a ".syn" will not necessary affect the corresponding ".int".

Line-Oriented Approach with Indentation

An important goal is to provide a modern block structured language with a line-oriented approach [Lein80][Long86]. Most languages are used in a line-oriented fashion whether or not they require it: **BEGIN**, **END**, **DO**, **OD**, **{**, **}** are placed on separated lines to improve readability. This line-oriented technique merely enforces what programmers do anyway by convention. Also, people use indentation and line separation in a meaningful manner in many languages, including natural languages. In the Synapse Report, the terminal symbol NL shows a new-line followed by zero or more indentation levels. So, like Occam [Inmos 84], Synapse uses indentation from the left margin and line separation to show program structure and statement context: this yields a clean repre-

sentation by enhancing the readability of the code. This approach solves the perceived ambiguities in context-free language associated with *If-Else* situations: *Else* is always associated with the closest previous *Else-less If* even if they are not properly indented [Kern88].

Indentation to Specify Task and Block Structure

Synapse makes the use of indentation to specify the task and block structures to both the programmer and the compiler. A syntax-oriented editor (SDE) should be a considerable help to aid the programmer in his code entry to get a "syntactically correct compilation unit." An SDE will take a typed token for a construct, and infer the correct indentation.

The following example a typical task unit with levels of indentation to specify the task and block structures:

```
# declaration of a task named Reader
# physically stored in the file "reader.syn"
task Reader in "reader.syn"
    # Declaration in a task (visible in handlers and task entry)
    var aGlobalVariable:byte

    handler Read(aParameter:byte)
        # Declaration of i (visible in handler Read only)
        var i:byte

        # Statement in a handler
        i := aGlobalVariable + aParameter

    # taskEntry
    entry
        # Declaration of i (visible in task entry only)
        var i:byte

        # Statements in task entry
        aGlobalVariable := 0
        i := 10
        # countdown 9 8 ... 0 go!
        loop
            # the 3 following statements are the loop's body
            i := i - 1
            put i
            exit when i = 0
        put "go!"
```

5.1.3 Language Summary

The following summary gives only a brief language's overview without discussion. Refer to §5.1.1 for design goals, comments on advantages and disavantages, and other people's experience.

A Synapse real-time application is composed of one or more task units. Each task unit is a file that defines one task and must be compiled separately.

Task Units

A task unit consists of two parts: the task interface and the task implementation.

The task interface is the public (visible) part of the task unit. It contains the task header followed optionally by its priority, time slice, import list, and export list.

The task implementation is the private (hidden) part of a task unit. A task implementation may have zero or more declarations of a constant, a variable, a device, a hardware handler, or a software handler and a task entry. All these declarations are logically and textually hidden from other task unit(s), except software handlers that can be exported.

Declarations

Constant, variable, and device declarations introduce respectively: a constant identifier, one or more variable names, and a device name.

Information is transferred between tasks by sending and handling interrupts. A handler is an interrupt service routine (ISR) declared only in a task implementation dedicated to serve an interrupt. Once an interrupt call has been received, the handler body is executed. There are two kinds of handlers: software and hardware. Only software handlers can be visible (exported) to other client tasks, these exportations are specified in the

task interface. Handlers are like critical sections, they provides mutual exclusive access to shared variables declared within a task. So, it is highly recommended to test or update these variables in handler's body. Any access on them in the task body should be done in a time-critical section of code to avoid preemption.

Statements

The sequence of statements describes a sequence of actions that are to be performed.

An *Assignment* statement changes the value of a variable, a device, a task's priority, or a task's timeslice.

The *If* statement allows the selection of an inner sequence of statements based on the value of an expression.

The *Loop* statement is the basic iterative mechanism, and specifies a sequence of statement(s) to be executed repeatedly until *Exit When* statement causes an immediate exit.

An *Interrupt* statement is used to send information to any handler of any task.

The *Enable/Disable-hardware* statement enables/disables the interruption of all hardware handlers.

The *Enable/Disable-device* statement enables/disables specific device number(s).

The *Enable/Disable-preemption* statements are used when some time-critical sections of code must be guaranteed to be executed until completion without preemption.

The *Enable/Disable* statement enables/disables all software handlers and thus allow/prevent them from being interrupted by other tasks.

The *Delay* statement allows a task to suspend itself for a given interval.

The *Start* statement creates and schedules the execution of task name(s), on the basis

of the priority and the timeslice established in each task interface.

The *Schedule* statement resumes the execution of task(s) by inserting it(them) onto the ready queue.

The *Suspend* statement stops the execution of task(s) indefinitely, until a *Schedule* statement is made on behalf of that(those) task(s).

The *Terminate* statement allows the running task to kill another task.

The *Reschedule* statement moves the running task onto the ready queue, and completes a context switching by scheduling a new task to run.

There is basic support for input and output with *Get* and *Put* statements implemented to test real-time applications, and to ease software development.

Data Types

The main classes of types are standard types, array types, and open-array types.

A standard type is a byte or a word. An array type is a structure made up of a fixed number of items (of standard type). An open-array type is the base address of an array and is used as parameter for handlers to work with arrays of arbitrary size.

5.1.4 Method of Description and Syntax Notation

The form of Synapse task units is described by line-oriented syntax with context-dependent requirements expressed by narrative rules and indentations.

The meaning of Synapse task units is described by narrative rules defining both the effects of each construct and the composition rules for constructs. This narrative employs technical terms whose precise definition is given in each section.

Synapse uses indentation from the left to right margin and line separation to identify

the task structure and statement context, to give a clean representation by enhancing the readability of the code. The terminal symbol *NL* (new-line) is a special symbol in the Synapse character set. One or more indentations take effect only if they immediatly follow a *NL*. The pattern of indentation is crucial for the interpretation of Synapse by the compiler.

The syntax of Synapse is described in a variant of BNF (Backus/Naur Form). The syntactic symbols are of two kinds: terminals and non-terminals.

The **boldface typewriter** style identifies terminal symbols including keywords and single characters (or special symbols).

Examples:

```
task  priority  import  delay  var
```

The *italic* style identifies non-terminal symbols. These symbols represent names using upper and lower case letters.

Examples:

realTimeApplication taskUnit variableDeclaration expression

A syntax rule is a non-terminal followed by a colon to introduce its definition.

Example:

delayStatement :
    delay *expression*

An alternative syntax rule lists its definitions on separate lines, except when the word "one of" followed the colon.

Example of equivalent rules:

multiplicativeOperator :
    *****
    /
    mod

multiplicativeOperator : one of
`*` `/` **mod**

Braces *{* and *}* enclose a repeated symbol. The symbol may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.

Example of equivalent rules:

realTimeApplication :
taskUnit *{* *taskUnit* *}*

realTimeApplication :
taskUnit
realTimeApplication *taskUnit*

Brackets *[* and *]* enclose an optional terminal or non-terminal symbol.

Example of equivalent rules:

deviceDeclaration :
device *deviceName* : *standardType* *[* *atDeviceLocation* *]* *NLs*

deviceDeclaration :
device *deviceName* : *standardType* *NLs*
device *deviceName* : *standardType* *atDeviceLocation* *NLs*

The syntax does not explicitly defined the following non-terminal symbols. They are equivalent to *identifier* (§5.2.2):

constantName *variableName* *deviceName*
handlerName *taskName* *parameterName*

The syntax does not explicitly defined the following non-terminal symbols. They correspond to the general rule of a *nonTerminalList* given below:

variableNameList *handlerNameList* *expressionList* *constantExpressionList*
taskIdList *putItemList* *getItemList*

nonTerminalList :
nonTerminal *{* , *nonTerminal* *}*

5.2 Lexical Elements

The text of a real-time application in Synapse consists of the texts of one or more compilations. A compilation text is a file called a task unit which is a sequence of lexical elements, each composed of characters. Each task unit must be compiled separately.

5.2.1 Character Set

characterSet : one of
letter digit specialCharacter SP NL

letter : one of
```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z _
```

digit : one of
```
0 1 2 3 4 5 6 7 8 9
```

specialCharacter : one of
```
' ( ) * + , - . / : < = > [ ] " \
```

SP :
the space character

NL :
the new-line character

The character set used in Synapse is the printable subset of the ASCII character set. The Synapse lexical scanner simply filters any character that does not belong to the following set except the non-printable new-line character.

The character set is classified into letters, digits, special characters, space characters, and new-line characters. The corresponding upper and lower case letters are considered different.

5.2.2 Symbols

symbol : one of
separator delimiter identifier keyword constant comment

A Synapse task unit is represented as a text file, consisting of lines each ended by an *NL* character. Text lines are broken into symbols (also called tokens). Each symbol is the minimal lexical element which is composed of characters in the set. A symbol may be a separator, a delimiter, an identifier, a keyword, a constant, or a comment.

Separators

separator : one of
SP NL

A separator is required to separate adjacent symbols. A separator is a space character (*SP*) or a new line character (*NL*). Spaces are required to specify indentations after an *NL* (starting at the beginning of a source line). They are also used to separate identifiers and must not occur within identifiers or operators. Extra spaces are otherwise normally used to improve readability. A space character is a separator except within a comment, a string constant, or a space character constant.

Delimiters

delimiter : one of
`' ( ) * + , - . / : < = > [ ] " \ := /= >= <=`

A delimiter is a special character or two adjacent special characters except within a comment, a character constant, or a string constant.

Identifiers

identifier:
letter { *identifierCharacter* }

identifierCharacter: one of
letter *decimalDigit* _

An identifier is a sequence of letters, digits, or underlines which must begin with a letter. The programmer uses identifiers to refer to the constants, variables, devices, handlers and tasks used in a given application. Upper and lower case letters in identifiers are distinct.

Examples:

```
Synapse    synapse    SYNAPSE    are distinct identifiers
```

Keywords

keyword: one of

and	**else**	**loop**	
as	**enable**		**self**
at	**entry**		**shl**
	exit	**mod**	**shr**
	export		**start**
byte			**state**
		name	**suspend**
	false	**not**	
	from	**null**	**task**
			terminate
caller			**timeslice**
child	**get**	**parent**	**true**
const		**preemption**	
create	**handler**	**priority**	**var**
	hardware	**put**	
			wait
	id	**of**	**within**
	if	**or**	**when**
delay	**import**		**word**
device	**in**	**reschedule**	
disable	**interrupt**	**resume**	**xor**

These predefined identifiers are keywords that have special meaning to the Synapse compiler. They can be used only as defined (they cannot be redefined), and they must be written entirely in lower case letters.

Constants

constant : one of
integerConstant *stringConstant*

A constant is a number or a string of characters that can be used as a value in an application. The value of a constant does not change during execution. The Synapse language has two kinds of constants: integer constants and string constants.

integerConstant : one of
binaryConstant *decimalConstant* *hexadecimalConstant*

An integer constant is unsigned and can be represented in three different bases: binary, decimal and hexadecimal.

binaryConstant :
0b *binaryDigit* [*binaryDigit*]

decimalConstant :
digit [*digit*]

hexadecimalConstant :
0x *hexadecimalDigit* [*hexadecimalDigit*]

binaryDigit : one of
0 **1**

hexadecimalDigit : one of
digit **a** **b** **c** **d** **e** **f** **A** **B** **C** **D** **E** **F**

A binary constant consists of the prefix **0b** followed by a sequence of binary digits. A decimal constant consists of a sequence of decimal digits. A hexadecimal constant consists of the prefix **0x** followed by a sequence of the decimal digits and the letters **a** (or **A**)

through `f` (or `F`). The value of a binary constant is computed to base 2; that of a decimal constant to base 10; that of a hexadecimal constant to base 16. The lexically first digit is the most significant. Integer constants are taken to be words. The keywords `true` and `false` are integer constants that indicate the boolean values `1` and `0` respectively.

Examples:

```
16    = 0x10   = 0b10000
65535 = 0xFFFF
true  = 0b1
false = 0b0
```

stringConstant :
" *stringCharacterSequence* "

stringCharacterSequence :
stringCharacter { *stringCharacter* }

stringCharacter :
any character in the source character set or escapeSequence
except double-quote ("), backslash (\ ), and *NL*

escapeSequence : one of
`\'` `\"` `\\` `\f` `\n` `\r` `\t` `\v` `\0`

A string constant is a sequence of zero or more characters enclosed in double-quotes. A string has a global storage duration and a type "array of bytes" and is initialized with the given characters. A null byte `\0` is appended to the string to determine its end. Certain non-printable characters may be represented according to the following table of escape sequences:

`\'` is a single-quote (')
`\"` is a double-quote (")
`\\` is a backslash (\)
`\f` is a form feed
`\n` is a newline
`\r` is a carriage return
`\t` is a horizontal tab
`\v` is a vertical tab
`\0` is a null byte

Comments

comment :
    # *anyCharactersExceptEndOfLineCharacter NLs*

A comment is a sequence of characters that is considered as an *NL* by the compiler. Except within a character constant or a comment, a sharp sign (#) marks the start of a comment. It is ended by one or more *NLs*. This means that a comment is always clearly identified on all its lines.

Examples:

```
# This is a comment

# This is a long comment split onto
# consecutive lines
```

5.3 Types

type : one of
    standardType arrayType

Data values are grouped into classes called types. The main classes of types are standard types, array types, and open-array types. A type is a named set of values with each value being of one, and only one, type. The operands that yield data values during execution are constants, variables, and expressions. Any value yielded by an operand during execution will be of one and only one type.

5.3.1 Standard Types

standardType : one of
byte word

The types **byte** and **word** are standard types. They are also called integral types.

The following table summarizes the storage associated with each standard type. The table gives also the range of values that can be stored in a variable of each type.

Table 5-1 Standard Types

Type	Storage	Range
byte	1 byte	0 to 255
word	2 bytes	0 to 65,535

5.3.2 Array Types

arrayType :
standardType [*constantExpression*]

An arrayType is a structure made up of a fixed number of items (constantExpression specifies the size of the array) which are all of standardType. The items of the array are accessed by indices that range from 0 to the size of the array minus one (constant-Expression - 1).

Examples:

```
byte [80]       # array type of 80 bytes
word [20]       # array type of 20 words
```

5.3.3 Open-Array Types

openArrayType :
byte **[]**
word **[]**

An open-array type is the base address of an array. This type is used as parameter for handlers (§5.9) to work with arrays of arbitrary size.

Examples:

```
byte []        # open-array type of bytes
word []        # open-array type of words
```

5.4 Constant Declarations

constantDeclaration :
const *constantName* : *standardType* := *constantExpression* *NLs*

A constant declaration introduces a constant identifier *ident* of type *standardType* as a synonym to a *constantExpression* value. A constantExpression is an expression which can be evaluated at compile time and is of a standard type.

Examples:

```
const LINE_LENGTH:byte := 80
const MAX_WORD    :word := 65535
const BUF_LENGTH  :word := 4 * LINE_LENGTH
const OK          :word := true
```

5.5 Variable Declarations

variableDeclaration :
var *variableNameList* : *type* *NLs*

A variable declaration introduces one or more variable names of a given *type* .

Examples:

```
var aByte:byte          # declaration of a byte variable "aByte".
var aWord, count:word   # declaration of two word variables
                        #             "aWord" and "count".
var anArray:word [4]    # declaration of an array variable
                        # "anArray" of 4 words.
```

5.6 Device Declarations

deviceDeclaration :
device *deviceName* : *standardType* [*atDeviceLocation*] *NLs*

atDeviceLocation :
at *constantExpression*

A device declaration introduces one device name *ident* of a given *standardType*. The device registers (memory or port mapped) can be specified to be located at an absolute device location corresponding to a hardware device. The device access in the language is completely independent of the actual device mapping (the compiler takes care of these implementation details).

Examples:

```
# declaration of a device variable "printerDataReg"
device printerDataReg:byte
# declaration of a device "printerBaseAddress" at the location 10
device printerBaseAddress:byte at 10
```

5.7 Expressions and Operators

This section describes the form of Synapse expressions. A Synapse expression is a sequence of operands which are evaluated using a set of rules which specify the order of precedence and also upon which operands the operations are to be performed. An operand can be a variable identifier, a constant, or a task expression. Evaluation of an expression yields a result (e.g., a value of a certain type) determined by the types of the operands and by the operators. Parentheses are used to modify the order in which the operators are to be applied to the operands. These subsections present operators in order of decreasing precedence (with highest precedence first).

5.7.1 Primary Expressions

primaryExpr :
- *integerConstant*
- *characterConstant*
- **true**
- **false**
- *constantName*
- *variableOrDeviceReference*
- **create** *taskName* [(*actualTaskParameterList*)]
- **name of** *taskId*
- **state of** *taskId*
- **priority of** *taskId*
- **timeslice of** *taskId*
- (*expression*)

actualtaskParameterList :
- *expressionList*

taskId :
- **id of** *taskName*
- **caller**
- **parent**
- **child**
- **self**

An integer constant, a character constant, and the keywords **true** and **false** are primary expressions. These operands have a standard type (§5.4).

A constant name is a primary expression that refers to an integer or to a character constant (§5.4).

A variable or a device reference is a primary expression that provides a name for a variable (§5.5) or a device (§5.6). Every name (or identifier) operand has an associated type.

A task is known throughout the application by two identities: its name (*taskName*) and its identification number (*taskId*).

A task's name *taskName* is assigned by the programmer during a task creation or when it is started (§5.8.9). A *taskName* is an identifier that gives a name to a task during its declaration (§5.10.1).

A unique task identification number *taskId* is assigned to a task (in the task control block) during its creation by the real-time executive. A *taskId* is an integer constant of type **byte** between 1 and 255, inclusive.

The **id of** operator returns the *taskId* of the task corresponding to the string constant that is contained in the *taskName* given at the time of its declaration(§5.10.1). An integer constant zero is returned if the named task is not found. The type of the result is byte.

The **caller** operator returns the *taskId* of the task that just sent an interrupt. This operator can only be used in a software handler during the servicing of an interrupt.

The **parent** operator returns the *taskId* of the parent task (the creator of the currently active task).

The **child** operator returns the *taskId* of the last child task created by this task.

The **self** operator returns the *taskId* of the running task.

The expression **create** creates a new task and puts it in suspended state. It can resume execution only through a schedule statement. The creation is based on the priority and the timeslice established in the task interface. A task can be created with actual parameters if required. This expression returns the *taskId* of the task created.

The expression **name of** *taskId* returns the task name of the corresponding *taskId*. The type of the result is string constant.

The expression **state of** *taskId* returns the state of the corresponding *taskId* . The type of the result is byte. See the Synapse Real-Time eXecutive (SynRTX) document for the different states of a task.

The expression **priority of** *taskId* returns the priority (§5.10.2) of the corresponding *taskId* . The type of the result is byte.

The expression **timeslice of** *taskId* returns the timeslice (§5.10.3) of the corresponding *taskId* . The type of the result is byte.

Example·

```
task MyTask in "mytask.syn"
    handler IdentifyTheCaller
        put "Interrupted by task # ", caller
        put "named : ", name of caller

    entry
        var myChildAnotherTask, myPriority:byte
        var myTimeslice:word

        # creation of a child named AnotherTask
        myChildAnotherTask := create AnotherTask

        # prints the child taskid in three different ways
        put child, id of AnotherTask, myChildAnotherTask
```

```
# prints my taskid and my parent taskid
put self, parent

# gets my priority and my time slice
myPriority := priority of self
myTimeslice := timeslice of self

# raises my priority
priority of self := myPriority + 1

# raises my time slice by 10 ticks
timeslice of self := myTimeslice + 10
```

5.7.2 Unary Operators

unaryExpr :
primaryExpr
unaryOperator typeCastExpr

unaryOperator : one of
+ - not

The resulting type of all unary operators is the standard type. The result of the unary + operator is the value of its operand. The result of the unary - operator is the negative (two's complement) of its operand. The result of the unary **not** operator is the bitwise one's complement of its operand.

Examples:

```
+   expr          is equivalent to    (0 + expr)
-   expr          is equivalent to    (0 - expr)
not expr          is equivalent to    (0 - expr - 1)
not 0b11001100    is equivalent to    0b00110011
```

5.7.3 Type-Cast Expressions

typeCastExpr :
    unaryExpr
    standardType (*typeCastExpr*)
    openArrayType (*typeCastExpr*)

Type-cast conversions are discussed in §5.7.9; standard types are discussed in §5.3.1; open-array types are discussed in §5.3.3.

5.7.4 Multiplicative Operators

multiplicativeExpr :
    typeCastExpr { *multiplicativeOperator* *multiplicativeExpr* }

multiplicativeOperator : one of
    * / **mod**

The result of the * operator is the product of the operands. The * operator is commutative and associative, and expressions involving several multiplications at the same level may be regrouped. The result of the / operator is the quotient of the operands. The result of the **mod** operator is the remainder from the division of the first operand by the second.

Example:

The expression `(x/y)*y + x mod y` is equal to `x` (if `y` is not zero)

5.7.5 Additive Operators

additiveExpr :
multiplicativeExpr { additiveOperator additiveExpr }

additiveOperator : one of
+ -

The result of the + operator is the sum of the operands. The + operator is commutative and associative, and expressions involving several additions at the same level may be regrouped. The result of the - operator is the difference of the operands.

5.7.6 Relational Operators

relationalExpr :
additiveExpr { relationalOperator relationalExpr }

relationalOperator : one of
< > <= >= = /=

Each of the operators less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) yields one (1) if the specified relation is true and zero (0) if it is false. The operators equal to (=) and not equal to (/=) yields one (1) if the specified relation is true and zero (0) if it is false. The result has standard type.

Examples:

```
9 = 8      is equal to   0
9 /= 8     is equal to   1
0 = 0      is equal to   1
0 /= 0     is equal to   0
```

5.7.7 Bitwise and Logical Operators

bitwiseOrLogicalExpr :
relationalExpr [*bitwiseOrLogicalOperator* *bitwiseOrLogicalExpr*]

bitwiseOperator : one of
shr shl and or xor

The result of the **and** operator is the bitwise "and" of the operands. Each bit in the result is set if both corresponding bits in the converted operands are set; otherwise the result bit is cleared. The **and** operator is commutative and associative. The result of the **or** operator is the bitwise inclusive "or" of the operands. Each bit in the result is set if either or both corresponding bits in the converted operands are set; otherwise the result bit is cleared. The **or** operator is commutative and associative. The result of the **xor** operator is the bitwise "exclusive or" of the operands. Each bit in the result is set if both corresponding bits in the converted operands contain opposite value (one is set, the other is cleared); otherwise the result bit is cleared. The **xor** operator is commutative and associative. The result of *expr1* **shl** *expr2* is *expr1* shifted left by the number of bits specified in *expr2*. Zeros are shifted in on the right. The result of *expr1* **shr** *expr2* is *expr1* shifted right by the number of bits specified in *expr2*. Zeros are shifted in on the left.

Examples:

```
not 0b00110011                 is equal to   0b11001100 (or 0xCC)
0b11001100 and 0b10101010      is equal to   0b10001000 (or 0x88)
0b11001100  or 0b10101010      is equal to   0b11101110 (or 0xEE)
0b11001100 xor 0b10101010      is equal to   0b01100110 (or 0x66)
0b0110000110000011 shr 1       is equal to   0b0011000011000001
0b1100011111101111 shl 2       is equal to   0b0001111110111100
```

Note that the resulting value (true or false) from a relational expression can be combined

significantly with logical operators:

```
not  ( 8 > 3 )          is equal to   0x0000 (false)
8 < 3   and   9 > 4     is equal to   0x0000 (false)
8 > 3    or   9 > 4     is equal to   0x0001 (true)
```

5.7.8 Precedence and Order of Evaluation

The precedence and associativity of Synapse operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only in the presence of other operators having higher or lower precedence. Expressions with higher precedence operators are evaluated first. An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either right to left or left to right. The following table summarizes the precedence and associativity of the Synapse operators, listing them in order of precedence from highest to lowest. Operators that appear together in a line have equal precedence and are evaluated according to their associativity.

Table 5-2 Precedence and Associativity of Synapse Operators

Symbol	Type of Operation	Associativity
`+ - not`	Unary	Right to left
`* / mod`	Multiplicative	Left to right
`+ -`	Additive	Left to right
`< > <= => = /=`	Relational	Left to right
`shr shl and or xor`	Bitwise and Logical	Left to right

5.7.9 Type Conversions

Type conversions occur:

- when a value is assigned to a variable of a different type,
- when a value is explicitly cast to another type,
- when an operator converts the type of its operand or operands before performing an operation, and
- when a value is passed as an argument to a handler.

Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. The methods of carrying out conversions depend upon the type, as given in Table 5.3.

Table 5-3 Conversions to and from Standard Types

From	To	Method
`byte`	`word`	Zero Extend
`word`	`byte`	Preserve low-order byte

An open-array to one type of value can be converted to an open-array to a different type. The result will depend of the alignment sizes of different types in storage (byte or word).

An open-array value can be converted to a standard type value, and vice-versa. The methods of carrying out conversions depend upon the type, as given in Table 5.4.

Table 5-4 Conversions to and from Open-Array Types

From	To	Method
`byte`	`byte []`	Zero Extend
`byte`	`word []`	Zero Extend
`word`	`byte []`	Preserve pattern (no change)
`word`	`word []`	Preserve pattern (no change)
`byte []`	`byte`	Preserve low-order byte
`byte []`	`word`	Preserve pattern (no change)
`byte []`	`word []`	Implementation dependent
`word []`	`byte`	Preserve low-order byte
`word []`	`word`	Preserve pattern (no change)
`word []`	`byte []`	Preserve pattern (no change)

Type-Cast Conversions

Explicit type conversions can be made by a type cast. A type cast has the form

standardType (*typeCastExpr*)
openArrayType (*typeCastExpr*)

where standardType and openArrayType specify a particular type and typeCastExpr is a value to be converted to the specified type. The conversion rules for assignments (outlined above) apply also to type casts. An openArrayType can be converted to another openArrayType. However, the result is implementation dependent, because of the alignment requirements of different types (byte or word) in storage. See the compiler documentation on openArrayType conversions.

Operator Conversions

The conversions performed by Synapse operators depend on the operator and on the type of the operand or operands. Table 5-3 lists all operator conversions.

Handler-Call Conversions

The type of conversion performed on the argument in a handler call depends on the declared parameter type for the called handler. Tables 5-3 and 5-4 list all argument conversions.

5.8 Statements

statement :
- *nullStatement*
- *assignmentStatement*
- *ifStatement*
- *loopStatement*
- *exitWhenStatement*
- *interruptStatement*
- *enableStatement*
- *disableStatement*
- *delayStatement*
- *startStatement*
- *resumeStatement*
- *suspendStatement*
- *terminateStatement*
- *rescheduleStatement*
- *ioStatement*

Statements are used to specify the flow of control through an application. The execution of these statements is achieved in a body by task entries and handlers.

As mentioned in §5.1.2, there is no need for compound statements in Synapse, since indentations perform an equivalent function.

5.8.1 Null Statements

nullStatement :
null

Its execution is an empty operation, has no effect, and always ends. This statement is useful as a filler when the syntax requires a statement.

Example: A busy loop

```
loop
    null
```

5.8.2 Assignment Statements

assignmentStatement :
variableOrDeviceReference **:=** *expression*
priority of *taskId* **:=** *expression*
timeslice of *taskId* **:=** *expression*

The expression is evaluated and the resulting value is assigned to a variable, a device, a task's priority, or a task's time slice. In an assignment (:=), the value of the expression replaces the old object specified on left. An assignment to an indexed variable of an array variable assigns a value to the indexed variable without changing the values of the remaining position(s) in the array.

Examples:

```
a := 4
anArray[2] := 5

# set bit 0 in aDevice (independently if it is
#                       memory or port mapped device)
aDevice := aDevice or 0x01
```

5.8.3 If Statements

ifStatement :
if *expression NLs statements [elsePart] [elseTruePart]*

elsePart :
[**else** *expression NLs statements]*

elseTruePart :
else *NLs statements*

Each conditional expression following each *If* and *Else* is evaluated until one of them is found to be true, in which case the statement(s) following the expression is (are) executed. An *Else* immediately followed by a new-line (*NL*) or a comment is considered to be true. If none of the expressions evaluate to true then the statement(s) following *Else* is(are) executed; if no *Else* is present the execution continues following the *If* statement. An *Else* is associated with the lexically immediately preceding aligned *If* or *Else* . For example, in

```
d := 3
e := 4
if a > 0
    if  b > c
        e := b
    else c > d
        e := c
    else
        e := d
```

all *Else* statements go with the aligned *If* , as shown by indentation. As result,

if a = 0, b = 1, c = 2 then e = 4

if a = 1, b = 1, c = 2 then e = 3

if a = 1, b = 5, c = 2 then e = 5

if a = 1, b = 1, c = 6 then e = 6

Different indentations will change the association. The indentations determines what you want:

```
d := 3
e := 4
if a > 0
    if  b > c
        e := b
    else c > d
        e := c
else
    e := d
```

Will evaluate as follows:

if a = 0, b = 1, c = 2 then e = 3

if a = 1, b = 1, c = 2 then e = 4

if a = 1, b = 5, c = 2 then e = 5

if a = 1, b = 1, c = 6 then e = 6

5.8.4 Loop Statements

loopStatement :
loop *[* **within** *integerConstant] NLs statements*

The statement(s) indented inside a loop is(are) repeated indefinitely. The **within** clause limits the iterations to no more than a some real-time units or ticks (integerConstant). If the loop times out, an exception handler will be activated. Exception handlers are discussed in §5.16.

Example: Keep getting a byte from the keyboard and printing it on the terminal ... forever.

```
var aByte:byte
loop
    get aByte
    put aByte
```

5.8.5 ExitWhen Statements

exitWhenStatement :
exit when *expression*

An Exit When statement causes an immediate exit to the nearest enclosing loop if the expression evaluates to true. Otherwise, the execution of the loop continues. An Exit When statement cannot appear outside of a loop.

Examples: Keep getting a byte from the keyboard, exit from the loop if it is 0, otherwise print it on the terminal.

```
var aByte:byte
loop
    get aByte
    exit when aByte = 0
    put aByte
```

Same as above, except that "exit from the loop" must be within the next 20 seconds (2000 x 10ms/tick).

```
var aByte:byte
loop within 2000
    get aByte
    exit when aByte = 0
    put aByte
```

5.8.6. Interrupt Statements

interruptStatement :
interrupt *taskName* . *handlerName* [(*expression* ,]
interrupt *taskId* . *handlerName* [(*expression*)]

An Interrupt statement is used to send information to any handler of any task. It is sent to the specified handlerName with an optional actualParameterList. The effect of an interrupt on a task depends on whether or not this interrupt is enabled and on its priority among all the other interrupts that are now enabled for this task. As any two interrupts

may not have equal priorities, a non-deterministic choice will never happen. If an interrupt (and possibly information) is sent to a non-enabled handler, then the interrupt will be pending (with sent information saved if any) until the handler is enabled. A handlerName selects a particular handlerId that belongs to a specific *taskId* . Each handlerName must be prefixed by a *taskName* or a *taskId* and the dot (.) operator.

Example: Send **true** to the handler Granted in task PrinterServer.

```
interrupt PrinterServer.Granted(true)
# or
interrupt id of PrinterServer.Granted(true)
```

5.8.7 Enable and Disable Statements

enableStatement :

enable hardware
enable device *constantExpressionList*
enable preemption
enable *handlerNameList*
enable *constantExpressionList*
enable wait *handlerNameList [* **within** *integerConstant]*
enable wait *constantExpressionList [* **within** *integerConstant]*

disableStatement :

disable
disable hardware
disable device *constantExpressionList*
disable preemption

Because the above statements are machine-dependent, the programmer should refer to the corresponding device numbers listed in the Synapse Real-Time eXecutive document.

The *Disable-hardware* statement disables all hardware handlers and thus prevents them from being interrupted (like the disable interrupt instruction available on microprocessors). The *Enable-hardware* statement enables all hardware handlers to be

interrupted (like the enable interrupt instruction available on microprocessors).

The *Enable-device* statement enables specific device number(s) (constantExpressionList). The *Disable-device* statement disables specific device number(s) (constantExpressionList). This statement is implementation dependent and has no effect if devices share an interrupt level. See the compiler documentation on device interrupt levels.

The *Enable/Disable-preemption* statements are used when some time-critical sections of code must be guaranteed to be executed until completion without preemption. The *Disable-preemption* statement execution assures that the scheduler will not try to preempt the calling task, until it next executes an *Enable-preemption* statement. Executing the enable preemption statement when the task is already preemptible, or the Disable-preemption statement when the task is already nonpreemptible, is permitted, and has no effect.

The *Disable* statement disables all software handlers and thus prevents them from being interrupted by other tasks.

All other *Enable* statements are used for software handler(s). The constantExpressionList represents at least one handler number specified by the order of the handler's declaration of the corresponding task (from 0 to 15, inclusive). The handlerNameList enables the software handler(s) with a priority given by the order of their appearance in the text so it is not possible to have software handlers with equal priority. The software handler(s) which is(are) enabled will remain enabled until a subsequent software enable or disable occurs. The *Enable-wait* statement behaves like the *Enable* statement, except that it blocks the current task; it then waits for the first interrupt to come along which will be processed with the lexical ordering priority given. The **within** clause limits

the iterations to no more than a number of real-time units or ticks (integerConstant). If the loop times out, an exception handler will be activated. Exception handlers are discussed in §5.16.

Examples:

```
# Enable all hardware handlers to be interrupted.
enable hardware

# Enable the Real-Time Clock device (RTC = 0)
enable device    0

# A time-critical section of code guaranteed to be
# executed to completion without preemption
disable preemption
# time-critical section of code
enable preemption

# Enable software handler 'Current' and 'Last' (where Current is
# prior to Last), then resumes execution.
# All other software handlers are disabled.
enable Current, Last

# Enable software handler 'Current' and 'Last' (where Current is
# prior to Last), block the task and relinquish the control to
# the scheduler. All other software handlers are disabled.
enable wait Current, Last

# Same as above, except that one of these software handlers
# 'Current' and 'Last' must be interrupted within the next
# 20 seconds (2000 x 10ms/tick).
enable wait Current, Last within 2000
```

5.8.8 Delay Statements

delayStatement :
delay *expression*

The delay statement allows a task to suspend itself for a given interval. The duration in increments of 10 ms per tick is given by the expression.

Example:

```
# the task is delayed (sleeping)
# for next 2 seconds (200 x 10ms/tick)
delay 200
```

5.8.9 Start Statements

startStatement :
start *startTaskNameList*

startTaskNameList :
startTaskName [, startTaskName]

startTaskName :
taskName [(actualTaskParameterList)]

actualtaskParameterList :
expressionList

The start statement creates and resumes the execution of task name(s) based on the priority and the timeslice shown in each task interface. Task(s) can be started with actual parameters if required. This statement does not reschedule the currently active (running) task.

Example:

```
# start Control task and three Reader tasks
start Control, Reader(0), Reader(1), Reader(2)
```

5.8.10 Resume Statements

resumeStatement :
resume *taskIdList*

The resume statement resumes the execution of task(s) by inserting it(them) onto the ready queue. This statement does not reschedule the currently active (running) task.

Example:

```
# create Control task and three Reader tasks
controlTid := create Control
readerNumber := 0
loop
    readerTid[readerNumber] := create Reader(readerNumber)
    readerNumber := readerNumber + 1
    exit when readerNumber = 3
...
# resume Control task and three Reader tasks
resume controlTid, readerTid[0], readerTid[1], readerTid[2]
```

5.8.11 Suspend Statements

suspendStatement :
suspend *taskIdList*

The suspend statement stops the execution of task(s) specified by *taskIdList* . A task may suspend itself, and/or any other task(s). It can release execution only by some other task(s) executing the schedule statement. This statement initiates a reschedule (i.e., a task switch).

Example: To respond to processor overload situations.

```
...
if overload
suspend id of Background
...
if recovered
schedule id of Background
...
```

5.8.12 Terminate Statements

terminateStatement :
terminate *taskIdList*

The terminate statement allows the currently active (running) task to kill another task(s). A task may terminate itself (commit suicide).

Examples:

```
# birth and death of a child
start MyChildTask
terminate child   # same as "terminate id of MyChildTask"
# suicide
terminate self
# same as the two above statements
terminate child, self
```

5.8.13 Reschedule Statements

rescheduleStatement :
reschedule

The reschedule statement moves the currently active (running) task onto the ready queue, completes a context switching, selects a new task to run, and makes this new task running. The purpose of this statement is to allow a task to give up part of its time slice.

Examples:

```
reschedule # the running task
```

5.8.14 Input and Output Statements

ioStatement :
    putStatement
    getStatement

putStatement :
    put *putItemList*

putItem :
    expression
    stringConstant

getStatement :
    get *getItemList*

getItem :
    variableReference

An I/O statement is a **get** (input) or a **put** (output) statement. A get statement reads getItem(s) from the keyboard. A getItem is a variable reference. The following input:

```
255 65535
```

is used in this example:

```
var aByte:byte
var aWord:word
get aByte, aWord
# aByte = 255, aWord = 65535
```

A put statement writes putItem(s) to the screen. A putItem is an expression or a string constant.

Examples:

```
put aByte
put 65535
put "Hello world!"
put "The answer ", 5, " is correct"
```

5.9 Real-Time Application Structure and Task Units

realTimeApplication :
taskUnit { taskUnit }

A Synapse real-time application is composed of one or more task units, where each task unit is a file that defines one task and must be compiled separately. A task is a separate "thread of control" which executes conceptually in parallel with other tasks. It is an independent active entity with a collection of statements executed strictly in sequence, which goes into action automatically as soon as the task is activated.

taskUnit :
{ NL } taskInterface taskImplementation

A task unit consists of two parts: the task interface and the task implementation.

5.10 Task Interface

taskInterface :
taskHeader [priority] [timeSlice] [importList] [exportList]

The task interface is the public (visible) part of the task unit. It contains the task header followed optionally by its priority, time slice, import, and export lists.

5.10.1 Task Header

taskHeader :
task *taskName [formalParameterList]* **in** *stringConstant NLs*

A task header gives a name (*taskName*) to a task. A task may optionally take parameters, the types of which are defined in the formal parameter list. The names of parameters, as well as the name of the task, are visible inside the task. The identifiers declared in the *formalParameterList* must be distinct from other identifiers inside the task. The task header also specifies in which physical file (*stringConstant*) this task unit is located.

5.10.2 Priority

priority :
priority *integerConstant NLs*

The task's priority is an integer constant from 1 (lowest) to 255 (highest). The priority is 1 by default.

5.10.3 Time Slice

timeSlice :
timeslice *integerConstant NLs*

In Synapse, the timeSlice is mesured in real-time units (or ticks). One real-time unit is equal to ten milliseconds. At every clock interrupt, the real-time clock driver calls the scheduler to prepare the next task ready to be executed. The task's time slice is an integer constant that corresponds to the number of real-time units during which the task should be executed before preemption. The time slice is 2 (20 milliseconds) by default, if not

specified.

5.10.4 Import List

importList :
(from taskName [asRole] import handlerNameList NLs)

asRole :
as caller
as child
as parent

A task interface should import only those handler(s), from other task units that are required in the implementation part. This reduces its dependence on other task units. The task name (also called a qualified task name) specifies the task unit whose imported handler(s) needs to be accessed (handlerNameList). Obviously, to import handlers, these must be visible (exportable) from the specified task name. This selective import facility allows the import of different handlers of the same name from different task units and still avoid conflicts of names.

The optional asRole is useful when a task statement references the target task because it is the caller, the parent or the child.

5.10.5 Export List

exportList :
(export handlerNameList NLs)

The export clause lists the software handlers (handlerNameList) that are accessible to other task units.

Example of a task interface:

```
task Reader in "reader.syn"
    priority  2
    timeSlice 10
    from Control as caller import Read
    export Read
```

5.11 Task Implementation

taskImplementation :
{ declarationInTask } taskEntry

The task implementation is the private (hidden) part of a task unit. It contains declaration(s) and a task entry which are logically and textually hidden from other task unit(s).

5.11.1 Declarations in Task

declarationInTask :
declaration
hardwareHandlerDeclaration
softwareHandlerDeclaration

declaration :
constantDeclaration
variableDeclaration
deviceDeclaration

A task implementation may have zero or more declaration(s) of a constant, a variable, a device, a hardware handler, or a software handler. The scope of declaration(s) within the task implementation extends from the task interface to the last statement in the task body. This declarative region is global to an inner (enclosed) declarative region. Inner declarative regions are optional handler declaration(s) and a mandatory task entry.

5.11.2 Task Entry

taskEntry :
    entry *NLs body*

body :
    (declaration) statements

A task entry is where the task begins its execution. A body is composed of zero or more declarations of a constant, a variable, or a device followed by at least one statement. The scope of declaration(s) within the task entry extends from the **entry** to the last statement in the body. This declarative region is local to the task implementation. A declaration is hidden within a task entry if it contains the same identifier of this declaration.

5.12 Handler Declarations

hardwareHandlerDeclaration :
    hardwareHandlerHeader body

softwareHandlerDeclaration :
    softwareHandlerHeader body

Information is transferred between tasks by sending and handling interrupts. A handler is an interrupt service routine (ISR) declared only in a task implementation dedicated to serve an interrupt. Once an interrupt call has been sent, the handler body is executed. There are two kinds of handlers: software and hardware.

hardwareHandlerHeader :
    handler *handlerName* **hardware** : *integerConstant NLs*

softwareHandlerHeader :
    handler *handlerName [formalParameter] NLs*

The software interrupt handler is a natural extension of the hardware interrupt mechanism. Only a software handler can receive an argument from interrupts (via a formal parameter). A handler behaves like a routine and returns after execution to the point of interrupt. Only software handlers can be visible (exported) to other client task, these exportations are specified in the task interface (§5.10). A hardware handler must specify the interrupt vector number (integerConstant). This is implementation-dependent; the programmer should refer to the corresponding device numbers listed in the compiler document.

Example: An interrupt-vector 0 for hardware handler DivisionByZero.

```
handler DivisionByZero hardware:0
    put "Division by Zero\n"
```

Example: A software handler Read that receives a value from a task.

```
handler Read(aValue:word)
    put "Value received from the caller", caller, "is", aValue
```

5.13 Multi-Tasking

A task is a separate "thread of control" which executes concurrently with other tasks. It is an independent active entity with a collection of statements executed strictly in sequence that goes into action automatically as soon as the task is started. A task ends by executing its last statement or by executing a Terminate statement in its body. Concurrent execution of a task is created and scheduled by the Start statement with actual parameters if required.

For example, the following are two separate files. The first file called baby.syn contains a task named Baby that prints its own babble, and the second file startup.syn is the UsersTaskStartUp task that creates and schedules two tasks Baby printing (babbling) at undefined relative speeds:

```
task Baby(babble:byte[]) in "baby.syn"
    entry
        loop
            put babble


task StartUpUserTasks in "suuser.syn"
    entry
        start Baby("Do"), Baby("Ga")
        # StartUpUserTasks is terminated at this point.
```

The output is a sequence of Do's and Ga's where each sequence is ended by the context switch of each task reaching the end of its time slice.

```
DoDoDoDoDoGaGaGaGaGaDoDoDoDoDoGaGaGaGaGaDoDoDoDoDoGaGaGaGaGa...
```

5.14 Mutual Exclusion

When tasks need to update common data, the data may be corrupted if more than one update takes place in parallel. In Synapse, handlers guarantee mutually exclusive access to global data in a task. A handler guarantees that only one task is active inside a handler at a given time. The following application shows how such an example can be safely programmed using handlers. The task Observer waits for an event, and increments the eventCount in task Update as soon as he sees one. The task Reporter waits for a while, prints and resets the eventCount in the Update task. The task Update is started first to

initialize count with the value zero and gives a fair access to the Reporter and the Observer by rotating interrupt priorities.

Example:

```
task StartUpUserTasks in "suuser.syn"
    entry
        start Update, Observer, Reporter


task Update in "update.syn"
    export Report, Observe
    var eventCount:word
    handler Report
        put eventCount
        eventCount := 0
    handler Observe
        eventCount := eventCount + 1
    entry
        eventCount := 0
        loop
            # give a fair access by rotating interrupt priorities
            enable wait Observe, Report
            enable wait Report, Observe


task Observer in "observer.syn"
    from Update import Observe
    entry
        loop
            # wait for an event ...
            interrupt Update.Observe


task Reporter in "reporter.syn"
    from Update import Report
    entry
        loop
            # wait for a while ...
            interrupt Update.Report
```

5.15 Synchronization and Communication between tasks

In most real-time applications, tasks must interact with one another. Two tasks can interact by synchronizing their activities or by passing messages between themselves. Synchronization is two tasks adjusting their relative rates of execution to meet certain timing requirements. Communication (with message passing) means that data (a message) is exchanged between two tasks. Because tasks must synchronize to exchange data, the semantics of message passing is higher level than that of synchronization.

A real-time language needs a synchronization primitive to allow tasks to communicate, while preventing them from simultaneous access to data (see mutual exclusion §5.14).

Both intertask synchronization and communication are accomplished with handler entries. A task declares its handler entries in the task implementation.

5.15.1 Asynchronous Communication

The actions taken upon the arrival of a message are very similar in behavior to the handling of an interrupt on a single processor. This type of inter-task communication is a very natural mechanism based on the importance of interrupts in real-time systems. Synapse offers asynchronous message passing as a general communication construct that should be viewed as an orthogonal form of communication.

Asynchronous Communication without data

Example: A task Receiver waits the reception of an interruption from task Sender, by means of the statement **enable wait** Sync in the task entry of Receiver:

```
task Sender in "sender.syn"
    from Receiver import Sync
    entry
        # interrupt and continue ...
        interrupt Receiver.Sync
        ...

task Receiver in "receiver.syn"
    export Sync
    handler Sync
        ...
    entry
        enable wait Sync
```

Asynchronous Communication with data

Example: A task Receiver awaits the reception of an interruption with data from another task Sender, by means of the statement **enable wait** Sync in the task entry of Receiver:

```
task Sender in "sender.syn"
    from Receiver import Sync
    entry
        var data:word
        # computation on data
        # interrupt and continue ...
        interrupt Receiver.Sync(data)
        ...

task Receiver in "receiver.syn"
    export Sync
    var receivedData:word
    handler Sync(data:word)
        receivedData := data
        ...
    entry
        enable wait Sync
```

5.15.2 Synchronous Communication

Synchronous communication requires a rendezvous between tasks which desire to transfer data (unidirectionally or bidirectionally). The sender will be blocked until the other task (the receiver) sends back an interrupt, and then both tasks continue.

Synchronous Communication without data

```
task Sender in "sender.syn"
    from Receiver import Sync
    export Reply

    handler Reply
        ...

    entry
       # interrupts and waits on Reply
       interrupt Receiver.Sync(data)
            enable wait Reply
       ...


task Receiver in "receiver.syn"
    from Sender import Reply
    export Sync

    handler Sync
        interrupt caller.Reply

    entry
        # waits on Sync
        enable wait Sync
        ...
```

Synchronous Communication with data

```
task Sender in "sender.syn"
    from Receiver import Sync
    export Reply
    var sentData:word

    handler Reply(data:word)
        sentData := data

    entry
        # interrupts and waits on Reply
        interrupt Receiver.Sync(sentData)
            enable wait Reply
        ...


task Receiver in "receiver.syn"
    from Sender import Reply
    export Sync
    var receivedData:word

    handler Sync(data:word)
        # computation on data
        receivedData := data

    entry
        # waits on Sync
        enable wait Sync
        interrupt Sender.Reply(receivedData)
        ...
```

5.16 Exception Handling

Exceptional situations are generally influenced by internal hardware interrupt mechanisms or by run-time error conditions. Exception handlers are responsible for handling the exceptions that are raised within a unit, and behave exactly like standard handlers. An exception is raised via an interrupt statement.

Example:

```
task DivisionByZero in "div0.syn"

    handler DivByZero hardware:0
        enable hardware  # allow further interrupts
        put "Division by Zero\n"
        interrupt DivisionByZero.Exception

    handler Exception
        # action after a division by zero ...

    entry
        # on entry, the interrupt hardware vector 0 is installed
        loop
            enable wait Exception
```

Chapter 6

Synapse Applications

This chapter describes some applications written in Synapse.

6.1 Producer-Consumer Problem

The application is shown diagrammatically in Figure 6-1, followed by the corresponding Synapse source code.

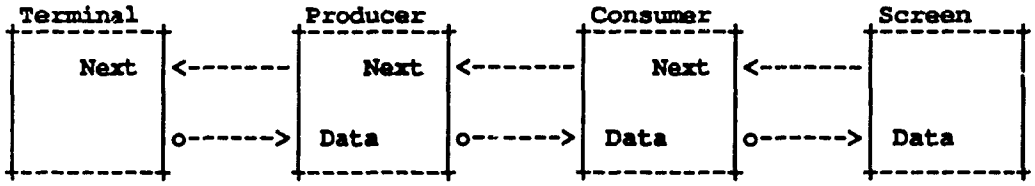

Figure 6-1 Producer-Consumer Inter-task Communication Diagram

The StartUpUserTasks starts the Screen, Consumer, Producer, and Terminal tasks (suuser.syn, line 3). The Screen task will be blocked (screen.sys, line 11) waiting for an interrupt coming from the Consumer task. The Consumer and Producer tasks will be in the same situation (consumer.syn, line 15; producer.syn, line 15) until the terminal reads a byte from the keyboard, sends it to the Producer task, and waits for a synchronization acknowledge for a Next byte (terminal.syn, lines 11, 12 and 13). Then, in the Producer task (producer.syn), the handler Data receives the byte through the input parameter b (line 9),

sends it to the Consumer task (line 10), and signals the terminal, which is the caller, for the next byte (line 11). The Consumer task (consumer.syn) receives the byte (line 9), sends it to the Screen task (line 10), signals the Producer caller task (line 11), and waits for the Screen task to signal the completion of output on the screen (line 16). The Screen task (screen.syn) receives the byte (line 5), sends it to the screen using put (line 6) and then sends the synchronizing interrupt to the Consumer task (line 7).

```
1  task StartUpUserTasks in "suuser.syn"
2      entry
3          start Screen, Consumer, Producer, Terminal
```

```
1  task Terminal in "terminal.syn"
2      from    Producer import Data
3      export  Next
4
5      handler Next
6          null
7
8      entry
9          var b:byte
10         loop
11             get b
12             interrupt Producer.Data(b)
13                 enable wait Next
```

```
1  task Producer in "producer.syn"
2      from    Consumer           import Data
3      from    Terminal as caller import Next
4      export  Next, Data
5
6      handler Next
7          null
8
9      handler Data(b:byte)
10         interrupt Consumer.Data(b)
11         interrupt caller.Next
12
13     entry
14         loop
15             enable wait Data
16             enable wait Next
```

```
1  task Consumer in "consumer.syn"
2      from    Screen              import Data
3      from    Producer as caller import Next
4      export  Next, Data
5
6      handler Next
7          null
8
9      handler Data(b:byte)
10         interrupt Screen.Data(b)
11         interrupt caller.Next
12
13     entry
14         loop
15             enable wait Data
16             enable wait Next
```

```
1  task Screen in "screen.syn"
2      from    Consumer as caller import Next
3      export  Data
4
5      handler Data(b:byte)
6          put b
7          interrupt caller.Next
8
9      entry
10         loop
11             enable wait Data
```

6.2 Bounded Buffer Problem

The application is shown diagrammatically in Figure 6-2, followed by the corresponding Synapse source code.

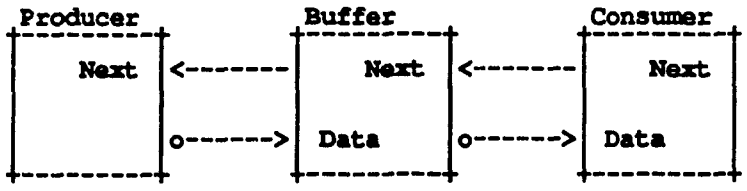

Figure 6-2 Bounded Buffer Inter-task Communication Diagram

The StartUpUserTasks starts the Buffer, Consumer, and Producer tasks (suuser.syn, line 3). The Buffer task (buffer.syn) initializes its variables (lines 21, 22, and 23), and waits for an interrupt coming from the Producer task (producer.syn, line 10). The Consumer sends an interrupt to Buffer task (consumer.syn, line 10). This interrupt will be pending (buffered) since the Buffer task has not yet enabled the handler Next. With the next statement, it gets blocked (consumer.syn, line 11) waiting for an interrupt that will come from Buffer task (buffer.syn, line 16) as soon as a datum will be available. The Producer task (producer.syn) will be blocked (line 11) waiting to serve an interrupt in handler Next (line 5). When the Producer task produces a byte b, it sends b via interrupt to the Buffer task (producer.syn, line 10). Then, the handler Data (buffer.syn, line 9) receives the byte b and stores it in the buffer (lines 10, 11, and 12). If the buffer if full, then it enables only the handler Next (line 26), and waits until the consumer releases a free slot in the buffer (lines 16, 17, and 18). If the buffer is empty (line 27), then it enables only the handler Data (line 28), and waits until the producer stores a byte in the buffer (lines 10, 11, and 12).

```
1  task StartUpUserTasks in "suuser.syn"
2      entry
3          start Buffer, Consumer, Producer
```

```
1  task Producer in "producer.syn"
2      from    Buffer import Data
3      export  Next
4
5      handler Next
6
7      entry
8          loop
9                  # produces b
10                 interrupt Buffer.Data(b)
11                 enable wait Next
```

```
1  task Consumer in "consumer.syn"
2      from    Buffer import Next
3      export  Data
4
5      handler Data(b:byte)
6          # consumes b
7
8      entry
9          loop
10             interrupt Buffer.Next
11             enable wait Data
```

```
1  task Buffer in "buffer.syn"
2      from Producer as caller import Next
3      from Consumer as caller import Data
4      export Next, Data
5      const NUMBER_OF_SLOTS:byte := 20
6      var buffer:byte[NUMBER_OF_SLOTS]
7      var nFull, slotToFill, slotToEmpty:byte
8
9      handler Data(b:byte)
10         buffer[slotToFill] := b
11         nFull := nFull + 1
12         slotToFill := (slotToFill+1) mod NUMBER_OF_SLOTS
13         interrupt caller.Next
14
15     handler Next
16         interrupt caller.Data(buffer[r])
17         slotToEmpty := (slotToEmpty+1) mod NUMBER_OF_SLOTS
18         nFull := nFull - 1
19
20     entry
21         nFull := 0
22         slotToFill := 0
23         slotToEmpty := 0
24         loop
25             if nFull = NUMBER_OF_SLOTS  # buffer full ?
26                 enable wait Next
27             else nFull = 0              # buffer empty ?
28                 enable wait Data
29             else
30                 enable wait Data, Next
```

6.3 Reader-Writer Problem

The application is shown diagrammatically in Figure 6-3, followed by the corresponding Synapse source code.

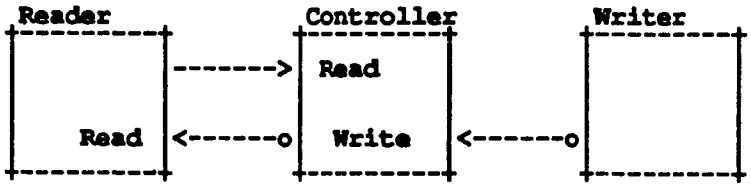

Figure 6-3 Reader-Writer Inter-task Communication Diagram

```
task StartUpUserTasks in "suuser.syn"
    entry
        start Controller, Reader, Writer
```

```
task Reader in "reader.syn"
    from   Control import Read
    export Read
    var    value:word

    handler Read(v:word)
           value := v

    entry
        loop
            # ...
            interrupt Control.Read
                enable wait Read
            # ...
```

```
task Writer in "writer.syn"
    from Control import Write

    entry
        var  value:word

        value := 0
        interrupt Control.Write(value)
        loop
            # ...
            value := value + 2
            interrupt Control.Write(value)  # update the value
            # ...

task Control in "control.syn"
    from   Reader as caller import Read
    export Write, Read
    var    value:word

    handler Write(v:word)
        value := v

    handler Read
        interrupt caller.Read(value)

        entry
            loop
                enable wait Read,  Write
                enable wait Write, Read
```

6.4 Basic Resource Device Driver

Tasks competing for shared resources must synchronize their accesses. Once a resource is acquired by a task, another task claiming the resource should be blocked until the owner task releases it. The following figure illustrates a basic resource device driver. Each task must follow the same access protocol: acquire the resource, then use it, and finally release it.

```
Acquire { Use } Release
```

The task Resource blocks itself by the enable wait Acquire when the resource is available. The Acquire handler keeps the task identification (the caller) in owner, and acknowledges it by sending an interrupt to the caller's Grant handler. The owner can used the resource, and it is the sole task that can release it. The Release handler verifies whether the task that attempts to release the resource is really the owner, then the resource is freed by assigning NOBODY to owner; otherwise, an exception is raised to indicate which task does not respect the access protocol.

The application is shown diagrammatically in Figure 6-4, followed by the corresponding Synapse source code.

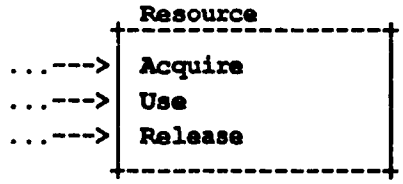

Figure 6-4 Basic Resource Diagram

```
task Resource in "resource.syn"
    export Acquire, Release
    const  NOBODY:byte := 0
    var    owner:byte

    handler Acquire
        owner := caller
        interrupt caller.Grant

    # Use ...

    handler Release
        if owner = caller
            owner := NOBODY
        else
            interrupt self.Exception(caller)
```

```
handler Exception(who:byte)
    # who is the task that has accessed the keyboard without
    # using the access protocol:  Acquire  { Use }  Release

entry
    owner := NOBODY
    loop
        enable wait Acquire
        loop
            # the owner can now used the resource ...
            # only the client (owner) can release it
            enable wait Release, Exception
            exit when owner = NOBODY
```

6.5 Event Timer Problem

The event timer problem [Kerr84] is one timer resource which services several tasks competing for the timer. These tasks have priority so a higher priority task can pre-empt a lower priority task. The pre-empted task is informed of the time which has elapsed up to the interruption. The Clock task generates a software tick triggered by a real-time clock (served by the hardware handler HardwareTick).

The application is shown diagrammatically in Figure 6-5, followed by the corresponding Synapse source code.

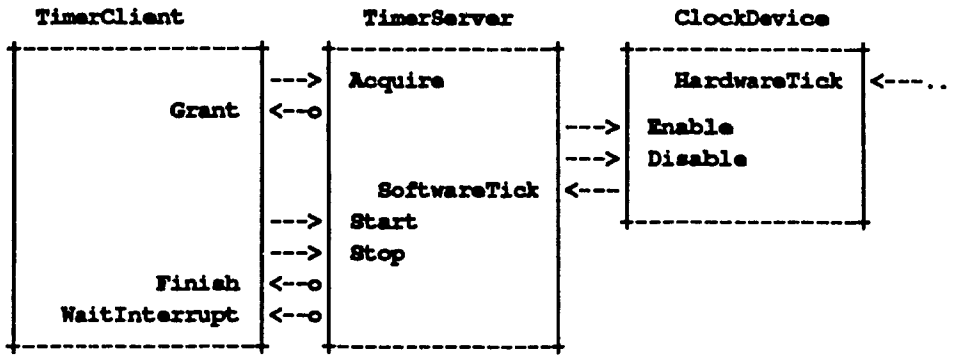

Figure 6-5 Event Timer Inter-task Communication Diagram

```
task StartUpUser in "suuser.syn"
    entry
        const    STOP_TIMER:word     := 0
        const    WAIT_INTERRUPT:word := 1


        start ClockDevice, TimerServer

        # start 4 timer clients with priority 1, 2, 3, and 4
        start TimerClient(WAIT_INTERRUPT) # already prio 1
        start TimerClient(STOP_TIMER)
        priority of child := 2
        start TimerClient(WAIT_INTERRUPT)
        priority of child := 3
        start TimerClient(STOP_TIMER)
        priority of child := 4
```

```
task TimerClient(actionToDo:word) in "tmclient.syn"
    const    STOP_TIMER:word     := 0
    const    WAIT_INTERRUPT:word := 1
    var      stopped, granted:word
    export   Start, Grant, Finish, Interrupt
```

```
    handler Start
        stopped := false
        put "Started by ", name of self, "\n"

    handler Grant(aGrant:word)
        granted := aGrant

    handler Finish(elapsed:word)
        stopped := true
        put "Stopped by ",  name of self
        put " elapsed time: ", elapsed, "\n"

    handler Interrupt(elapsed:word)
        stopped := true
        put "Stopped by another task, elapsed time: ", elapsed

    entry
        loop
            loop
                interrupt TimerServer.Acquire
                    enable wait Grant
                exit when granted
                # wait for a while and retry ...

            interrupt TimerServer.Start
            enable Interrupt
            # do something ...

            loop
                exit when stopped
                if actionToDo = STOP_TIMER
                    interrupt TimerServer.Stop
                        enable wait Finish, Interrupt
                else
                    enable wait Interrupt



task TimerServer in "tmserver.syn"
    priority 5
    from   ClockDevice           import Enable, Disable
    from   TimerClient as owner  import Interrupt
    from   TimerClient as caller import Finish, Grant
    export Acquire, Start, Stop, SoftwareTick
    const  NOBODY:word := 0
    var    elapsed, running:word
    var    owner:byte
```

```
handler Acquire
    if running  # a timer is already acquired
        if priority of caller > priority of owner
            # send the elapsed time to the current owner
            interrupt owner.Interrupted(elapsed)

            # stop and reset the timer
            running := false
            elapsed := 0

            # give it to the caller
            owner := caller
            interrupt caller.Grant(true)
        else
            interrupt caller.Grant(false)
    else
            owner := caller
            elapsed := 0
            interrupt caller.Grant(true)


handler Start
    if owner = caller
        running := true
        interrupt ClockDevice.Enable
    else
        interrupt self.Exception(caller)


handler Stop
    if owner = caller
        running := true
        owner := NOBODY
        interrupt ClockDevice.Disable
        interrupt caller.Finish(elapsed)
    else
        interrupt self.Exception(caller)


handler SoftwareTick
    elapsed := elapsed + 1


handler Exception(who:byte)
    put "Illegal access by ", name of who, "\n"
```

```
    entry
        running := false
        owner := NOBODY  # in case of illegal access
        loop
            loop
                enable wait Acquire
                exit when owner = caller

            loop
                enable wait Start
                exit when running

            # timer can be stopped by the owner, or
            # acquired by another timer client of higher priority
            loop
                enable wait Stop, SoftwareTick, Acquire
                exit when not running




task ClockDevice in "clockdev.syn"
    priority 5
    from   Timer import SoftwareTick
    export Enable, Disable
    var    enabled:word


    handler HardwareTick hardware:8
        if enabled
            interrupt Timer.SoftwareTick


    handler Enable
         enabled := true


    handler Disable
         enabled := false


    entry
         enabled := false
         loop
             enable wait Enable, Disable
```

6.6 Robot Arm Controller

The robot arm controller is shown diagrammatically in Figure 6-6, followed by the corresponding Synapse source code.

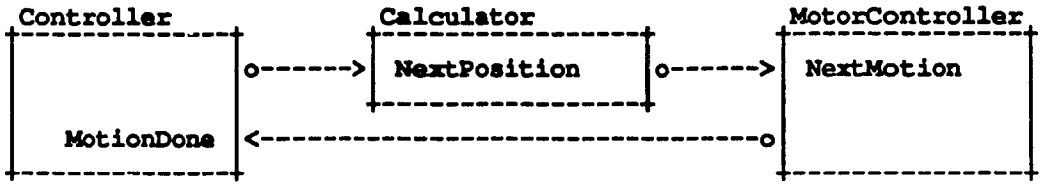

Figure 6-6 Robot Arm Controller Inter-task Communication Diagram

The robot arm allows real parallelism of operation within the movements of the arm. This one has three degrees of freedom, each of which is manipulated by a separate stepper motor which needs the direction and the number of steps to move [Kerr84]. The robot arm can be moved to any position in the x, y and z directions. The home position (0,0,0) indicates that the arm is vertically above the base at its maximum extension. The Controller task accepts new coordinates from the operator and moves the appropriate robot arm by sending the position to 3 joint motion calculator tasks.

```
task Controller in "controlr.syn"
    export MotionDone
    const  X:word = 0
    const  Y:word = 1
    const  Z:word = 2
    var    moved, pos, calcId: byte[3]


    handler MotionDone(jointNumber:byte)
        moved[jointNumber] = true
```

```
entry
    start Calculator(X)
    calcId[X] := child
    start Calculator(Y)
    calcId[Y] := child
    start Calculator(Z)
    calcId[Z] := child
    loop
        # get position from the operator
        get pos[X], pos[Y], pos[Z]

        # compute motion for each motor in parallel
        loop
            moved[X] := false
            moved[Y] := false
            moved[Z] := false
            interrupt calcId[X].NextPosition(pos[X])
            interrupt calcId[Y].NextPosition(pos[Y])
            interrupt calcId[Z].NextPosition(pos[Z])

        # wait for acknowlegment
        loop
            enable wait MotionDone
            exit when moved[X] and moved[Y] and moved[Z]

        put "Arm is at:"
        put pos[X], ":", pos[Y], ":" pos[Z], "\n"
```

```
task Calculator(jointNumber:byte) in "calc.syn"
    export  NextPosition
    from    MotorController as child import NextMotion
    var     step, direction, oldPosition, newPosition:byte
    var     motion:word


    handler NextPosition(nextPos:byte)
        nextPosition := nextPos
```

```
    entry
        oldPosition := 0
        start MotorController(jointNumber)

        loop
            enable wait NextPosition

            # compute next motor step
            # and direction from old and new positions
            motion := step shl 8    and    direction

            # motion is a word, where:
            #    - step       is the Most  Significant Byte
            #    - direction is the Least Significant Byte
            oldPosition := newPosition
            interrupt child.NextMotion(motion)




task MotorController(jointNumber:byte) in "motor.syn"
    from Controller import MotionDone
    export NextMotion


    handler NextMotion(motion:word)
        # Move motor according to step and direction (motion)


    entry
        loop
            enable wait NextMotion
            interrupt Controller.MotionDone(jointNumber)
```

Chapter 7

Conclusion

7.1 Experience and State of the Implementation

The Synapse environment is currently composed of an executive, a compiler, and an editor.

7.1.1 Syntax-Directed Editor

A syntax-directed editor called SynEd (Synapse Editor) is now under development. Its sole purpose is to be a helpful companion to the programmer by providing an environment to develop syntaxically correct Synapse task units. It identifies task structure and statement context with correction proposal on error detection. It also provides an on-line context-sensitive help facilities, an automatic indentation of control structures, and a complete set of basic editing functions. A first release has been implemented and integrates the lexical scanner and the parser of Synapse. SynEd development is funded in part by the Department of National Defense of Canada (DND).

7.1.2 Real-Time Executive

A real-time executive called SynRTX (Synapse Real-Time eXecutive) has been implemented to provide all services needed by the language. It is completly written in the C language (4000 lines), except few modules in assembly language (400 lines). The actual implementation runs on IBM PCs or compatibles. SynRTX has been used since November 1988 as teaching tool for better understanding of real-time concepts and to help writing device drivers in several undergraduate courses at College Militaire Royal (CMR) de St-Jean. Students have been writing their real-time applications by using the following software development methods:

- draw a complete inter-task communication diagram of all tasks involved in the application (as Figure 5-3),
- for each task, write the equivalent task unit in Synapse code,
- translate by hand (with all the techniques shown in class) the Synapse code into the C language, with the appropriate SynRTX system calls.[1]

SynRTX makes no use of the IBM PC/BIOS functions, because they are not reentrant. So, various device drivers (polled and interrupted driven) have been written for kinds of interface: serial (RS232C), parallel (printer), and keyboard. A window management has been implemented reusing the windowing primitives of SynEd. Floppy disk and StarLAN device drivers are now under development.

1 This translation was required because the Synapse compiler was still under development at that time

7.1.3 Compiler

The Synapse compiler has two kind of input files: task interface file suffixed by `.int`, and a task unit file suffixed by `.syn`. The compiler translates the Synapse source code in C source code by inserting all the proper system calls of the run-time Synapse Real-Time eXecutive (SynRTX). Using C as intermediate language makes Synapse portable to any environment having a C compiler. A C compiler/linker will generate an executable module ready to be loaded onto the target system for execution.

After proof of proper handling of several real-time and concurrent classical problems (§6), the Synapse language definition was stable enough to start writing a compiler in February 1989. The lexical scanner (written in C) and the parser (using YACC) were completed by March proving that the language's grammar was not ambiguous. A complete compiler with C code generation is expected to be ready soon.

7.2 Final Remarks

This thesis has described a new real-time programming language called Synapse. In particular, the language design concentrated on simple explicit constructs express without ambiguous semantics.

In programming language design, several successful designers have defined rules, methods, and principles to produce a better language. Based on these principles, they have reached common agreement on several general goals.

The design of a real-time programming language needs (with the above goals) specific requirements. They are task control management, timing constraints, and hardware en-

vironment.

A survey on these requirements shows that Real-Time Euclid is the most qualified real-time programming language (with a total of 12/15). All others are nearly system programming languages (less than 10.5/15) since they have several weaknesses in real-time.

A complete language report on Synapse mentions its general goals and how the language supports the above requirements. A section informally presents basic terms and concepts that differentiate Synapse from other real-time languages. The method of description and the syntax notation of Synapse follows a brief language summary. The last section contains the formal definition of both syntax and semantics of the language.

Several applications describes the language's capability to test and check the validity of the design requirements. A large set of applications exercices each construct of the language, specially those related to the executive.

Even if there is still place for further experimentations with the language, all feedback from users so far has been very positive. They agreed that Synapse is a useful potential real-time language for small embedded real-time applications and writing device drivers.

7.3 Future Directions

The Synapse project is available to any people interested to use it in a real-time application. Performance measurements will be done for all SynRTX system calls. Research in new debugging techniques for multitasking will be explored to offer a better aid to developpers of real-time software. A plan for rapid porting on a 16-bit or 32-bit embedded controller has been set up to provide another test bed for the experimentation of Synapse.

Appendix A

The Synapse Language Syntax Summary

This summary of Synapse syntax is intended to be an aid to comprehension. It is a recapitulation of grammar that was given throughout the report.

A.1 Method of Description and Syntax Notation

The form of Synapse task units is described by line-oriented syntax with context-dependent requirements expressed by narrative rules and indentations. Synapse uses indentation from the left to right margin and line separation to identify the task structure and statement context, to give a clean representation by enhancing the readability of the code. The terminal symbol *NL* (new-line) is a special symbol in the Synapse character set. One or more indentations take effect only if they immediatly follow a *NL*. The pattern of indentation is crucial for the interpretation of Synapse by the compiler.

The syntax of Synapse is described in a variant of BNF (Backus/Naur Form). The syntactic symbols are of two kinds: terminals and non-terminals.

The **boldface typewriter** style identifies terminal symbols including keywords and single characters (or special symbols).

Examples:

```
task  priority  import  delay  var
```

The *italic* style identifies non-terminal symbols. These symbols represent names using upper and lower case letters.

Examples:

realTimeApplication taskUnit variableDeclaration expression

A syntax rule is a non-terminal followed by a colon to introduce its definition.

Example:

delayStatement :
    delay *expression*

An alternative syntax rule lists its definitions on separate lines, except when the word "one of" followed the colon.

Example of equivalent rules:

multiplicativeOperator :
    *****
    /
    mod

multiplicativeOperator : one of
    *** / mod**

Braces *{* and *}* enclose a repeated symbol. The symbol may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule.

Example of equivalent rules:

realTimeApplication :
    taskUnit { taskUnit }

realTimeApplication :
    taskUnit
    realTimeApplication taskUnit

Brackets *[* and *]* enclose an optional terminal or non-terminal symbol.

Example of equivalent rules:

deviceDeclaration :
    device *deviceName* : *standardType [atDeviceLocation] NLs*

deviceDeclaration :
    device *deviceName* : *standardType NLs*
    device *deviceName* : *standardType atDeviceLocation NLs*

The syntax does not explicitly defined the following non-terminal symbols. They are equivalent to *identifier* (§5.2.2):

constantName variableName deviceName
handlerName taskName parameterName

The syntax does not explicitly defined the following non-terminal symbols. They correspond to the general rule of a *nonTerminalList* given below:

variableNameList handlerNameList expressionList constantExpressionList
taskIdList putItemList getItemList

nonTerminalList :
    nonTerminal { , nonTerminal }

A.2 Lexical Syntax Grammar

characterSet: one of
letter digit specialCharacter SP NL

letter: one of
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z _

digit: one of
0 1 2 3 4 5 6 7 8 9

specialCharacter: one of
' ( ) * + , - . / : < = > [ ] " \

SP:
the space character

NL:
the new-line character

symbol: one of
separator delimiter identifier keyword constant comment

separator: one of
SP NL

delimiter: one of
' ( ) * + , - . / : < = > [ ] " \ := /= >= <=

identifier:
letter { identifierCharacter }

identifierCharacter: one of
letter decimalDigit _

keyword: one of

and	**else**	**loop**	
as	**enable**		**self**
at	**entry**		**shl**
	exit	**mod**	**shr**
	export		**start**
byte			**state**
		name	**suspend**
	false	**not**	
	from	**null**	**task**
			terminate
caller			**timeslice**
child	**get**	**parent**	**true**
const		**preemption**	
create	**handler**	**priority**	**var**
	hardware	**put**	
			wait
	id	**of**	**within**
	if	**or**	**when**
delay	**import**		**word**
device	**in**	**reschedule**	
disable	**interrupt**	**resume**	**xor**

constant: one of
integerConstant *stringConstant*

integerConstant: one of
binaryConstant *decimalConstant* *hexadecimalConstant*

binaryConstant:
0b *binaryDigit* { *binaryDigit* }

decimalConstant:
digit { *digit* }

hexadecimalConstant:
0x *hexadecimalDigit* { *hexadecimalDigit* }

binaryDigit: one of
0 **1**

hexadecimalDigit: one of
digit **a** **b** **c** **d** **e** **f** **A** **B** **C** **D** **E** **F**

stringConstant :
" *stringCharacterSequence* "

stringCharacterSequence :
stringCharacter [*stringCharacter*]

stringCharacter :
any character in the source character set or escapeSequence
except double-quote ("), backslash (\ ), and *NL*

escapeSequence : one of
\' \" \\ \f \n \r \t \v \0

comment :
anyCharactersExceptEndOfLineCharacter NLs

A.3 Language Syntax Grammar

realTimeApplication :
taskUnit { *taskUnit* }

taskUnit :
{ *NL* } *taskInterface* *taskImplementation*

taskInterface :
taskHeader [*priority*] [*timeSlice*] [*importList*] [*exportList*]

taskImplementation :
{ *declarationInTask* } *taskEntry*

taskHeader :
task *taskName* [*formalParameterList*] **in** *stringConstant* *NLs*

priority :
priority *integerConstant* *NLs*

timeSlice :
timeslice *integerConstant* *NLs*

importList :
{ **from** *taskName* [*asRole*] **import** *handlerNameList* *NLs* }

asRole :
as caller
as child
as parent

exportList :
{ **export** *handlerNameList* *NLs* }

NLs :
NL { *NL* }

declarationInTask :
deciaration
hardwareHandlerDeclaration
softwareHandlerDeclaration

declaration :
    constantDeclaration
    variableDeclaration
    deviceDeclaration

taskEntry :
    entry *NLs body*

body :
    { declaration } statements

constantDeclaration :
    const *constantName* : *standardType* := *constantExpression NLs*

type : one of
    standardType arrayType

standardType : one of
    byte word

arrayType :
    standardType [*constantExpression*]

variableDeclaration :
    var *variableNameList* : *type NLs*

deviceDeclaration :
    device *deviceName* : *standardType [atDeviceLocation] NLs*

atDeviceLocation :
    at *constantExpression*

hardwareHandlerDeclaration :
    hardwareHandlerHeader body

softwareHandlerDeclaration :
    softwareHandlerHeader body

hardwareHandlerHeader :
    handler *handlerName* **hardware** : *integerConstant NLs*

softwareHandlerHeader :
    handler *handlerName [formalParameter] NLs*

primaryExpr :
integerConstant
characterConstant
true
false
constantName
variableOrDeviceReference
create *taskName* [(*actualTaskParameterList*)]
name of *taskId*
state of *taskId*
priority of *taskId*
timeslice of *taskId*
(*expression*)

variableOrDeviceReference :
variableName [[*expression*]]
deviceName

taskId :
id of *taskName*
caller
parent
child
self

expression :
unaryExpr

unaryExpr :
primaryExpr
unaryOperator typeCastExpr

unaryOperator : one of
+ - not

typeCastExpr :
unaryExpr
standardType (*typeCastExpr*)
openArrayType (*typeCastExpr*)

multiplicativeExpr :
castExpr { *multiplicativeOperator multiplicativeExpr* }

multiplicatveOperator : one of
*** / mod**

additiveExpr :
multiplicativeExpr { *additiveOperator additiveExpr* }

additiveOperator: one of
    + -

relationalExpr:
    additiveExpr [*relationalOperator relationalExpr*]

relationalOperator: one of
    < > <= >= = /=

bitwiseOrLogicalExpr:
    relationalExpr [*bitwiseOrLogicalOperator bitwiseOrLogicalExpr*]

bitwiseOperator: one of
    shr shl and or xor

statements:
    statement NLs [*statement NLs*]

statement:
    nullStatement
    assignmentStatement
    ifStatement
    loopStatement
    exitWhenStatement
    interruptStatement
    enableStatement
    disableStatement
    delayStatement
    startStatement
    resumeStatement
    suspendStatement
    terminateStatement
    rescheduleStatement
    ioStatement

nullStatement:
    null

assignmentStatement:
    variableOrDeviceReference := *expression*
    priority of *taskId* := *expression*
    timeslice of *taskId* := *expression*

ifStatement:
    if *expression NLs statements* [*elsePart*] [*elseTruePart*]

elsePart:
    [**else** *expression NLs statements*]

elseTruePart :
else *NLs statements*

loopStatement :
loop *[* **within** *integerConstant] NLs statements*

exitWhenStatement :
exit when *expression*

interruptStatement :
interrupt *taskName . handlerName [(expression)]*
interrupt *taskId . handlerName [(expression)]*

enableStatement :
enable hardware
enable device *constantExpressionList*
enable preemption
enable *handlerNameList*
enable *constantExpressionList*
enable wait *handlerNameList [* **within** *integerConstant]*
enable wait *constantExpressionList [* **within** *integerConstant]*

disableStatement :
disable
disable hardware
disable device *constantExpressionList*
disable preemption

delayStatement :
delay *expression*

startStatement :
start *startTaskNameList*

startTaskNameList :
startTaskName { , startTaskName }

startTaskName :
taskName [(actualTaskParameterList)]

actualtaskParameterList :
expressionList

resumeStatement :
resume *taskIdList*

suspendStatement :
suspend *taskIdList*

terminateStatement :
    terminate *taskIdList*

rescheduleStatement :
    reschedule

ioStatement :
    putStatement
    getStatement

putStatement :
    put *putItemList*

putItem :
    expression
    stringConstant

getStatement :
    get *getItemList*

getItem :
    variableReference

constantExpressionList :
    constantExpression [, *constantExpression*]

constantExpression :
    expression

formalParameterList :
    (*parameter* [, *parameter*])

formalParameter :
    (*parameter*)

parameter :
    parameterName : *standardType*
    parameterName : *openArrayType*

openArrayType :
    byte []
    word []

References

Ande86 Anderson, T.L., The Scope of Imported Identifers in Modula-2, ACM Sigplan Notices, vol. 21, no 9, September 1986.

ANSI83 Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, Washington, D. C.: U.S. DoD., Jan. 1983.

ARTE87 Ada Run Time Environment Working Group (ACM SIGAda), A Catalog of Interface Features and Options for the Ada Run Time Environment, Release 2.0, 1987.

Appe82 Appelbe, W.F., Ravn A.P., Encapsulation Constructs in Systems Programming Languages, Dept. of EECS, UCSD, Computer Science Tech. Report No. CS-057, August 1982.

Appe85 Appelbe, W.F., Hansen K., A Survey of Systems Programming Languages: Concepts and Facilities, Software Practice and Experience, vol. 15, no 2, February 1985.

Belz86 Belzile, C., Coulas, M., MacEwen, G. H., and Marquis G., RNet: A Hard Real-Time Distributed Programming System, Proc. of the IEEE Real-Time System Symposium, 1986.

Brin75 Brinch Hansen, P., The Programming Language Concurrent Pascal, IEEE Trans. Software Eng., SE-1, 199-207, 1975.

Brin82 Brinch Hansen, P., Programming a Personal Computer, Prentice-Hall, 1982.

CCIT80a CCITT, Introduction to CHILL, the CCITT High Level Programming Language, 1980.

CCIT80b CCITT, The CCITT High Level Programming Language, 1980.

CCIT83 CCITT, The CCITT High Level Programming Language Report, 1983.

DOD78 U.S. Department of Defense, Requirements for High Order Computer Programming Languages, "Steelman", June 1978.

DOD80 U.S. Department of Defense, "Stoneman": Requirements for Ada Programming Support Environment, Feb. 1980.

Evan81 Evans, A. Jr, et.al., Praxis Language Reference Manual. UCRL-15331, Lawrence Livermore National Laboratory, 1981.

Feld86 Feldman, M.B., Ada vs. Modula-2: a Response from the Ivory Tower, ACM Sigplan Notices, vol. 21, no 5, May 1986.

Geha86 Gehani, N. H., Roome, W. D., Concurrent C, Software Practice and Experience, vol. 16, 821-844, 1986.

Ghez87 Ghezzi, C., Jazayeri, M., Programming Language Concepts, 2nd ed., John Wiley & Sons, 1987.

Glig83 Gligor, V. D. and Luckenbaugh, G. L., An Assessment of the Real-Time Requirements for Programming Environments and Languages, Proc. of the IEEE Real-Time System Symposium, 1983.

Gree86 Greenwood, J.R., Comments on "A View from the trenches" Ada vs Modula-2 vs Praxis. ACM Sigplan Notices, vol. 21, no 5, May 1986.

Hoar73 Hoare, C.A.R. Hints on Programming Language Design, Report CS403, Stanford University, Stanford, Calif. October 1973.

Hoar74 Hoare, C.A.R. Monitors: An Operating System Structuring Concept, Comm. ACM 17 10, Oct. 1974. Corrigendum: February 75.

Holt83 Holt, R.C., Concurrent Euclid, The Unix System, and Tunis, Addison-Wesley, 1983.

Holt83b Holt, R.C., and Cordy, J.R., The Turing Language Report, Tech. Report CSRI-153, CSRI, U. of Toronto, Dec. 1983.

Holt88 Holt, R.C., Matthews, P.A., Rosselet, J.A., Cordy, J.R. The Turing Programming Language Design and Definition. Prentice-Hall, 1988.

Hopp86 Hoppe, J., Another Approach to the Implementation of Synchronization Primitives, Software Practice and Experience, vol. 16, no 12, December 1986.

Howa76a Howard, J. H., Proving Monitors,
Comm. ACM, vol. 15, no 5, May 1976.

Howa76b Howard, J. H., Signaling in Monitors,
Proc. of the 2nd Intl. Conference on Soft. Eng., IEEE cat. no. 76CH1125-4C, Oct. 1976.

Ichb79 Ichbiah, J. et al. Rationale for the design of the Ada programming language, ACM Sigplan Notices, vol. 14, no 6, June 1979.

Inmo84 Inmos Limited, OCCAM Programming Manual,
Prentice-Hall, 1984.

Inte85a Intel Corporation, PL/M-86 User's Guide,
Literature Department, September 85.

Inte85b Intel Corporation, Implementing StarLAN with the Intel 82588 Controller, AP-236 Literature Department, September 85.

Jens75 Jensen, K., Wirth, N. Pascal User Manual and Report,
2nd ed., Springer-Verlag, 1975.

Jons87 Jonsson, D., Pancode and Boxcharts: Structured Programming Revisited,
SIGPLAN Notices, vol. 22, no 8, 1987.

Kern78 Kernighan, B.W., Richie D.M., The C programming Language,
Prentice-Hall, 1978.

Kerr84 Kerredge, J. M., Simpson D., Three Solutions for a Robot Arm Controller Using Pascal-Plus, Occam, and Edison,
Software Practice and Experience 14, 1984, 3-15.

Klig86 Kligerman, E., Stoyenko, A., Real-Time Euclid: A Language for Reliable Real-Time Systems,
IEEE Trans. on Software Eng. SE, vol. 12, no 9, 1986.

Kung85 Kung, A., Kung, R.,
Galaxy: A Distributed Real-Time OS Supporting High Availability,
Proc. of the IEEE Real-Time System Symposium, 1985.

Lee84 Lee, I., A Programming System for Distributed Real-Time Applications,
Proc. Real-Time Systems Symposium, 1984.

Lee85 Lee, I., Gehlot, V.,
Language Constructs for Distributed Real-Time Programming,
Proc. Real-Time Systems Symposium, 1985.

Lein80 Leinbaugh, D. W., Indenting for the Compiler,
SIGPLAN Notices, vol. 15, no 5, 1980.

Ligh82 Light, R. A., A Real-Time Executive for Multiple Microprocessor Systems.
Proc. Real-Time Systems Symposium, 1982.

MacL87 MacLennan, B.J., Principles of Programming Languages:
Design, Evaluation, and Implementation,
2nd ed., Holt, Rinehart and Winston, 1987.

Mart78 Martin, T., Real-Time Programming Language PEARL -
Concepts and Characteristics,
Proc. COMPSAC, pp. 301-306, Chicago, 1978.

Mok84 Mok, A. K., The Design of Real-Time Programming System
Based on Process Models,
Proc. Real-Time Systems Symposium, 1984.

Myer78 Myers, G., Composite/Structured Design,
Van Nostrand Reinhold, 1978.

Parr88 Parrish, L., Running in Real Time: A Problem for Ada,
Defense Computing, Sept.-Oct. 1988.

Pete85 Petermann, U., Szalas, A.,
A Note on PCI: Distributed Processes Communicating by Interrupts,
SIGPLAN Notices, vol. 20, no 3, March 1985.

Pope77 Popek, G.J., Horning, J. J., Lampson, B.W., Mitchell, J.G., London, R.L,
Notes on the design of Euclid,
Proceedings of the ACM Conference on Language Design for
Reliable Software, ACM Sigplan Notices vol. 12, no 3, March 1977.

Risi88 Rising, L., Tasking Troubles and Tips,
SIGPLAN Notices, vol. 23, no 8, 1988.

Smed83 C.H. Smedema, C.H., Medema P., Boasson, M.
The Programming Languages: Pascal, Modula, CHILL, and Ada,
Prentice-Hall International, 1983.

Snoo78 Snook, T., Report on the Programming Language PLZ/SYS, Springer-Verlag, 1978.

Spec72 Spector, D., Ambiguities and insecurities in Modula-2, ACM Sigplan Notices, vol. 17, no 8, August 1972.

Stan88 Stankovic, J., Real-Time Computing Systems: The Next Generation, Tech. Report TR-88-06, COINS Dept., U. of Massachusetts, Jan. 1988.

Trem85 Tremblay, J.P., Sorenson, P.G., The Theory and Practice of Compiler Writing, McGraw-Hill, 1985.

VRTX86 VRTX/86 User's Guide, Hunter and Ready, 1985.

Wirt77 Wirth, N., Toward a Discipline of Real-Time Programming, Comm. of ACM, vol. 20, no 8, Aug. 1977.

Wirt83 Wirth, N., Programming in Modula-2, Springer-Verlag, 1983.

Wirt85 Wirth, N., From Programming Language Design to Computer Construction, Comm. ACM, vol. 28, no 2, February 1985.

Wulf72 Wulf, W.A., A Case against the GOTO, ACM Sigplan Notices, vol. 7, no 11, November 1972.

Zavo86 Zavodnik, R. J., and Middleton, M. D., YALE: The Design of Yet Another Language-based Editor, SIGPLAN Notices, vol. 21, no 6, June 1986.