Obelix Development Environment[1]

Anders Davidsson, Joakim Lindgren
{adn99006, jln99023}@student.mdh.se

Department of Computer Engineering at the University of Malardalen

July 18, 2000

[1]Part of the Asterix project at the department of computer engineering, University of Malardalen 2000. *Write something cool...*

Abstract

Bakgrund, problem, bidrag, metod. The devleopment environment for a real-time operating system is an important Many of todays Real-Time Operating Systems do not supply the necessay infrastructure needed to apply modern scheduling and analysis theory.

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 Requirements . . . 5
- 1.3 Summary . . . 6

2 Name definitions and terminology **7**
- 2.1 Asterix framework . . . 7
- 2.2 Obelix Development Environment (ODE) . . . 8

3 Obelix Configuration Tool design **9**
- 3.1 The configuration file . . . 10
  - 3.1.1 Application modes . . . 10
  - 3.1.2 Tasks . . . 10
  - 3.1.3 Semaphores . . . 11
  - 3.1.4 Signals . . . 11
  - 3.1.5 Wait- and Lock-free communication . . . 11
  - 3.1.6 Schedules . . . 11
- 3.2 Front-End . . . 11
- 3.3 Intermediate representation . . . 11
- 3.4 Semantic analysis . . . 12
- 3.5 Back-End . . . 12

4 Obelix Down-loader Tool Design **13**

5 Obelix Up-loader Tool Design **14**

6 Obelix Cross Compiler Design **15**

7 RCX Library Design **16**
- 7.1 Serial port driver . . . 16
- 7.2 Sensor driver . . . 16
- 7.3 Motor controller . . . 16
- 7.4 Console buttons . . . 16
- 7.5 Display window . . . 17

8 Implementation **18**
- 8.1 Obelix Configuration Tool . . . 18
- 8.2 Obelix Down-loader Tool . . . 18
- 8.3 Obelix Up-loader Tool . . . 18
- 8.4 Obelix Cross Compiler . . . 18
- 8.5 RCX Library . . . 18

9 Summary and conclusions **20**

A Obelix configuration language specification **22**

List of Figures

1.1 The Asterix architecture 5

3.1 The phases of a compiler 9
3.2 The phases of the Obelix Configuration Tool 10
3.3 The syntax tree 12
4.1 Obelix - Target System Communication 13

5.1 The OSI Reference Model. 14
5.2 Transport layer 14

6.1 The Obelix Cross-Compiler 15
7.1 The role the RCX library an Asterix system. 16

A.1 Obelix configuration language 23

Chapter 1

Introduction

1.1 Background

Embedded systems are a fast growing and exciting market. An example is a computer system that controls the speed of an electrical motor. A system like this is often called a Real-Time System (RTS). In [13] RTS is defined by:

> *A **real-time system** is a system that reacts on external events and perform actions within a fixed time. Correctness is not only depending on correct result but also on the point in time when the result was delivered.*
>
> *A **hard real-time system** is a system where the cost of not fulfilling the functional and temporal requirements is very high.*
>
> *A **soft real-time system** tolerates occasional errors with respect to the functional and temporal requirements. This means that constraints may be broken occasionally (typically with an upper bound defined over a time interval), and that a service may be accomplished a bit late occasionally (again within an upper bound).*

In order handle such systems an operating system that supports mentioned criteria is needed, in other words a real-time operating systems (RTOS).

When selecting an embedded software solution, for a new embedded system, developers face a number of technology choices:

- *Microprocessor* must be chosen on basis of cost, performance, power and application requirements.
- *Real-Time Operating System*, commercial or in-house development to fulfill the requirements of the embedded system.
- *Software Development Tools* which may be bundled with the commercial RTOS or selected for a particular microprocessor.

There exists a large number of commercial RTOS on the market. Most of them are based on old assumptions of RTS and do not supply the necessary infrastructure needed to apply modern/contemporary scheduling and analysis theory. This causes problems when new ideas need to be tested and evaluated, since the real-time aspect differs between state-of-the-practice and state-of-the-art. There already exists state-of-the-art RTOS, for example MiThos [11], the Spring kernel [18] and Emeralds [21], all with different advantages and drawbacks. These systems are often not general solutions but aim at something special, for example special hardware or scheduling theory. Most of the state-of-the-practice (commercial, off-the-shelf) RTOS are in some way configurable in order to fit in a general embedded system solution. The features in these RTOS vary a lot, and also the way they are distributed to the purchaser. Some of them are delivered as modules and others as open source code that can be modified and compiled to a specific system. If the developer needs to adjust the RTOS, the source-code is necessary, which may be a problem if the source-code is not open. Different kinds of synchronization, communication and collaboration between tasks are supported and so on. There exists more or less practical development environments that could be used to facilitate the configuration of a specific embedded system [5], [20], [19]. In fact many of these commercial RTOS (if not all) lack desirable state-of-the-art features of a Real-Time Operating System. Many questions are hard to find answers to, for example:

- Is the system predictable?
- Does the system support debugging of any kind?
- Which scheduling principles are supported?
- How does the error handler work, if there is one?
- Is priority inversion prevented?
- Is the system multitasking, is it preemptive?
- Which development tools exists?

This gave birth to the idea of a new RTOS at the department of computer engineering at Mälardalens Högskola. The name of the project is the Asterix framework. The idea is to develop a new analyzable distributed Real-Time Operating System, a communication system, a powerful development environment and analysis tools. In other words a complete set of development tools to configure and analyze a real-time system. The Asterix framework should have the following features:

- A task-model which supports state-of-the-art scheduling theory.
- Support for debugging and monitoring.
- Wait-free/lock-free interprocess communication.
- Minimized kernel jitter.
- Compiling, in other words, only the parts of the system that are utilized in a certain system is included.

The task-model includes both preemptive scheduling of statically generated schedules [] and fixed priority scheduling (FPS) []. It also supports both strictly periodic and event triggered tasks. The tasks terminate each time they finish execution.

A multi-tasking RTOS, like Asterix RTOS, enables the user to divide an application into separate, individual programs called *tasks*. A task is the basic unit of execution in any application that runs under Asterix. Each task can be started, suspended, resumed and terminated separately. Asterix handles hard and soft tasks, they can both be periodic or aperiodic. A hard task has hard time-constraints which must be kept or the whole system may crash. The hard tasks in the system are included in the system analysis, which makes sure that all time-constraints are possible to fulfill. A soft task do not have hard time-constraints and a missed deadline, for example, will not endanger the system. In other words, soft tasks will be assigned execution resources when there is time left from the hard task execution.

When Asterix suspends an executing task, it stores the current hardware register values, the task state, in a memory area dedicated to the task. Later when Asterix resumes the task it retrieves the register images from memory, loads them into the hardware registers and continues execution as though there had been no break. This process which is known as a *context switching*, is invisible to the task. Thus, a multitasking OS makes the system more efficient in time by permitting several activities to proceed concurrently.

In order to interest embedded real-time system developers, the Asterix framework must be portable and the source-code open. The architecture of the Asterix framework is illustrated in figure 1.1.

This first version of the Asterix framework aims at the Lego RCX micro-controller.

This document describes the Obelix development environment.

1.2 Requirements

The specification of the thesis was to build the development environment (DE) for the Asterix framework, named Obelix. Requirements:

- Down-loading from DE to the target system.
- Up-loading from the target system to the DE.

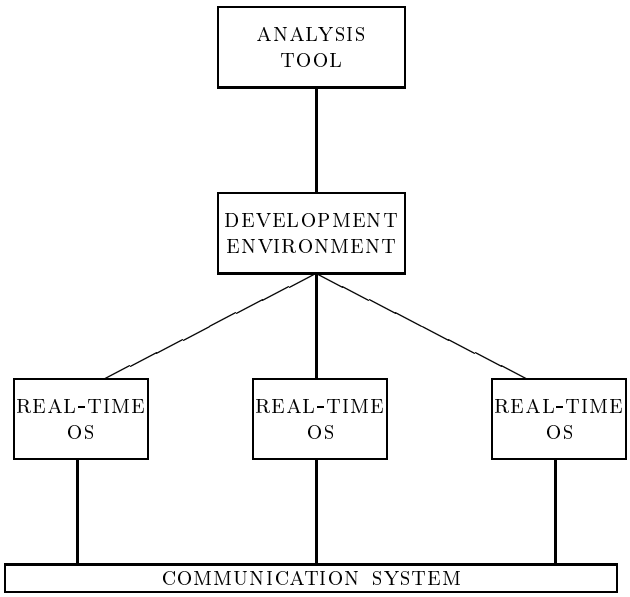

Figure 1.1: The Asterix architecture

- Front-end to a configuration tool
  a) Syntax, grammar and parser.
  b) Descriptions of tasks and relations between tasks, with regard to task-model, synchronization, communication, error-handling, etc.
- Back-end to initiate the kernel.
  a) Generate C source code.
  b) Compile along with the kernel.
- Supply for measuring of OS-overhead, and execution times for tasks. Data should be stored in suitable format in a system description file.
- To prove the correctness, of both the DE and the kernel, a sharp test of the system, from configuration file to working real-time system, should finish of the project.

The development environmnet should consist of some different tools:

- A down-loader that launches a complete Asterix system on the target hardware.
- An up-loader that allows the kernel to upload data from the target system back to the DE.
- A configuration tool that should be utilized to set up a Asterix system from the user point of view. The configuration tool should take a configuration file as input an then produce C-code that initiates the kernel.

- A cross compiler for the target system was also required. The compiler should be used to compile the Obelix generated c-code, the kernel code and the user written code into a working Asterix system.

1.3 Summary

Obelix is a development environment for the Asterix real-time kernel. Obelix consists of a configuration tool, a down-loader, an up-loader and a cross compiler for the target system. There are also some specific hardware routines developed for the Lego RCX microcontroller. The configuration tool generates C-code that, when compiled along with the kernel code, initiates the kernel on the basis of a application programmer specified configuration file. The cross compiler is used to compile and link the Obelix generated C-code, along with the kernel- and the application programmer written code into a working real-time system. The system is then downloaded on the target hardware and launched by the down-loader. The kernel may call the up-loader to send data back to the development environment.

In chapter two some definitions are and the remaining chapters of this document describes the design and implementation of Asterix framework development environment, Obelix, which is the authors master thesis.

Chapter 2

Name definitions and terminology

2.1 Asterix framework

The complete distributed real-time system, including analysis tools, development environment, real-time kernel and communication system.

Asterix

Asterix is the small, smart, powerful and free Real-Time kernel.

Obelix

Obelix is the potent and easy-to-use Development Environment in the Asterix framework.

Miraculix

Miraculix is the analysis tool for Asterix and Drakar.

Drakar

Drakar is the communication system, connecting a distributed Asterix system.

System / Asterix System

A system is a complete Asterix system, in other words both the real-time executive and the user defined application. An Asterix system are supposed to be downloaded on a target hardware system.

Application

An application is a program written by some developer. The application may consist of arbitrary number of tasks.

Mode

An Asterix system (see above definition) can run in two different modes, test-mode and normal-mode. An application may be programmed to have different application modes (schedules), which is supported by Asterix.

Task

A task is the basic unit of execution in any application that runs under Asterix. The Asterix framework supports two classes of tasks, hard and soft.

Wait- and lock-free communication

Wait- and lock-free communication is a form of state based inter-process communication (IPC). Writers and readers communicate over a channel that is made up by a number of buffers. Wait-free communication guarantees instant access of the channel and a task cannot be locked. In this document we name wait- and lock-free communication as wait-free communication.

Target system

The target system is the platform/hardware on which Asterix are going to be executed. In our case it is a Lego Mindstorm RCX micro controller with a Hitachi H8 microprocessor. Asterix is supposed to be portable to any kind of hardware system.

Configuration file

It is the input file to the Obelix Configuration Tool (OCT). This file describes the user application on a higher level than ordinary source code. It should be written in ASCII format, therefore it is (human-) readable. The producer of this file could be a application programmer or a application design tool.

System Description File

This file will be generated by the measuring mechanisms in the kernel. It contains execution time for the tasks, OS overhead for a number of cases. To upload the timing information we will use the Obelix Up-loader Tool (OUT).

Lego RCX micro-controller

A suitable platform under the development phase is the LEGO Mindstorm computer unit. A good example of a embedded system, based on a Hitachi 8-bit microprocessor that runs at 16 Mhz and equiped with 32k ram.

2.2 Obelix Development Environment (ODE)

Development platform

The development platform is PC running Windows NT. It is the system where Obelix run, in other words the platform where Asterix real-time embedded systems are developed.

Obelix Configuration Tool (OCT)

The configuration tool generates initiation files for the kernel on the basis of a user defined configuration file. The initiation files describes the system properties and which modules that must be compiled into the system.

Obelix Down-loader Tool (ODT)

This is the down-loader tool in the Obelix environment. It takes a binary file as argument and tries to contact the RCX unit. If a connection is establish, the file is transferred and placed in the memory of the RCX. Finally, the RCX receiver program launch the down-loaded system.

Obelix Up-loader Tool (OUT)

This tool sends data back from the Asterix OS to the DE during run-time. The data can be timing- or debugging information. In fact, this tool delivers a communication service between the RCX and the development platform. It consists of code on both the development and the target platform.

Obelix Cross Compiler (OCC)

The cross compiler is the tool used for compiling an Asterix system into a proper binary file. The compiler run in the development environment, on the development platform, and generate machine code for the target hardware. In our case, the compiler is a GNU gcc cross compiler ported to generate code for the Hitachi H8 micro-processor.

RCX Library

Hardware device driver library for the Lego Mindstorm RCX platform.

Chapter 3

Obelix Configuration Tool design

The Obelix Configuration Tool (OCT) takes a configuration file as input and generates kernel initiation files as output. The output files are then included when the whole Asterix system is compiled.

Asterix is a compiling Real-Time Operating System. The idea is to let the application programmer write a configuration file to specify his Asterix Real-Time System, and then feed the Obelix Configuration Tool with it. The configuration tool then generates C-code to initiate the kernel. After that the whole system is compiled in the development environment by a cross-compiler and only those parts of the system that are utilized will be included in the target system. The target system is then downloaded to the target and launched on the target system. The kernel code is static for all systems (on the same hardware target), the Obelix configuration tool generated C-code is the dynamic part of the system and it specifies a Asterix real-time system together with the application programmer written code.

The Obelix Configuration Tool consists of a front-end, an internal representation of the configuration, semantic analysis of the specified system and a back-end that generates C-code. An illustration of a typical compiler is given in figure 3.1 and more described in the book *Compilers: Principles, Techniques, and Tools*[1]. As the picture illustrates a compiler has several phases. OCT must perform lexical-, syntactic- and semantic analysis on the configuration input file. In our model of the configuration tool, the front-end translates the source code (the configuration file) into an intermediate representation, which is semantically analyzed and from which the back-end generates the target code as illustrated in figure 3.2. The details of the target language are part of the back-end. It would be possible to translate the input directly to the target language but we decided to use a intermediate representation due to:

- Re-targeting is easier, it is possible to attach a new back-end.
- Good availity through an internal representation facilitate the optimization of the memory usage. It is possible to apply an independent code optimizer.
- We found it suitable to apply semantic analysis when the internal representation exists.

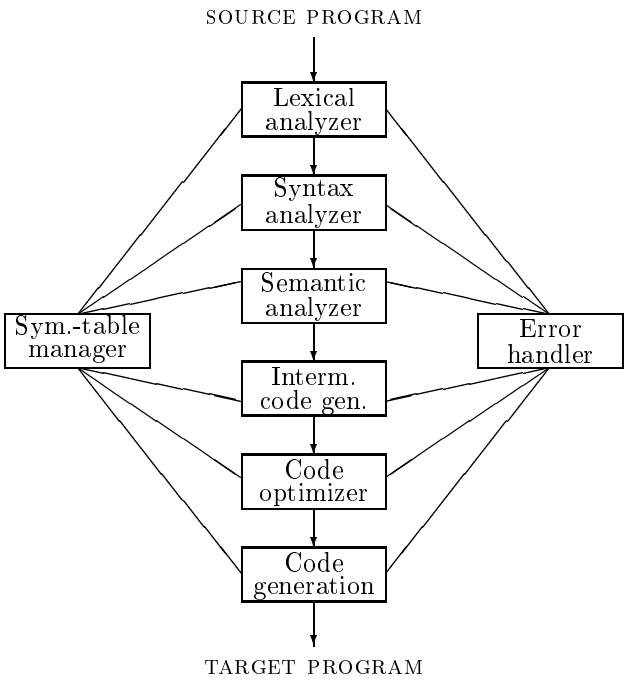

Figure 3.1: The phases of a compiler

The objective of Obelix Configuration Tool is to set up and initiate data structures for the kernel. The tool is closely related to the kernel and the requirements of the tool are specified on the basis of the kernel. A model of the kernel was used when designing the tool, since the kernel was developed in parallel.

The kernel supports preemptive execution and fixed priority scheduling with or without offset. The requirements of the kernel consists of support for tasks, task synchronization by semaphores and signals, wait-free interprocess communication, and support for specification of application modes along with clock resolution.

There was also requirements of two Asterix system modes, one for running a sharp system and one for measuring the target hardware and also measure the WCET (Worst Case Execution Time) for tasks. The timing information will then be sent back to the development platform by the Obelix Up-loader Tool.

The kernel supports periodic tasks that are triggered periodic by the clock, and aperiodic tasks that are triggered by events. The tasks terminates each time they

finish execution.

Early, different main parts of a system was identified. We found it suitable to arrange the configuration according to those parts. The main parts of a system were identified as: application modes, tasks, synchronization, and communication. The synchronization is facilitated by semaphores and signals and the communication, in this first time version, is wait- and lock-free inter-process communication. See chapter 2 for further information about wait-free communication.

To specify an Asterix system, the system mode, sharp or measuring, has to be defined along with the different application modes. We also decided to, from a optimization point of view, to input the available RAM size of the target hardware .

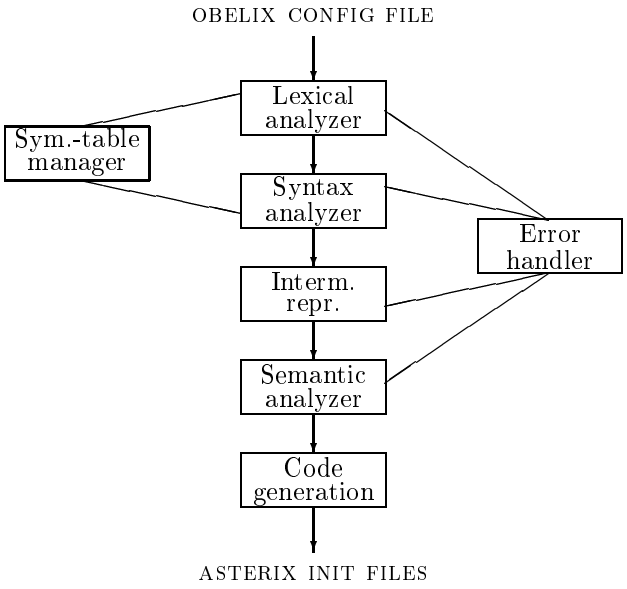

Figure 3.2: The phases of the Obelix Configuration Tool

3.1 The configuration file

The syntax of the configuration file was defined by a context free grammar[1].

The grammar specification is given in appendix A and one reason for using such a formal description of the language is the support for such grammars in the Bison tool, which is used in the configuration tool front-end. When the syntax of the grammar was chosen we had in mind that both humans and machines should be able to read and, in some way, understand the contents.

[1]Formally named Backus Naur Form. A widely used notation for specifying the syntax of a language.

3.1.1 Application modes

A desirable feature of a RTOS is support for different application modes. The application can have several natural modes, for example an aircraft control system may work in one mode while the plane is on the ground and another mode after take off. If an error, for example a dropped wing, occur a third mode is necessary. Different application modes may consist of different task schedules and some other parts that defines the mode. In the Obelix Configuration Tool we decided to support different application modes. A system is defined by the different application modes. One mode is defined by the tasks that operate in the mode, the resources that exists, the clock resolution and the mode identity. The resources are synchronization, by semaphores and signals, and inter-process communication by wait-free communication. An Asterix system may have an arbitrary number of different modes. Mode changes are supposed to be supported by the kernel. A mode in an Asterix system is totally isolated from other modes in the system. This means that the same identity can be used in different modes It also imply that all tasks and resources has to be declared in each mode they will participate in.

3.1.2 Tasks

An application mode, in the configuration of an Asterix system, has to consist of clock resolution, tasks, synchronization and communication resources. The Obelix configuration tool supports two task classes, hard and soft tasks. The difference between the classes, from an analytical point of view, is that hard tasks have hard real-time requirements but soft tasks have not. To define a task (both hard and soft) some attributes have to be stated:

- A unique identity.
- The way the task is supposed to be triggered need to be clear, it may be time triggered or triggered by a signal in the system.
- To be able to define precedence relationship among tasks an offset is associated with each task. The offset is a displacement from the period start of the task.
- The kernel supports deadline supervision and therefore the deadline, relative period start, has to be specified for each task.
- The system demands unique user specified priorities for tasks.
- To minimize the memory of the embedded system the stack size of each task should be application programmer specified.
- For each task, a start function has to be specified along with optional arguments to that function.

- The kernel also needs to know which error routine should be invoked if something goes wrong during run-time.

Those attributes specifies the Asterix task model from the configuration tool point of view.

3.1.3 Semaphores

An important issue for semaphores, in an RTOS, is the handling of priority inversion, where a high priority task is prevented from running because it needs a semaphore owned by a lower priority task. In the Asterix system this is handled with the immediate inheritance protocol [17]. The protocol prevents priority inversion by assigning each semaphore a ceiling, pre-runtime, on the basis of the highest priority of the tasks that accesses the semaphore. When a task, under run-time, access the semaphore it immediately inherits the priority corresponding to the semaphore ceiling. When the task releases the semaphore, the priority is restored. A semaphore, thus, needs the attribute ceiling along with unique identity and a list of tasks that is authorized to access the semaphore. The list is necessary in order to guarantee the real-time requirements of the hard tasks in the system, we cannot allow both hard and soft tasks to access the same semaphore. A soft task does not have any guarantees for finishing execution within a certain time, and this may imply blocking of a higher priority hard task.

3.1.4 Signals

A signal need to have a unique identity and a list of tasks that intend to use[2] it under runtine.

The kernel uses the list to make sure that only those tasks that are authorized accesses the signal.

3.1.5 Wait- and Lock-free communication

The inter-process communication supported by the kernel is called wait- and lock-free IPC. The reason for this is experimental and we have not found any other RTOS that supports it. Wait- and lock-free communication is a protocol that prevents a user of a communication channel to be blocked. The chosen wait-free algorithm is a simple one and it supports a single writer and multiple readers of each wait-free channel. A channel is a number of buffers which the kernel will distribute over the users, the number of buffers must be high enough to ensure that there always is one available when a task access the channel. The writer and the readers have to be known along with the number of buffers and which type of messages the channel should handle. If the number of buffers is large enough, it is possible to give history over written messages.

[2]Either rise or wait for (or both) a signal

3.1.6 Schedules

Asterix will support application mode changes and the design of the Obelix configuration tool was done on the basis of that. An application mode change is a change in the task execution order. The order is based on task-priority, period and offset. There may also be additional tasks in a new mode. A mode is, in other words, a schedule. An application mode in Obelix configuration tool is defined by the tasks and all resources in the mode. A new mode is totally separated from other modes. If tasks and resources are active in multiple modes they have to be defined in all these modes. An alternative to this is to introduce a new building block of the Asterix system, called *schedule*. Instead of the mode building block the schedule block can be used. The schedule building block must contain information about the task execution order, in other words the definition of which task that will execute at a given time. The necessary attributes for a schedule is then:

- The identites of the tasks that participates in the schedule.
- Task priorites
- How the tasks are activated, that is period or the event the task should react on.
- Task offsets.

It is possible to remove the schedule information of the tasks from the task declaration in the configuration. The kernel then has to support schedule changes instead of mode changes, which is, in fact, the same thing.

3.2 Front-End

The front-end requires a scanner and a parser to interpret the configuration file. We decided to make use of the reliable GNU tools Flex and Bison [15]. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1)[3] context-free grammar into a C program to parse the grammar. Flex is a tool for generating programs that performs pattern matching on text (scanning). A combination of these is our configuration tool front-end. The alternative was to write our own scanner and parser, but that seemed to be unnecessary. Further information on the front-end is given in [3].

3.3 Intermediate representation

Bison gives opportunities to apply semantic actions while parsing the specified grammar. We will use this to build up our internal representation of the configuration specified in the input file. The internal representation is an abstract syntax tree, consisting of

[3]LookAhead Left-to-right Rightmost-derivation

nodes corresponding to the building blocks of the configuration as shown in figure 3.3. In the figure we see a root of the tree pointing at an Asterix System. An Asterix System consists of, at least, one application mode. If there is additional application modes, the next pointer is attached to it. Each application mode has tasks, synchronization and communication parts. The Tasks part consists of hard and soft tasks and each of those consists of a list of user specified tasks. The Synchronization part consists of semaphores and signals, which also consists of lists of user specified entries. Finally the communication is, in this version of Asterix, application programmer specified wait-free communication between tasks. A more detailed description of the internal representation is given in [7].

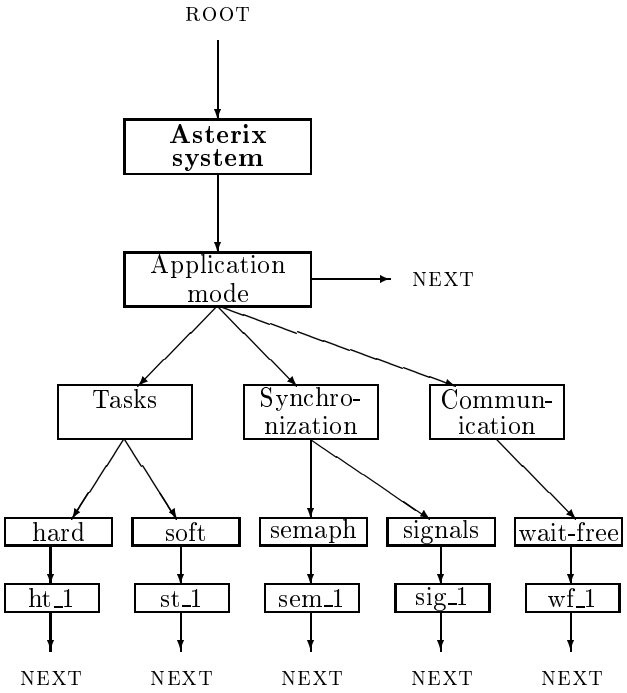

Figure 3.3: The syntax tree

3.4 Semantic analysis

When the configuration file is scanned, parsed and the internal representation exists, we need to apply some semantic controls and computation. This is done in several phases on the syntax tree.

The identites of the tasks, semaphores, signals, and wait-free channels used in the configuration file must be declared. If a task is triggered by a signal, the signal must be declared and the task must be in the user list of the signal. All tasks must be assigned unique priorities and all hard tasks have to have higher priorities than all soft tasks. We also check if the user-assigned stack size seams to be reasonable. A warning is given if the stack size is less then 10 bytes.

Each semaphore ceiling needs to be calculated and we also must check if the tasks that accesses the semaphore belong to conjunct classes. For each signal we need to state that all users are declared as tasks in the system, in other words a semaphore, can not be a user of a signal. The last semantic action deal with the wait- and lock-free channels, the users must also be tasks and there have to be enough buffers to ensure wait- and lock-free communication. Further information is given in [3].

3.5 Back-End

After controls and calculations we know that the configuration is formally correct and the internal representation corresponds to it. The remaining work is to generate source code that initiates the kernel. Typically that is accomplished by a back-end. The Obelix configuration tool back-end produces two files. One C-file and one header file, the C-file consists of declarations and initiations of appropriate data structures, and the header file of external declarations and definitions. The back-end traverses the syntax tree, application mode by application mode, and writes corresponding C-code to the initiation files. The procedure is very straight forward and the intermediate representation contains all necessary information. See [2] for a more technical description of the back-end.

Chapter 4

Obelix Down-loader Tool Design

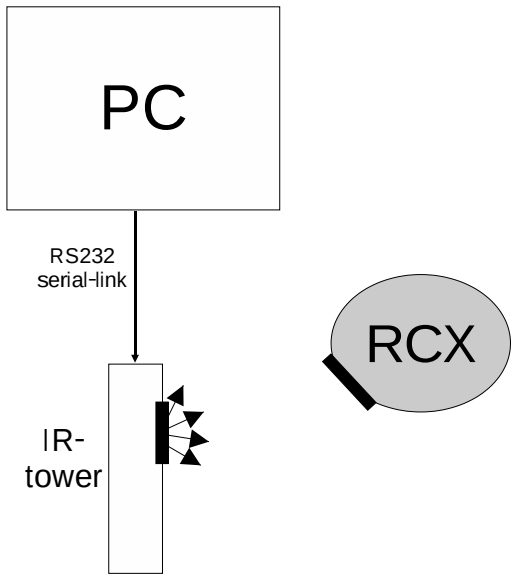

Figure 4.1: Obelix - Target System Communication

The Obelix Down-loader Tool takes an Asterix system file and download it to the target platform and launch the actual system. Communication with the target platform was in our case a serial bus over a IR-link, see figure 4.1.

The Asterix OS is constructed for embedded computer systems. The down-loader tool has therefore a central role in the development environment. Without it, the system cannot be started. Further, each time a change is done in the application or in the kernel the whole system has to be recompiled and down-loaded. The down-loader tool is also responsible for the startup of the kernel and the tool is very platform dependent.

The down-loader tool must take an Asterix system binary file and download it to the memory of the target platform. A program at the target platform has to receive the file and call the "start"- function of the down-loaded file.

A down-loader tool thus consist of two programs, one at the development platform that sends the binary file, and one that is already placed at the target platform. The receiver program must then be placed in memory in the target platform, when turning on the power to the computer. This could be accomplished in a number of ways, for instance with a ROM memory that is preprogrammed or a start-technique over a network. In the latter case, there has to be support in the hardware on the network card.

The tool will take a binary file as input, and the srec format were chosed as file format. The srec format is devided into frames with startaddress where to put the code along with a check sum for each frame.

When looking at the hardware of our target platform, we see that it has a serial bus over an IR-link. The original Lego Mindstorm software uses this to transfer the program to the RCX, and we will do the same. One aspect that can be taken into account at this stage of the design is the bandwidth of the serial bus. One difference from the Lego original system is that Asterix RTOS is compiled with the application into one binary file and Lego uses a dynamic task initiation at the RCX. Therefore we have to download more data each time a change is done in the application, but we get the advantage that we only have those modules that are in use in the application. So let us look at the transfer rate versus the size of the binary file(the load file). Assume that we have a binary file with maximum size, that is approximate 32 KB and the bandwidth of the serial bus is 2400 BPS. This gives a transfer rate of approximate 110 seconds, which is acceptable. Some overhead will be added and depending on the transfer format the transfer rate may change, but we assume it still will be reasonable. An improvement could be to increase the bit rate to, lets say, 4800 BPS, if that still works fine with the IR-link.

Chapter 5

Obelix Up-loader Tool Design

With the Obelix Up-loader Tool, information from the target platform can be up-loaded to the development environment. It can be debugging- or timing information.

One of the features that the Obelix development environment was specifed to support was a possibility to give a respons from the kernel to the development platform. That could be debugging information or data from the time measuring mechanism in the kernel. We decided to design and implement some kind of a protocol stack. One important aspect was to make the tool as small and as simple as possible. We found that it would be enough to define a package delivery service from the target platform (RCX) to the development platform (PC). Our work on this has of course some relation to the communication protocol called Drakar, and we found that we had come quite near that part of the Asterix framework.

The protocol can be split up in a few layers, inspired from the OSI reference model [4], illustrated in figure 5.1. The physical layer is a serial bus over an IR-link. Read and write serial primitives had to be written and we choose to make polling versions, see chapter 7. It would be harder to predict the behavior of an interrupted serial primitive. Anyway, the up-loader tool was not created to be used by the tasks, but rather by the kernel. The protocol was not designed as reentrant, and it must be overlooked again if someone has intention of using it for the tasks.

The next layer in the protocol-stack is the data-link layer, it deliver data in frames with a maximum data size of 16 bytes. It uses the send-and-stop principle for safe frame transmissions.

To be able to send larger pieces of data than 16 bytes, we let a transport layer split up a data stream up to 64 KB into 16 bytes frames, and send them individual, see figure 5.2. On the receiver side we reassemble and deliver them to the application. To be able to decide how big a package is, a frame with the size of the package is sent in the beginning of each package.

The last layer from the OSI-model that we have defined is the application layer, that is currently a memory dump application and a time measuring function.

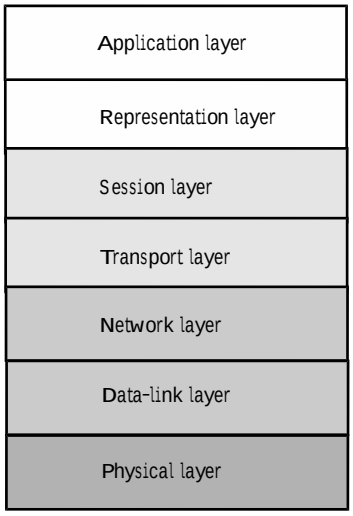

Figure 5.1: The OSI Reference Model.

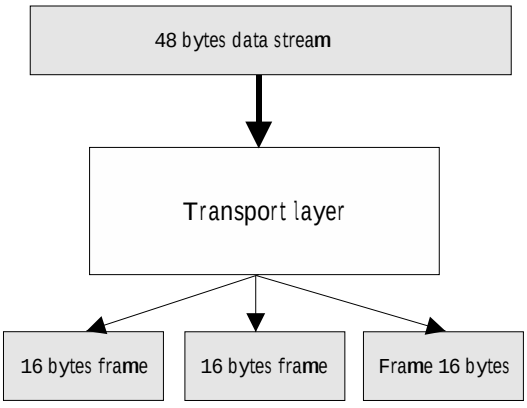

Figure 5.2: Transport layer

Chapter 6

Obelix Cross Compiler Design

The cross compiler in the Obelix environment compile all source code that should be running at the target platform. All compiling is done off-line, before the system is down-loaded.

An important part of the Obelix development environment is the cross compiler. The compiler will run under the development environment (Windows NT/Linux) and compile the source language, C-code, into executable machine-code for the target platform. This means that the cross compiler is target dependent and the compiler back-end has to be replaced , when developing Asterix systems for a new target hardware architecture.

In this, first, version of the Asterix system the target system is a Lego RCX micro-controller, with a Hitachi H8 3292 processor [6].

Since the compiler generates code for a different target system than the development platform, it requires information about the memory space in the target platform. This information is fetched from a linker file that was written by us.

The compiler takes four different types of source files. First, the kernel source, that is a number of both C- and assembler-files, next there is the output files from the configuration tool (OCT). Third, the application programmer files that includes global memory, task- and error-routines to the application, and finally, optional hardware dependent library files (OHL), see figure 6.1.

From these files the compiler generates a load-file/binary-file in srec format. This format is used because the down-loader tool was designed for those file types.

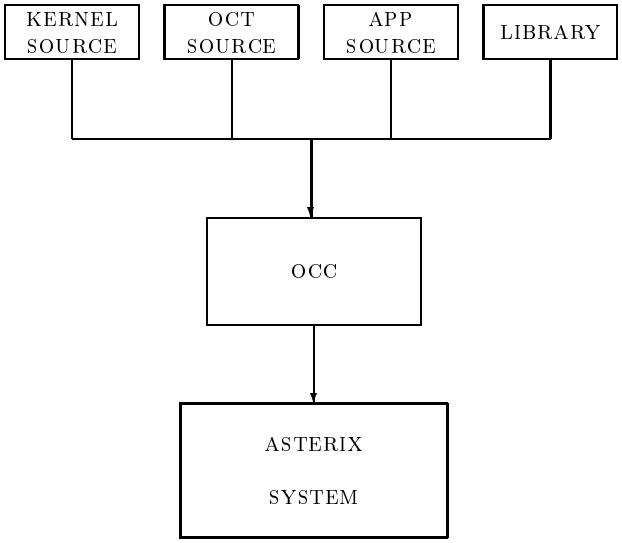

Figure 6.1: The Obelix Cross-Compiler

Chapter 7

RCX Library Design

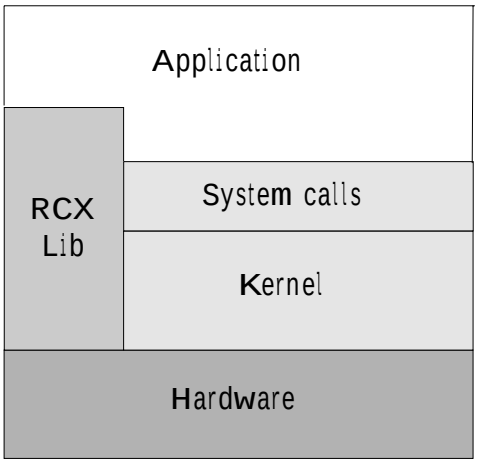

Figure 7.1: The role the RCX library an Asterix system.

This part of the Obelix Development Environment was not specified from the beginning, but rather a part that became natural when we began to design and implement other tools. We cannot really say that the library is a part of the Obelix development environment, but rather a separate support library to access the hardware of the RCX. The following section describes each device driver. In figure 7.1 the role of the RCX library in an Asterix system is illustrated.

The result is a hardware dependent device driver library that supports most of the devices on the RCX. The devices can be used to make a application that communicate with the environment, e.g. make use of sensors ar motors. When an application programmer wants to use one or more of the device drivers, he or she will be able to include the specific header file for the device and compile the required source code associated to the device. One device on the RCX that is not support is the internal speaker.

When specifying the library, one requirement was to use as little memory as possible, since there is limited memory on the RCX. Further, the device drivers should be interruptible, but not reentrant at this level. If a device driver should be used in multiple tasks it is the application programmer responsibility to consider the synchronization problem.

7.1 Serial port driver

When looking at the Obelix Up-loader Tool, it is obvious that read and write serial primitives are necessary to access the serial port. They were made in a polling version, because it is easier to predict the behavior of a polled than an interrupted version. The driver supports one serial port since the RCX is only equipped with one [6].

7.2 Sensor driver

Routines to access the sensor input port will certainly be useful to the application programmer. There are a number of different sensors present in the LEGO assortment. A few them require quite extensive software to be useful and we wanted a more general driver. A simple sampler routine would be enough. With the sampler routine it is possible to write more sophisticated reader routines for specific sensor types.

7.3 Motor controller

There are also three motor outputs on the RCX. A routine to control them was specified. The routine will take two arguments, one to identify which motor to control, and one to tell what state to put it in.

They can be in the following states:

- running forward
- running backward
- stoped
- floating

The difference between stop and float is that stop holds the motor fixed and float only turns the power of.

7.4 Console buttons

On the RCX unit there are four digital buttons. A routine that read the digital value of them was planed.

7.5 Display window

There is also a display on the RCX unit, that may be very useful to access. A routine that could control all LCD segment on the display and also a routine that print decimal values on it was desirable. There is only space for four digits, so it would be convenient to make a routine that prints a 16 bits integer in hexadecimal format.

Chapter 8

Implementation

8.1 Obelix Configuration Tool

The implementation of the Configuration Tool is straight forward. In the front-end [3], the GNU tools Bison and Flex were utilized to build the scanner and parser for the configuration file. A package for symbol analysis [10], was used to make sure that all identitie's in the configuration file was declared as identities somewhere, and only used once in an application mode. A set of data-structures was created to represent the configuration file internally and some routines for building a syntax tree was also created, see [7]. A few routines for semantic analysis were constructed. The back-end traverses the syntax tree in order to put C-code into the kernel initiation files. All code were written i C and a more detailed description of the back-end is given in [2].

8.2 Obelix Down-loader Tool

The RCX hardware was already programmed with a down-loader receiver routine that can be used. This facilitated the implementation, since the platform already had been used for similar operations. The receiver routine on the RCX ROM support a serial connection at 2400 BPS. To increase the transfer rate up to 4800 BPS changes has to be done on the RCX. This could be accomplished by changing in the serial status register in the UART at the RCX, and than letting the ROM receiver routine run as normal.

When we were looking for background information we soon discovered that there were other groups that had done some work that we could reuse. This was the case with the down-loader tool. Kekoa Proudfoot had implemented a down-loader called firmdl3 that supported a bit rate up to 4800 BPS, that would satisfy our specifications [16].

Kekoa Proudfoot's down-loader tool takes binary files in srec format, that is a hexadecimal format that gives addresses and the data but also a checksum for chucks of 42 bytes, and download them to the RCX.

We had problems when using the read and write primitives under Windows NT. The down-loader tool was original written for Linux. We found that it would be advantageous to reuse it so we ported the tool to Windows32 API. This means that the routines *createFile*, *readFile* and *writeFile* were re-written. This resulted in a stable working, compile-able version for both Windows NT and Linux.

8.3 Obelix Up-loader Tool

The implementation where initiated by creating the hardware primitives "read" and "write" for the RCX. They are deeper described in RCX library manual [9]. The protocol was implemented layer by layer on the RCX and the PC. We had to consider the fact that the Hitachi GNU compiler used little endian and Borland C++ used big endian representation. Further information is given in [8].

At the PC side we used the Windows32 API functions *readFile* and *writeFile*, which worked out real fine. This lead to that this tool only is compile-able on a windows 32 bits system. We used Borland C++ to compile the tool.

8.4 Obelix Cross Compiler

We used a GNU gcc Hitachi cross compiler [14] that already was configured for our target platform. We added a linker script that tells the compiler which memory areas that are available as RAM. For instance, there were a ROM memory in the first Hex4000 bytes of the memory space. The RAM began at Hex8000 and there were also some memory mapped hardware devices in the RAM space.

8.5 RCX Library

Before any code was written the ROM memory in the RCX was examined to see if the original LEGO operation system had any support for the hardware devices. As expected, a large part of the Lego's OS was placed in the ROM memory, and also the device drivers. Unfortunately, the device drivers were placed in interrupt routines that would run with a certain frequency in the background of the user application. Thus we had to implement the device drivers of our own, except for those drivers that access the display and the buttons on the RCX. These ROM routines accessed the devices directly.

Some part of the code is influenced by the LegOS project [12], especially when we worked with the routine that initialize the serial port.

The C language was used to implement the library but that was not actual appropriate, due the diligent use of bit operation when accessing the status registers in the UART:s and A/D converter. The source code was compiled into a library, and when an application programmer wants to use a device on the RCX he, or she, include the specific header-file for the device. The accessible header-file for the application programmer is placed in the /system/lib/include directory in the project root catalogue.

Chapter 9

Summary and conclusions

In this paper the Obelix Development Environment (ODE) has been described. ODE is a part of the Asterix frame-work, which for the moment consists of the Asterix real-time kernel and the Obelix development environment. The role of Obelix in the compiling real-time system has been clarified. The main task is to facilitate software development for the chosen hardware system. The tools in the environment has been presented and some design trade-offs have been discussed. Obelix consists of a configuration tool, a down-loader, an up-loader and a cross compiler. This Obelix version aims at the Lego Mindstorm RCX micro-controller, and a hardware specific library is also included in Obelix. The final product is a set of tools that fulfill the requirements of the specification.

There are a few characteristics that is not yet analyzed and implemented in the environment, for example support for multiple modes in the Obelix configuration tool. This requires investigation of how the system should handle the mode changes semantic and how this should be implemented without disturbing the task model. Yet, we have presented two ways of attacking the problem. The interrupt handling has not either been defined, but we have presented the semantics and a way to express it in the configuration file.

Still there are some work to do on the time measuring management. The time measuring function in the kernel has only supports execution time measuring of one task at the time. This could be extended to measure the time of a whole task-set, the system calls and the OS overhead. Another interesting thing to work on is the communication protocol Drakar. This work includes analysis and specification of the protocol as well as implementation. Integration with the Obelix configuration tool and the kernel is also needed.

Finaly we would like thank our supervisor Henrik Thane and Krisitian Sandstrom, at the deparartment of computer science at the University of Malardalen, for their support.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers; principles, techniques and tools. *Addison-Wesley*, 1986.

[2] A. Davidsson. Obelix configuration tool back-end. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[3] A. Davidsson. Obelix configuration tool front-end. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[4] J.D. Day and H. Zimmermann. The osi referece model. *Proc. of the IEEE, vol. 71, pp. 39-55*, 1995.

[5] G. Ragazzini G. Castelli. Eos: a real-time operating systems adapts to application architecture. *IEEE Micro, vol. 15 5, pp. 41- 49*, 1995.

[6] Hitachi. Hitachi single-chip microcomputer h8/3297 series. *Hardware manual, 3rd edition.*, 2000.

[7] J. Lindgren. Obelix configuration tool intermediate representation. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[8] J. Lindgren. The obelix up-loader tool. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[9] J. Lindgren. Rcx library manual. *Master Thesis, Malardalens University, Department of Computer Engineering, Vasteras June 2000.*, 2000.

[10] J. Maki-Turja and C. Eriksson. Paket for enkel symbolhantering. *Idt/CUS University of Malardalen*, 1996.

[11] F. Mueller, V. Rustagi, and T. P. Baker. Mihtos, a real-time micro-kernel threads operating system. *IEEE*, 1995.

[12] M.L. Noga. legos. *http://www.informatik.hu-berlin.de/ mueller/rt/mindstorm/www.multimania.com/legos/*, 2000.

[13] C. Norstrom, C. Sandstrom, J. Maki-Turja, H. Hansson, and H. Thane. Robusta realtidssystem. *Malardalen Real-Time Research Center*, 1999.

[14] GNU's not UNIX. The gcc homepage. *http://www.gnu.org/software/gcc/gcc.html*, 2000.

[15] GNU's not UNIX. The gnu homepage. *http://www.gnu.org/software/*, 2000.

[16] Kekoa Proudfoot. Rcx tools. *http://graphics.stanford.EDU/ kekoa/rcx/tools.html*, 1998.

[17] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.*, 1990.

[18] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE*, 1991.

[19] H. Hansson H. Lawson O. Birdal C. Eriksson S. Larsson H. Lon M. Stromberg. Basement: An architecture and methodology for distributed automotive real-time systems. *IEEE Transactions on Computers, vol. 48 9*, 1997.

[20] H. Neugass G. Espin H. Nunoe R. Thomas D. Wilner. Vxworks: an interactive development environment and real-time kernel for gmicro. *TRON Symposium, Proc. Eighth, pp. 196-207*, 1995.

[21] K. M. Zuberi and K. G. Shin. Emeralds: A microkernel for embedded real-time systems. *IEEE*, 1996.

Appendix A

Obelix configuration language specification

This is the context free grammar for the Obelix configuration language. The grammar is a 4-tuple:
`(V, T, P, S)`

- `V` is syntactic variables, nonterminals, that denote sets of strings.

```
V = { file, systemmode, ram, modes, mode, resolution, tasks, hardtasks,
      softtasks, task, activator, args, error_routine, resources,
      communication, waitfrees, waitfree, num_buf, readers, reader,
      synchronization, signals, signal, users, user, semaphores, semaphore }.
```

- `T` is terminals, basic symbols from which strings are formed, The word "token" is a synonym for "terminal".

```
T = { SYSTEMMODE, SYSMODE, RAM, INT_CONST, MODE, ID, RESOLUTION,
      HARD_TASK, SOFT_TASK, ACTIVATOR, OFFSET, DEADLINE, PRIORITY,
      ROUTINE, ARGUMENTS, ERR_ROUTINE, STRING_CONST, WAITFREE, WRITER,
      TYPE, NUM_BUF, READER, SIGNAL, USER, SEMAPHORE, STACK }.
```

- `P` is productions which specifies the manner in which the terminals and nonterminals can be combined to form strings. `P = { See figure A.1 }`.

- `S` is the start symbol of the grammar.

```
S = { file }.
```

```
file            ->  systemmode ram modes
systemmode      ->  SYSTEMMODE = SYSMODE;
ram             ->  RAM = INT_CONST;
modes           ->  modes mode
                 |  mode
mode            ->  MODE ID { resolution tasks resources };
resolution      ->  RESOLUTION = INT_CONST;
tasks           ->  hardtasks softtasks
hardtasks       ->  hardtasks HARD_TASK task
                 |  epsilon
softtasks       ->  softtasks SOFT_TASK task
                 |  epsilon
task            ->  ID { ACTIVATOR = activator;
                         OFFSET = INT_CONST;
                         DEADLINE = INT_CONST;
                         PRIORITY = INT_CONST;
                         STACK = INT_CONST;
                         ROUTINE = ID;
                         args
                         error_routine };
activator       ->  INT_CONST
                 |  ID
args            ->  ARGUMENTS = STRING_CONST;
                 |  epsilon
error_routine   ->  ERR_ROUTINE = ID;
                 |  epsilon
resources       ->  communication synchronization
communication   ->  waitfrees
waitfrees       ->  waitfrees waitfree
                 |  epsilon
waitfree        ->  WAITFREE ID { WRITER = ID;
                                  readers
                                  num_buf
                                  TYPE = STRING_CONST; };
num_buf         ->  NUM_BUF = INT_CONST;
                 |  epsilon
readers         ->  readers reader
                 |  reader
reader          ->  READER = ID;
synchronization ->  signals semaphores
signals         ->  signals signal
                 |  epsilon
signal          ->  SIGNAL ID { users };
users           ->  users user
                 |  user
user            ->  USER = ID;
semaphores      ->  semaphores sempahore
                 |  epsilon
semaphore       ->  SEMAPHORE ID { users };
```

Figure A.1: Obelix configuration language