LF: A Language for Reliable Embedded Systems

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF STELLENBOSCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
F.A. van Riet
November, 2001

Supervised by: Dr. P.J.A. de Villiers

Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Abstract

Computer-aided verification techniques, such as model checking, are often considered essential to produce highly reliable software systems. Modern model checkers generally require models to be written in **CSP**-like notations. Unfortunately, such systems are usually implemented using conventional imperative programming languages. Translating the one paradigm into the other is a difficult and error prone process.

If one were to program in a process-oriented language from the outset, the chasm between implementation and model could be bridged more readily. This would lead to more accurate models and ultimately more reliable software.

This thesis covers the definition of a process-oriented language targeted specifically towards embedded systems and the implementation of a suitable compiler and run-time system.

The language, **LF**, is for the most part an extension of the language **Joyce**, which was defined by Brinch Hansen. Both **LF** and Joyce have features which I believe make them easier to use than other **CSP** based languages such as **occam**. An example of this is a selective communication primitive which allows for both input and output guards which is not supported in **occam**.

The efficiency of the implementation is important. The language was therefore designed to be expressive, but constructs which are expensive to implement were avoided. Security, however, was the overriding consideration in the design of the language and runtime system.

The compiler produces native code. Most other **CSP** derived languages are either interpreted or execute as tasks on host operating systems. Arguably this is because most implementations of **CSP** and derivations thereof are for academic purposes only. **LF** is intended to be an implementation language.

The performance of the implementation is evaluated in terms of practical metrics such as the time needed to complete communication operations and the average time needed to service an interrupt.

Opsomming

Rekenaar ondersteunde verifikasietegnieke soos programmodellering, is onontbeerlik in die ontwikkeling van hoogs betroubare programmatuur. In die algemeen, aanvaar programme wat modelle toets **CSP**-agtige notasie as toevoer. Die meeste programme word egter in meer konvensionele imperatiewe programmeertale ontwikkel. Die vertaling vanuit die een paradigma na die ander is 'n moelike proses, wat baie ruimte laat vir foute.

Indien daar uit die staanspoor in 'n proses gebaseerde taal geprogrammeer word, sou die verwydering tussen model en program makliker oorbrug kon word. Dit lei tot akkurater modelle en uiteindelik tot betroubaarder programmatuur.

Die tesis ondersoek die definisie van 'n proses gebaseerde taal, wat gemik is op ingebedde programmatuur. Verder word die implementasie van 'n toepaslike vertaler en looptyd omgewing ook bespreek.

Die taal, **LF**, is grotendeels gebaseer op **Joyce**, wat deur Brinch Hansen ontwikkel is. **Joyce** en op sy beurt **LF**, is verbeterings op ander **CSP** verwante tale soos **occam**. 'n Voorbeeld hiervan is 'n selektiewe kommunikasieprimitief wat die gebruik van beide toevoer- en afvoerwagte ondersteun.

Omdat 'n effektiewe implementasie nagestreef word, is die taal ontwerp om so nadruklik moontlik te wees, sonder om strukture in te sluit wat oneffektief is om te implementeer. Sekuriteit was egter die oorheersende oorweging in die ontwerp van die taal en looptyd omgewing.

Die vertaler lewer masjienkode, terwyl die meeste ander implementasies van **CSP**-agtige tale geinterpreteer word of ondersteun word as prosesse op 'n geskikte bedryfstelsel— die meeste **CSP**-agtige tale word slegs vir akademiese doeleindes aangewend. **LF** is by uitstek ontwerp as implementasie taal.

Die evaluasie van die stelsel se werkverrigting is gedoen aan die hand van praktiese maatstawwe soos die tyd wat benodig word vir kommunikasie, sowel as die gemiddelde tyd benodig vir die hantering van onderbrekings.

Acknowledgements

I gratefully acknowledge the help and support of all the individuals and companies who made this thesis possible.

- Dr. P.J.A de Villiers for his support, guidance and countless pep talks.
- SAN People for their generous bursary which made this paper possible.
- All of my friends especially Pierre and Hanli for helping me maintain my sanity.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To produce correct software is inherently difficult. Commercial software is typically put through exhaustive testing to ascertain its correctness. This approach not only consumes much time, but does not identify all the errors in programs. Formal methods, such as model-checking, have emerged to try and address the problem of checking program correctness. A major stumbling block in the use of model checkers is the disparity between the notations of formal specifications and the programming languages in general use. Many model checkers employ specification languages that are either derived from **CSP** [17] (such as SPIN's **Promela**) or check **CSP** directly (FDR[12]).

If one wishes to incorporate the use of model checkers into the software development cycle, one of two approaches is possible. A model of the program can be developed as part of the specification process. The model can then be used as a basis from which to derive the source code. Alternatively one may infer the model from existing source code.

The abstract process algebraic notation of **CSP** does not have a direct mapping onto conventional programming constructs. The ideal would be to have a programming language that is both useful and practical as an implementation tool, but also allows for easy translation to and from a formal **CSP** specification.

In this thesis I propose a language that strives to meet this challenge. The proposed language, **LF**, aims to be an implementation language with constructs similar to that of **CSP**. This will facilitate automatic or semi-automatic conversion to (and possibly from) formal **CSP** notation[1]. The greatest challenges in this endeavour are:

1. To design a language that is expressive enough for the purpose of programming embedded applications.

2. To ensure that no construct within the language compromises the efficiency of **LF** as an

[1] Indeed the process of translating **LF** to **Promela** and model checking it, is the topic of another Masters project currently under way [9].

implementation language.

3. To ensure that translation to the notation accepted by a model checker is possible.

LF is intended primarily for use in embedded systems. A precise definition of an embedded system is an elusive concept. Broadly speaking one may define embedded software as: *software executing on a platform with relatively limited resources, typically performing control functions that are intimately related to the hardware, but which are imperceivable to the user.* Embedded systems exhibit the following attributes:

- The embedded software is often a monolithic piece of code. It may therefore be implemented in a single secure programming language.
- Embedded software is intricately involved with the hardware on which it runs. This implies that the hardware should be readily accessible to the programmer.
- Furthermore, it generally requires efficient management of limited resources. Scheduling that is highly responsive to interrupts, is often required. Efficiency both in terms of performance and memory usage is important.
- The need for low level operations introduces many opportunities for corrupting the system. A compromise needs to be reached between making the language expressive enough for low level programming, and imposing sufficient constraints on the programmer to allow some measure of safe-guarding in the language.

LF is claimed to be a secure language. Hoare describes the concept of security in the following way [15]:

> '... Firstly, the notations (of a programming language) should be designed to reduce as far as possible the scope for coding error; or at least to guarantee that such errors can be detected by a compiler ... Certain programming errors cannot always be detected in this way, and must be cheaply detectable at run time; in no case can they be allowed to give rise to machine or implementation dependent effects that cannot be explained in terms of the language itself ... '

Simply put, a **secure system** is one in which code that adheres to the syntactic and semantic constraints of the language is guaranteed not to corrupt the system. A corrupt system is loosely defined as a system existing in a state that is not defined by the program. Semantic constraints refer to the "rules of use" of constructs and include typing constraints.

In practice such a strict form of security is not a viable proposition. A less rigorous form of security is needed. **LF** therefore offers the guarantee that code that adheres to the syntactic and semantic constraints of the language cannot lead to corruption of the allotted memory of a process by another process.

The need for a relaxed form of security arises primarily because of the use of embedded systems as the target environment. Such systems need to access hardware directly. This has the potential of placing the processor or other hardware components in inconsistent states. A simple example should suffice:

Assume that an embedded system consists of two processes one of which, p, is an interrupt handler and some other process, q, that inadvertently writes to the wrong I/O port. Let us further assume that this port controls the interrupt controller hardware. If the interrupt which is to be handled by p is now disabled, process p will be rendered useless. All of this has however taken place within the syntactic and semantic constraints of the language.

Unfortunately no practical way of imposing a safe hardware abstraction at this low level could be identified.

As a somewhat philosophical aside, one might argue that a program that conforms to the stated criteria for security, may indeed not be secure in the sense formerly defined if compiled by a faulty compiler. For the purposes of this thesis however we will assume a correct compiler implementation.

LF is based on the **Joyce** system proposed by Brinch Hansen [7]. Essentially **Joyce** (and **LF**) presents the programmer with the means to implement concurrent processes that may communicate by way of synchronous message passing. Apart from the fact that Joyce was not put to much practical use, the system employed a P-Code interpreter which is not an efficient execution environment. As such this project offers valuable insight into the practicality of the programming paradigm introduced by Joyce.

Outline of this thesis:

- In **Chapter 2** the grammar and semantics of **LF** are presented. The choice of language constructs and features are examined. Important similarities as well as differences with other **CSP**-derived languages such as **Joyce** and **occam** are discussed [7, 21]. A brief discussion of the compiler and code generation concludes the chapter.
- Some examples of **LF** code are presented in **Chapter 3**. These examples highlight the most important features of the language.
- The runtime system is explained in detail in **Chapter 4**. The design of the system is motivated and performance figures are presented.
- **Chapter 5** is a critical evaluation of the system both in terms of the language and the runtime system. Future extensions to the system are also discussed.

Chapter 2

LF: A Process oriented language

LF is a practical, secure, programming language with strong ties to **CSP**. The compiler and run-time checks enforce the integrity of the system, thereby obviating the need for hardware memory protection. In this chapter the grammar and semantics of **LF** are discussed. Considering the concurrent nature of the language, the syntactic and semantic definition of the language should not preclude future multiprocessor implementations. The language definition is presented as it pertains to a single processor implementation.

2.1 Language concepts

The basic notation of **LF** is similar to that of **Oberon** and other members of the **Pascal** family [36, 34]. For this reason and the sake of brevity, the focus of this discussion is on the features of the language which relate to concurrency. The complete grammar of **LF** in **EBNF** notation may be found in Appendix A.

2.1.1 Processes

An **LF** program consists of a set of communicating processes. An initial process is instantiated and may then spawn other (child) processes dynamically. Child processes execute concurrently with their parents. A simplified **EBNF** definition for an **LF** program is shown below.

$\langle\text{program}\rangle$ ::= `PROGRAM` *identifier*; $\langle\text{declarations}\rangle\langle\text{processes}\rangle$ $\langle\text{body}\rangle$.
$\langle\text{processes}\rangle$ ::= { $\langle\text{process}\rangle$ }
$\langle\text{process}\rangle$ ::= `PROCESS` *identifier*[$\langle\text{parameter list}\rangle$] ; $\langle\text{declarations}\rangle$ $\langle\text{body}\rangle$;
$\langle\text{body}\rangle$ ::= `BEGIN` $\langle\text{command list}\rangle$ `END` *identifier*

```
PROGRAM Null;
(*
  The simplest possible LF program.
*)
BEGIN (* main process *)
END Null.
```

Figure 1: The null program

The null program is listed in Figure 1. Process instantiation follows the same syntax as procedure invocation in most imperative languages. Examples of process instantiation may be found in lines 17 and 18 of Figure 2. The instantiation of processes is hierarchical in nature. The `BEGIN-END` block which is associated syntactically with the reserved word `PROGRAM`, is called the *main process.* Lines 16 to 19 of Figure 2 constitute a main process. In the main process, processes `P1` and `P2` are instantiated. Process `P2` is instantiated with the value parameter `x`. From line 01 it should be evident that `x` has the value 42. All processes with the exception of the *main process* have a parent. Unbounded (recursive) instantiation of processes is supported.

In keeping with **CSP**, no global variables are allowed. The use of global variables are inherently dangerous in concurrent systems, as it amounts to shared memory. The variables of a process are inaccessible to other processes. *The encapsulation of data is therefore done at process level.*

Parent processes only terminate once all their children have terminated. This, in addition to the tree structure of process invocation, simplifies memory management: An *activation record* is created for each instance of a process. The tree structure of activation records can be physically represented as a stack —with the top of the stack representing the most recently activated process. As processes terminate, their activation records are popped from the stack. Processes reaching the end of their execution are automatically terminated, provided that the above constraint is satisfied. In Figure 2 process `P1` will execute indefinitely while process `P2` will terminate after setting local variable `x` to 1. When a process terminates, its local variables are destroyed.

Processes may pass information to one another in one of two ways:

- **Inter-Process Communication** primitives which will be discussed in the next section.
- **Value parameters** are used for passing values to processes upon instantiation.

2.1.2 Communication

Processes communicate by way of messages which are transmitted over *channels. A channel constitutes the logical link between two communicating processes.* Processes create channels dynamically. *Channels* are accessed by way of *port variables.* Port variables are therefore references to the logical entities called channels and are instances of a type definition, a *port type.* An analogue exists in terms of dynamic memory. A port may be likened to a pointer and a channel likened to the

```
PROGRAM ProcessInstantiation;
(*
  This program defines and instantiates
  two additional processes.
*)
CONST
  x = 42;

PROCESS P1;
VAR
   x : UINT8;
BEGIN
  x := 0;
  WHILE TRUE DO
    x := x+1
  END
END P1;

PROCESS P2( x : UINT32 );
(*
   x is a value parameter
*)
BEGIN
  x := 0;
  x := x+1
END P2;

BEGIN
  P1; (* instantiate P1 *)
  P2( x ) (* instantiate P2 *)
END ThreeProcess.
```

Figure 2: A program defining two processes

memory which was dynamically allocated and is referenced by the pointer. Just as pointers may be required to point to instances of a specific type in languages such as **LF** and **Oberon**, a port also has a specific type.

```
⟨porttype⟩ ::= [ ⟨alphabet⟩ ] ;
⟨alphabet⟩ ::= ⟨symbol⟩ {, ⟨symbol⟩ }
⟨symbol⟩ ::= identifier [( ⟨type⟩ )]
```

Ports (as all types) in **LF** are typed using name equivalence. Types are associated with ports by way of *alphabets*. An alphabet consists of a fixed set of *symbols*. A *symbol* may optionally have a type associated with it. When no type is associated with a *symbol*, the symbol is referred to as a *signal*. As the name implies, *signals* are used to signal events. A signal does not copy any data. As such, the use of signals is more efficient in cases where messages are used only for synchronisation.

In Figure 3 a port type named `Chan` is defined with an alphabet consisting of symbols `a` and `b`. *Port variable* `chan` is an instance of *port type* `Chan`. The type of symbol `a` is `UINT32`. Symbol `b` is a signal.

A message is a value of the same type as the appropriate *symbol* in the appropriate *port type*, which is copied from one process to another, using the *channel* as logical communication medium. *Signals* do not strictly conform to the above definition. One can however regard all signals as having some implicit generic type. As far as the actual transmission of data is concerned, a signal does not require the actual transfer of data, but the runtime system still needs to perform all of the operations needed to transfer a normal message except for the actual copying of the data. A *symbol* does therefore not directly relate to an entity but is merely used to facilitate strict type checking. The use of symbols and messages will become apparent shortly.

During a given communication between two processes, a channel may carry any one (and only one) of these symbols. The programmer must explicitly state the direction of communication over the channel. The reserved words `IN` and `OUT` indicate this direction. Note that the semantics of **LF** therefore prohibits the same process from communicating, in both directions, over the same channel. This makes it impossible for a process to deadlock by communicating with itself over the same channel. Clearly few programmers will make this mistake during design, but it is easy to mistype a '`!`' for a '`?`' (See below).

Syntactically, communication in **LF** is similar to **CSP**. This is evident from the following **EBNF** definition.

```
⟨io⟩ ::= ⟨bang⟩|⟨hook⟩
⟨bang⟩ ::= ⟨variable access⟩ ! identifier[(⟨expression⟩)]
⟨hook⟩ ::= ⟨variable access⟩ ? identifier[(⟨variable access⟩)]
```

The statement `c ! a(i)` in Figure 4 sends the value of the variable `i` over channel `c`. Assume that the type `Chan` has the same definition as in Figure 3. Notice the type equivalence between

```
PROGRAM AlphabetDefinition;
TYPE
  Chan = [a(UINT32),b];
PROCESS P1(IN c : Chan);
BEGIN
END P1;

VAR
  OUT chan : Chan;
BEGIN
  P1(chan)
```

Figure 3: Defining a port type

```
VAR
  OUT c : Chan;
  IN d : Chan;
  i : UINT32;
BEGIN
  c ! a( i ); c ! b
  d ? a( i ); d ? b
```

Figure 4: Simple communication

the definition of symbol `b` and the actual variable `i`. Symbols therefore represent the set of the types of messages that may be sent, whereas the **value** of variable `i` is sent over the channels as the actual message. The statement `d ? a(i)` receives a value of type `UINT32` over the channel and assigns this value to variable `i`. The commands `c ! b` and `d ? b` are analogous but use signals.

When an **LF** process p (the sender) is ready to send a message to another process q (the receiver) which is ready to accept the message, p and q are said to *match* and communication is *feasible*. Strictly speaking the communication commands within the processes are matched, but in the interest of brevity these two meanings are taken to be equivalent. The *Bang* and *Hook* then execute simultaneously. Both processes will continue concurrently. This approach to message passing is also used by **Joyce** and **occam**. On a uni-processor system only a single process can actually execute at any given time. The precise order of execution is determined by the scheduler. Scheduling is discussed in Chapter 4.

Several processes may share a channel. Channel sharing is achieved by passing port variables as value parameters to processes. In Figure 5 processes `P1` and `P2` have access to a common channel through port variable `c`. Because of channel sharing processes may match in various combinations. In Figure 5, the *Hook* in process `P2` may match with any one of the instances of process `P1`. Communication takes place in a first-come-first-served manner in this event. When a communication statement is not matched, the process issuing the command (either a *Bang* or

```
PROGRAM Share;
TYPE
  Chan = [a(UINT8)];

PROCESS P1( c : Chan );
VAR
  i : UINT8;
BEGIN
  c ! a( i )
END P1;

PROCESS P2;
VAR
  c : Chan;
  i : UINT8;
BEGIN
  NEW( c );
  P1( c ); P1( c );
  c ? a( i )
END P2;

BEGIN
END Share;
```

Figure 5: Channel sharing

Hook) behaves in one of two ways. If the communication statement was used as part of a simple input/output command (a *Bang* or *Hook* command) the process is blocked until communication is *feasible*. A process may also issue a communication command as part of a guard in a SELECT statement as explained below.

⟨select⟩ ::= SELECT ⟨select guard⟩ THEN ⟨command list⟩
    { [] ⟨select guard⟩THEN⟨command list⟩}
END.
⟨select guard⟩ ::= ⟨variable access⟩⟨io⟩[&⟨expression⟩].

As shown in the above box, a SELECT construct consists of a set of guards, with a command list associated with each guard. Each guard consists of a communication statement with an optional boolean expression. A guard evaluates to true when the communication statement is matched and the boolean expression evaluates to true. The SELECT will non-deterministically choose one of the guards which evaluates to true and execute its command list. When no guards are true the construct as a whole blocks until a guard becomes true. The structure of a SELECT statement is outlined in Figure 6. Each $\mathtt{S}_i$ refers to a command list and each $\mathtt{chan}_i$ refers to a channel. Note the use of boolean expressions in the guards and the fact that these expressions refer to the variables used in the communication. The relevance of this facility is discussed in Section 2.3.4.

```
SELECT
  chan₁ ? c( x ) & x > 0 THEN
    S₁
[] chan₂ ! s( y ) THEN
    S₂
END
```

Figure 6: The structure of a SELECT

Note that two *SELECT* statements cannot match. The need for this restriction is explained in Section 2.3.4.

2.1.3 Other Language attributes

The syntax and semantics of assignments as well as the definition of variables, constants and types are similar to that of **Pascal** and more specifically, **Oberon**. The syntax of these constructs can be seen in the examples presented in Chapter 3. A complete **EBNF** definition may be found in Appendix A. The most notable language attributes not already discussed and the reasoning behind their use are outlined below.

- The **LF** runtime system transforms an interrupt event into a message. An array of ports is predefined. Each element of the array corresponds to one of the possible interrupts that may occur on a specific target platform. The size of the array is therefore implementation specific (it is referred to as `MaxInt` below). Listed below is a definition of the array —called `IntChannels`— as it would appear to the programmer.

  ```
  TYPE
    IntPort = [sig];
  VAR
    IntChannel = ARRAY MaxInt OF IntPort;
  ```

  This introduces a platform dependence into the language specification, but since only the size of the array changes on different platforms it is of little consequence to the programmer. This approach makes the use of the system consistent by not having special constructs related to interrupts. From the programmer's point of view interrupt handling does not represent a deviation from the general language constructs. Figure 7 shows a process that continuously waits on interrupt 32. The definition of `IntChannels` clearly assumes an underlying architecture which supports interrupt vectoring, but the scheme as proposed here will work on platforms which do not implement it explicitly. Interrupt vectoring can be cheaply implemented in the runtime system. The variable `IntChannels` is visible to the programmer, but not the type `IntPort`. Therefore one cannot declare a variable of type `IntPort`. The details of interrupt handling within the runtime system are discussed in Section 4.6. Chapter 3 contains a complete example of interrupt handling within an **LF** process. The language **occam**

```
BEGIN
  NEW( IntChannels[32] ); (* use of predefined array IntChannels *)
  WHILE TRUE DO
    IntChannels[32] ? sig
  END
```

Figure 7: Waiting on an interrupt

```
(* Accessing a memory mapped display *)
TYPE
  DisplayPage = ARRAY 2000 OF RECORD
    char,attr : UINT8
  END;
VAR
  dp : DisplayPage AT $b8000;
BEGIN
  dp[0].char = 65; (* write "A" to (0,0) on the screen *)
⋮
```

Figure 8: Variables at physical addresses

uses a similar abstraction. In **occam**, however, each channel is associated with a specific device driver implemented in the **occam** runtime system —so-called *hard channels* [26]. **LF** offers more flexibility than **occam** since **LF** does not implement device drivers as part of its runtime system.

- **LF** allows the programmer to define a data structure at a specific physical address. This allows the development of device drivers that need to access memory mapped devices. The display memory is a good example. The possible violation of the system integrity is minimised by the fact that the address value has to be a constant. Once a variable is defined at a physical address, no other variable may be defined within an overlapping memory range. The range is defined by the size of the variable. The compiler keeps a table which ensures that these memory ranges do not overlap. Of course only one instance of a process that declares a specific absolute variable may be instantiated. A bitmap is kept by the runtime system which sets a bit corresponding to each process defining such a variable. When the process is instantiated, the bit is set and when it is destroyed, the bit is cleared. Once the bit is set, the creation of another such process will result in a runtime exception. In Figure 8, an example of how to access a memory mapped display page is given for the **IBM** PC.

- Typed pointers are supported. No arithmetic operations are allowed on pointers. This limits (but does not completely remove) the inherent dangers of pointers. A heap essentially introduces shared memory, as would be the case with reference parameters. Only structured types (`RECORDs,ARRAYs`) may be allocated on the heap. This discourages the indiscriminate use of small allocations on the heap. The allocation of a 32-bit integer variable would for

```
TYPE
  PAr = POINTER TO ARRAY 10 OF UINT16;
  P = POINTER TO Rec;
  Rec = RECORD
    n : P
  END;
VAR
  r : Rec;
  p : PAr;
BEGIN
  NEW( r );(* initialise r *)
  r↑.n := NIL;
  NEW( p );
  p↑[10] := 0
⋮
```

Figure 9: Simple use of pointers

instance require up to 8 bytes of heap management overhead. It should be clear from Figure 9 that syntactically the use of pointers is identical to that of **Pascal**. Pointers are dereferenced with the "↑" character.

- All variables are zero initialised. This allows for the detection of the dereferencing of uninitialised pointers and the detection of references to uninitialised channels. All references to pointers and channels (which are represented in the runtime system by pointers) are compared to `0`. An uninitialised channel will have a pointer with the value `0` and a runtime exception will be generated in such a case. A pointer with the value `0` is not necessarily uninitialised, as the programmer may have assigned the value `NIL` to it, but dereferencing of a pointer with the value of `0/NIL`, will always cause a runtime exception. Variable initialisation may introduce significant runtime overhead upon process instantiation, but is vital in ensuring the security of the system. **All** variables are zero initialised as this is generally more efficient, both in terms of code size and execution time, than selective initialisation.

- Record structures are packed i.e. no padding to align fields on machine word boundaries is performed. This leads to a significant reduction in execution speed on most hardware platforms. Padding, however, wastes a substantial amount of memory. Moreover, padding causes problems with the allocation of variables at specific physical addresses.

- Separate compilation is not supported. The whole program needs to be in a single file. The system produces a single statically linked code image that is loaded by a boot-loader. The image contains both the **LF** object code and the runtime system. This is well suited to small embedded applications. The boot-loader as well as the structure of the static image are discussed in Chapter 4 and Appendix B.

2.2 Type rules

Strict type checking, by way of name equivalence, is enforced over communication channels and also applied to all other variables. Type compatibility between integer types is governed by special rules. Six integer types, representing signed and unsigned values in 8, 16 or 32 bits, are defined, `(U)INT32` being a 32 bit variable, `(U)INT16` a 16 bit variable and `(U)INT8` an 8 bit variable. `(U)INT16` and `(U)INT8` may be regarded as subsets of `(U)INT32`. Similarly `(U)INT8` is a subset of `(U)INT16`. The 'U' prefix refers to unsigned values in each case.

All integer types within either the signed **or** unsigned class may be compared. Assignment however requires that a subset type be assigned to a superset type. Unsigned and signed types are **not** assignment compatible, nor may they be compared.

2.3 Design considerations

This section motivates many of the design decisions that were made with respect to communication and other aspects of the language. The options pertinent to each aspect of communication are discussed in **general terms** and the ones adopted for **LF** are then explained.

Barring the value parameters passed at process initialisation, inter-process communication (*IPC*) is the only means of exchanging information between processes. *IPC* may be implemented by way of two primitives [25]:

- *send(x,message)*
- *receive(y,message)*

The variables x and y refer to the destination and source of the message respectively. A process P wishing to transmit a message to process Q does so by invoking the *send* primitive. The *send* and *receive IPC* primitives in **LF** are called *Bang(!)* and *Hook(?)*, as in **CSP**. The process Q accepts the message by way of the *receive* primitive. A communication link must exist which logically connects the receiving and sending processes.

2.3.1 Ports

To send messages, processes must be logically linked to each other. Some means of referencing or *naming* the other process must therefore exist. Naming may be either *direct* or *indirect. Direct naming* requires a communication partner to directly name the other. The *send* and *receive* primitives would therefore be defined as follows:

- $send(P, message)$

- $receive(Q, message)$

In the above P and Q are the process identifiers for the processes participating in the communication. Direct naming would not work well within the **LF** system as processes are created dynamically. Explicitly referencing a given instance of a process would be cumbersome if not impossible. Each instance of a given process would have to be explicitly referenced. These references need to be resolved by the compiler. As the number of instances is not known at compile time it is unclear how the compiler would be able to resolve the references.

With *Indirect naming* messages are sent to and received from a *port*. A port uniquely identifies each link. It therefore functions as a means of access to a common communication link. The port p must be visible to, and is used by, both the sender and receiver. This implies the following notation for *send* and *receive*. In this case p refers to a port.

- $send(p, message)$
- $receive(p, message)$

In general, the use of ports has the following properties:

- A link is established between a pair of processes only if they share a port.
- Multiple processes may be associated with a single port.
- Processes may be linked by multiple ports.
- Links may be either bi- or uni-directional. In other words a message may be passed in either direction or only one direction over the link respectively.

Indirect naming clearly offers considerable flexibility. The use of ports presents the implementer of the IPC primitives with a number of choices:

Multiple receivers

A process P may issue a *send* command to a given port. This port may for instance have two processes Q and R which use it, both of which execute a *receive*. The question of which of these two processes should receive the message arises. Note that allowing both processes to receive the message is not an option. Only one process is allowed to receive any given message. Three possible solutions are:

- Allow only two processes to be associated with a given port. One process to *send* and one to *receive*. This circumvents the problem, by removing it by construction. This approach is inflexible for the simple reason that it does not allow multiple receivers of a message.

- Allow only one process to issue a *receive* primitive at a given time. This is cumbersome as runtime checks would have to be performed on each *receive* operation.
- Allow the runtime system to decide. This clearly offers much more flexibility —in terms of both implementation options and having a non-restrictive language definition— than the other two options. This is the solution employed in **LF**. The method also has the distinct disadvantage of implementation dependence. Therefore no assumptions should be made as to the recipients of messages. This caveat in my view is far surpassed by the flexibility offered.

Port ownership and process termination

According to Silberschatz and Peterson, ports may be owned either by processes or by the runtime system [25]. Port ownership refers to who has the ability to create and destroy the ports. It was decided to let processes own the ports as this arrangement fitted well into the hierarchical process instantiation scheme. A process may declare a port variable. Such variables may be passed to new instances of processes as parameters. A port variable is a reference to a channel. The parent process has the responsibility of initialising the port by way of the `NEW` system call. This process is then the owner of the port. Since reference parameters are not allowed, a child process which inadvertently initialises a port which was passed to it as a parameter, only cuts off its own access to the port.

Processes terminate in a bottom-up order. It is therefore impossible for the owner of a port to terminate (and subsequently destroy a port) before its children (which may potentially also use the port) have terminated. It is however possible for a parent process to attempt communication with a child process which has already terminated. This may result in the indefinite suspension of the parent process. References to uninitialised ports result in a runtime exception.

Uni- and bi-directionality

LF implements uni-directional communication for a number of reasons. Uni-directional communication is employed by **CSP**. It is easier to implement than bi-directional communication. Bi-directional links would require additional queueing space as both senders and receivers on a link need to be queued as opposed to either one in the case of uni-directional links. It is unclear to the author when one would benefit from using bi-directional links, apart from perhaps having fewer links. Uni-directional links may be combined to simulate a bi-directional connection in such cases. The use of uni-directional links should also help to avoid some programming errors by forcing the programmer to explicitly state the direction of travel of messages over the link.

Blocking vs. Non-blocking primitives

The communication primitives may be implemented as either blocking or non-blocking. Blocking primitives enforce synchronous transfer of messages as a process is blocked until its communication partner is ready to send(receive) the message. It has the advantage that it provides for an intrinsic method of synchronising processes. More importantly, there is no need to buffer messages within the runtime system as the message remains in the address space of the sender until another process is ready to accept it. Since a process blocks on the first message sent, no flow control techniques are required.

Non-blocking primitives do not impose synchronisation on the processes. This may lead to a possibly unbounded growth in the length of the message buffer for the relevant channel. Apart from the obvious concerns of running out of memory, large buffers also introduce a response latency as new messages are delayed, as previously buffered messages are cleared. Even when an upper bound is placed on the size of the buffer, once the buffer is full, blocking again needs to be imposed. Asynchronous primitives do however allow a higher degree of parallelism, as processes do not need to wait for their communication partner to receive the message. Blocking primitives are also significantly easier to implement than their non-blocking counterparts [22, 29]. Gehani presents similar arguments to the above in [13]. Blocking primitives are used in systems such as **Amoeba** and **V** [23, 5]. The **LF** system employs blocking primitives.

Loss or corruption of messages

When dealing with a distributed implementation of the **LF** system, the occurrence of message loss or corruption is a distinct possibility. For the purposes of the language definition however, a reliable transmission medium is assumed. The choice of reliable transfer protocol is left to the implementer of the runtime system.

2.3.2 Reference parameters

Reference parameters are not supported. Value parameters are supported to allow for startup conditions to be set. This allows for some measure of flexibility, without the dangers of reference parameters. The use of reference parameters would require complex mutual exclusion constructs in the runtime system. Furthermore, the implementation of reference parameters on a non-shared memory multiprocessor, would have to be implemented along similar lines to communication over channels. One would then have two syntactic constructs linked to the same runtime implementation. The extra complexity is therefore unwarranted. The functionality of reference parameters is implemented using **IPC**. Chapter 3 demonstrates how to use communication instead of reference parameters.

```
PROCESS Buffer(IN prod : Chan; OUT cons : Chan);
VAR
  buffer : ARRAY 9 OF CHAR;
  notFull,notEmpty : BOOLEAN;
  in,out : UINT32;
BEGIN
  (* initialise *)
  WHILE TRUE DO
    SELECT
      prod ? buffer[in] & notFull THEN
          S1
      [] cons ? sig(); notEmpty THEN
         cons ! buffer[out];
         S2
    END
  END
END Buffer;
```

Figure 10: No output guards

2.3.3 Process instantiation

LF allows for the dynamic creation of processes (as does **Joyce**[7]). This instantiation may be unbounded(recursive). **CSP** allows only for a fixed number of processes all of which terminate simultaneously. The dynamic instantiation of processes removes the need for indexed processes as in **CSP** and **occam**. Recursion was added with little complexity to the runtime system. The considerable elegance and flexibility of recursion surely justifies the added effort.

2.3.4 Polling semantics

The original **CSP** and **occam** both restrict polling to input commands [16]. In other words only *Hooks* and no *Bangs* are allowed as part of the guards of a SELECT statement. The **occam** equivalent of the **LF** SELECT is the ALT construct. **Ada** contains the `select-accept` construct. All of these constructs are based on the **CSP** general choice operator ('□'). This prevents a sender and receiver from polling the same channel at the same time. It was decided in the interest of flexibility and symmetry to allow output polling as well. Furthermore, Hoare states that output guards are 'mathematically possible' within the model of **CSP** [17]. A classic example of the advantage of allowing output guards in a SELECT is adapted from Raynal [27]. It shows the implementation of a buffer. First without the use of an output guard, and then with the use of one. Notice how in Figure 10 the required output is performed in two stages. Firstly the producer and consumer are synchronised. Secondly the desired output is performed. In Figure 11 an elegant, symmetrical, solution to the same problem, is presented.

Two SELECT statements cannot match. A process executing a SELECT can only match with a

```
PROCESS Buffer(IN prod : Chan; OUT cons : Chan);
VAR
  buffer : ARRAY 9 OF CHAR;
  notFull,notEmpty : BOOLEAN;
  in,out : UINT32;
BEGIN
  (* initialise *)
  WHILE TRUE DO
    SELECT
      prod ? buffer[in] & notFull THEN
        S1
      [] cons ! buffer[out] & notEmpty THEN
        S2
    END
  END
END Buffer;
```

Figure 11: Output guards

process executing an input/output command (a *Bang* or *Hook* command). This is similar to the scheme proposed by Silberschatz [30] and the scheme used in **Joyce**.

A number of 'enhanced' implementations of **CSP** have been proposed which generalise the SELECT construct. This generalisation allows for output guards and/or the ability for SELECT statements to match. Various protocols have been suggested to implement this functionality, for instance one by Bernstein [6]. Buckley and Silberschatz give an overview of several of these protocols, as well as defining one of their own [8]. Their protocol, albeit one of the more general protocols, was described by Raynal as *'cumbersome and complicated'* [27]. The basic problem in the implementation of a generalised input-output construct stems from the possible number of guards that need to be evaluated. Every guard G_i in a SELECT statement must be compared to every guard G_j in every other process it references (be it via ports or direct naming). If one factors in the unbounded instantiation supported by **LF** the problem becomes even more daunting. Moreover, no example could be envisioned in which matching SELECTs would be necessary.

It is the programmer's responsibility to ensure that a SELECT statement is matched by an input/output statement. If this restriction is violated the SELECT statements will **never terminate**, effectively blocking the processes containing it indefinitely and possibly leading to the deadlock of the system as a whole. One might view this as a flaw in the language design, which will lead to many defective programs. However, the security of the system is not compromised and the approach taken is much more efficient than the alternatives. Potential deadlocks are also relatively easy to detect by using a model checker.

LF allows for the specification of a boolean guard in conjunction with the input/output guard. Moreover, the boolean guard is allowed to reference the variables involved in the input/output guard. **SR** also allows this, but does not allow output guards [4]. Allowing such boolean guards makes the language more expressive. Essentially it allows a process to examine the message to see whether it *wants* to accept it. The alternative would be to have the process accept all messages and then re-send them to another process, or implement some sort of buffering scheme if it wishes to use the information in the message at a later stage.

2.3.5 Channel sharing

If every channel were to be used by two processes only, a resource process would have to be connected to users by means of a channel array. Reasonable performance is expected from shared channels for 'lightly' used channels. Brinch Hansen presents the same argument in [7]. Fairness problems may however arise for channels carrying large amounts of traffic. A quantifiable definition of light usage as well as the extent of fairness problems which may arise will vary from application to application and remains a topic for investigation.

2.4 Security claims

It is claimed that **LF** processes cannot adversely affect each other's address spaces. In conventional programming languages memory interference may occur in the following circumstances:

- **Stack overflows** are a common cause of corruption in programs. The **LF** compiler determines the maximum stack depth for a process at compile time and consequently allocates sufficient stack space.
- **Shared memory** manifests itself in different incarnations:
  - The use of global variables are a common cause of software failures. **LF** (as **CSP**) does not allow the use of global variables.
  - Reference parameters in the context of concurrent programs are dangerous because they essentially allow for the aliasing of a variable; allowing multiple processes to change the same variable without any form of synchronisation.
  - **LF** places severe restrictions on the use of pointers. Pointer arithmetic is disallowed. Pointers are strongly typed. Pointers may only point to variables i.e. pointers may not be assigned an arbitrary value.
- **LF** allows for variable instances at specified addresses by way of the `AT` construct. This construct is also subject to restrictions. Only a single instance of such a variable is allowed. This implies that only one instance of a process instantiating such a variable may be allowed. A runtime check enforces this constraint. The ranges of memory that are available for use by such variables are implementation specific. Such addresses are intended to correspond to those addresses used by memory mapped devices, and should therefore not overlap with the memory used by either process activation records or the runtime system. Compiler and runtime checks enforce this requirement.

`INT8,INT16,INT32`	The signed integer types
`UINT8,UINT16,UINT32`	The unsigned integer types
`CHAR`	The character type
`BOOLEAN`	The boolean type
`SET8,SET16,SET32`	The set types

Table 1: Predefined types

- Strict **index checking** is performed on arrays.
- Runtime checks trap references to uninitialised channels, and dereferences of `NIL` pointers. As noted, all pointers are initialised to `NIL` by the compiler, which makes the dereferencing of uninitialised pointers impossible.
- Overflows on arithmetic operations are detected by the runtime system.

Additional checks are also performed:

- Runtime checks detect when the system has run out of memory.
- Strict type checking as described in Section 2.2 ensures that variables of different types are not used interchangeably.

In the event of the failure of a runtime check, a machine exception is generated. The exception handler may be defined by the programmer by writing a handler process for the appropriate interrupt. Runtime failures however almost exclusively indicate a serious program error. By default, execution of the system in its entirety is stopped. Note that it is possible to redefine an exception handler that is implemented in the runtime system. This could render the system insecure. It was decided not to make any distinctions between the interrupt handlers that may be defined as this would lead to unnecessary overhead.

2.5 Predefined entities and implementational dependencies

LF predefines a number of entities. Foremost of these are the standard types as listed in Table 1.

Sets are generally implemented in one of two ways. Either as having a direct mapping onto a word in the target machine (**Oberon** and **LF**), or having some other arbitrary size. In **Pascal** the size of the set representation is implementation specific. Values typically range from `64` to `2040` members [10]. Compilers generally allow the cardinality of sets to be at least as large as that of the `char` type [10]. The approach of **Oberon** is efficient as it can make use of specialised machine instructions. A major disadvantage of using sets which are mapped onto a machine word becomes apparent when dealing with hardware devices. In such cases it is convenient to manipulate bits by using sets. More often than not, these bits need to be read(written) from(to) a hardware port. These ports may not have the same number of bits as the machine word. A proliferation of type

```
(* Oberon code *)
VAR
  s : CHAR;
BEGIN
  PORTIN( 123,s );
  IF 1 IN SYSTEM.VAL( SET,s ) THEN
```

Figure 12: Sets in **Oberon**

```
(* LF code *)
VAR
  s : SET8;
BEGIN
  PORTIN( 123,s );
  IF 1 IN s THEN
```

Figure 13: Sets in **LF**

casts then appears. The author has observed that failure to type cast the values read from ports in **Oberon** to the correct size has been a consistent source of program failures among novice programmers.

By using sets of differing sizes, **LF** syntactically does away with these type casts. Note that no runtime efficiency is gained as zero extensions of smaller data types to that of a machine word are still needed to use the specialised machine instructions. Figures 12 and 13 compare the usage of sets in the manipulation of hardware ports. Note the use of the `SYSTEM.VAL` type cast mechanism in the **Oberon** code, which is not needed in **LF** .

The integer and set types are listed in Table 1 as defined for the current implementation (Intel *i386*). Additional types may be added (for instance in the case of a 64-bit architecture). Similarly the data types used by the predefined procedures may also vary according to the target platform. The set types are intended to evolve in a similar fashion.

As was noted in Section 2.1.3 the array of ports `IntChannels`, is also predefined with an implementational dependence. This dependence is, however, less intrusive than variations in the size of the machine word.

Two constants **TRUE** and **FALSE** referring to the boolean values are defined. Finally **LF** also predefines a number of routines as listed in Tables 2 to 5.

2.6 The LF compiler

The **LF** experimental system generates code in two stages. The first stage is a single pass compiler which accepts the **LF** source code and generates assembly listings as output. A separate assembler

`ORD( a ) : UINT8`	Returns the ordinal value of character a.
`CHR( a ) : CHAR`	Returns the character (ASCII)value of ordinal value a.
`INC( a,b )`	Increments an integer variable *a* (signed or unsigned) by *b*. Note that both parameters must be signed or both unsigned.
`DEC( a,b )`	Decrements an integer variable *a* (signed or unsigned) by *b*. Note that both parameters must be signed or both unsigned.
`ABS( a ) : Signed Integer type`	Returns the absolute value of integer variable *a*. The result is a signed integer type with the same width as the argument.

Table 2: General functions

`INCL( a,b )`	Sets bit number *b*(unsigned) from set variable *a*.
`EXCL(a,b )`	Clears bit number *b*(unsigned) from set variable *a*.
`IN( a,b ) : BOOLEAN`	Check whether bit number *b* is set within set variable *a*. The same sign convention as above applies.
`SHL( a,b ) : Integer Type`	Performs a left bitwise shift of *b*(unsigned) bytes for integer variable *a*(any sign). The result is of the same type as the *a* argument.
`SHR( a,b ) : Integer Type`	Performs a right bitwise shift of *b*(unsigned) bytes for integer variable *a*(any sign). The result is of the same type as the *a* argument.
`AND( a,b ) : SetType`	Performs the bitwise AND operation on two variables of set types.
`OR( a,b ) : SetType`	Performs the bitwise OR operation on two variables of set types.
`XOR( a,b ) : Set Type`	Performs the bitwise XOR operation on two variables of set types.

Table 3: Bitwise functions

`PORTIN( a ) : Integer OR Set Type`	Reads a value from hardware port number *a*.
`PORTOUT( a,b )`	Writes the Integer OR SET value *b* to port number *a*. The number of bytes written depends on the width of *b*.

Table 4: I/O functions

`NEW( ptr )`	Creates a new instance of a given pointer or port type.
`DISPOSE( ptr )`	Releases the memory allocated by `NEW`.
`MEMFREE() : UINT32`	Returns the amount of free memory.
`MAXFREE() : UINT32`	Returns the size of the largest contiguous block of heap memory.
`NEWDMA( ptr,size )`	Allocated a DMA buffer.
`DISPOSEDMA( ptr )`	Releases the buffer allocated by NewDMA.
`RELEASEDMA( ptr )`	Relinquishes a given amount of memory allocated for DMA to the heap manager.

Table 5: Heap manager functions

```
PROCEDURE GetReg( VAR r : LONGINT );
VAR i : INTEGER;
BEGIN
  WHILE (i < 32) & (i IN regs) DO INC( i ) END;
  INCL( regs,i ); r := i
END GetReg;
```

Figure 14: GetReg for *rArch*

then produces the object code from these listings [32]. This route was taken because it allows the programmer to hand optimise speed critical sections of his code with relative ease. Although this clearly impacts on the security of the code generated, one should bear in mind that this is an experimental system. An optimising compiler falls beyond the scope of this thesis. The use of assembler listings made it possible to experiment with earlier incomplete compiler implementations. This section is intended to give the reader a broad idea of the quality of code generated to place the performance measures that will be presented in Chapter 4 into perspective. Parsing was implemented through standard recursive descent techniques.

2.6.1 Code generation

If one considers the fact that this is not an optimising compiler, very reasonable code is produced. The code generation of the compiler is based on a scheme used by Wirth to generate code for a theoretical RISC architecture which I shall call *rArch* (Chapter 9 of Wirth [35]).

Wirth's scheme was developed for a non-optimising compiler for a subset of **Oberon**. It was therefore a matter of applying the techniques used by Wirth to the Intel *i386* CISC architecture; the target platform. The most important problem that had to be contended with was the scarcity of registers in the *i386*.

The first challenge was therefore to develop a scheme to utilise this limited resource. Since this was not an exercise in compiler design a simple approach was favoured. With *rArch* the compiler designer has 32 general purpose registers at his disposal (R0..R31). This, for instance, allows Wirth the luxury of dedicating a register to contain the value 0.

Wirth uses a very simple method of register allocation. The registers are numbered 0 to 31. When a register is required the first available one is returned by a procedure called `GetReg` (Figure 14). The variable **`regs`** is a globally defined set. The reader will notice that the assumption is made that a free register is always available. This is a valid assumption considering the number of registers and the relative simplicity of expressions encountered in practice. A register is freed whenever its value is no longer needed. Because so few registers are available, no attempt is made to keep frequently referenced variables in registers.

With the *i386* four general purpose registers are placed at the disposal of the compiler designer

```
CONST
  rEBX=0; rEAX=3; rEDI=4; rESI=5;

PROCEDURE GetAdrReg( VAR r : LONGINT );
VAR i : INTEGER;
BEGIN
  i := rEDI;
  WHILE (i < rESI+1) & (i IN regs) DO INC( i ) END;
  INCL( regs,i ); r := i;
  IF i = 4 THEN Halt("Out of registers") END
END GetAdrReg;
```

Figure 15: GetAdrReg for *i386*

`EAX`,`EBX`,`ECX` and `EDX`. Two additional registers are available for indexing (`EDI` and `ESI`) [1] Each of the former four registers encapsulates a 16-bit register. For example `AX` maps onto the lower 16-bits of `EAX`. Similarly the 16-bit registers encapsulate two 8-bit registers i.e. `AL` and `AH` map onto the low order and high order byte of `AX` [2]. `ESI` and `EDI` do not have this sub-division. This forces the compiler designer to only use `EAX` through `EDX` for the evaluation of expressions. One may of course go to the trouble of implementing highly specialised code for dealing with 32-bit values only, but the added functionality is hardly worth the effort. The already dire situation is compounded by the highly irregular nature of the *i386* instruction set. The registers `EAX` and `EDX` are specifically required for a number of instructions, notably division and multiplication.

The above constraints led to the development of a register allocation mechanism which deviates slightly from that of Wirth's. Two different allocation procedures are used. It was decided to allocate `EDI` and `ESI` to store the addresses of variables to which assignments are made, so-called **lvalues** (Figure 15). The basic operation of the algorithm in Figure 15 and Figure 16 entails the following:

The globally defined `regs` is tested for inclusion of the presence of the bit corresponding to the appropriate register. If the register has already been allocated then the bit will be set. If the register is in use then the next register is examined. When a free register is found, the register is reserved by setting the pertinent bit.

Another procedure allocates general purpose registers for the evaluation of expressions (Figure 16). The `EAX` register is never available for allocation. This leads to a significant reduction in complexity in implementing the instructions which explicitly require this register. The alternative would have been to implement complex register shuffling, possibly augmented with store and load instructions.

The `EDX` register is allocated last since it is used in most of the instructions which specifically require `EAX` as well. As the code fragments (Figures [15,16]) indicate the compiler terminates

[1]The register `ESP` is used as the stack pointer and `EBP` is used to point to activation records.

[2]For the sake of brevity the name `EAX` may be used —depending on the addressing mode— in instances which actually refer to `AX` or `AL`

```
PROCEDURE GetReg( VAR r : LONGINT );
VAR i : INTEGER;
BEGIN
  i := 0;
  WHILE (i < 4) & (i IN regs) DO INC( i ) END;
  INCL( regs,i ); r := i;
  IF i = 4 THEN Halt("Out of registers") END
END GetReg;
```

Figure 16: GetReg for *i386*

```
VAR i : UINT32;
    b : UINT8;
BEGIN
  i := i+b
⋮
```

Figure 17: An expression requiring sign extension

execution in the event of running out of registers. This is clearly not an ideal situation. The fact that this is only an experimental version of the compiler and that the scheme works well in most cases is regarded as enough of a justification for the less than optimal implementation. Another simplification made to the register allocation scheme relates to the sub-division of registers. When an 8-bit operand is loaded into a register, the entire 32-bit register is allocated. This simplifies the implementation by virtue of its generality. If the registers were sub-divided in such cases the compiler would have eight more registers at its disposal when processing 8-bit expressions.

Another complication that had to be dealt with was the fact that the *rArch* only implements 32-bit operands (with the exception of the `load` and `store` instructions). The *i386* has 8-,16- and 32-bit operands. The compiler must generate type casts when processing expressions containing data types of differing sizes. A simple example of an expression requiring operand extension is presented in Figure 17.

This generates the assembly code shown in Figure 18. Zero- and sign-extensions often lead to the allocation of an additional register. The extension instructions require a register as target

```
LEA EDI,-4[EBP]     ; load linear address of 'i'
MOV EBX,-4[EBP]     ; load value of 'i' into EBX
MOV CL,-9[EBP]      ; load value of 'b' into ECX
MOVZX ECX,CL        ; zero extend contents of CL to ECX
ADD EBX,ECX         ; add ECX to EBX - result stays in EBX
MOV [EDI],EBX       ; store result
```

Figure 18: Sign extension

operand. It is therefore not only good programming practice to avoid mixing data types as much as possible, but may also mean the difference between being able to compile a complex expression or not.

The **LF** compiler produces labels instead of offsets when emitting control transfer instructions (`JUMP`,`CALL` and `RET` instructions). This makes it significantly easier to alter the resultant assembly listings. Unfortunately this requires a relatively large amount of bookkeeping to keep track of the textual label.

2.6.2 Examples of code generated

A few examples of the code generated are now presented. Comparisons are made to the code generated by the **Oberon** compiler for similar structures. One should bear in mind that the Oberon compiler is significantly more complex than the **LF** compiler, and therefore has an advantage when one compares the quality of the code produced. Please note that neither the code to create activation records nor the code to allocate and initialize local variables for either **LF** or **Oberon** is shown.

Code for common loop operations are shown below, in both **Oberon** (left) and **LF** (right):

```
PROCEDURE P;
VAR
 i : LONGINT;
BEGIN
 i := 0;
 WHILE i < 10 DO
  INC( i )
 END
END P;
```

```
PROCESS P;
VAR
 i : UINT32;
BEGIN
 i := 0;
 WHILE i < 10 DO
  INC( i,1 )
 END
END P;
```

```
PROCEDURE P                              labP0:
000DH: mov   ebx,0                       LEA EDI,-4[EBP]
0012H: mov   -4[ebp],ebx                 MOV [EDI],DWORD 0
0015H: jmp   9  (00000023H)
001AH: mov   ebx,-4[ebp]                 labWHILE1:
001DH: add   ebx,1                       MOV EBX,-4[EBP]
0020H: mov   -4[ebp],ebx                 CMP EBX,10
0023H: mov   ebx,-4[ebp]                 JAE labExitWHILE1
0026H: cmp   ebx,10                      labStatement1:
0029H: jl    -17  (0000001AH)            LEA EDI,-4[EBP]
                                         INC [EDI]
                                         JMP labWHILE1
                                         labExitWHILE1:
```

The reader will notice that the code produced by the **LF** compiler for the above example is virtually identical to that of the **Oberon** compiler.

```
PROCEDURE P;                             PROCESS P;
VAR                                      VAR
 i : LONGINT;                             i : UINT32;
BEGIN                                    BEGIN
 WHILE TRUE DO                            WHILE TRUE DO
  INC( i )                                 INC( i,1 )
 END                                      END
END P;                                   END P;

PROCEDURE P                              labP0:
000DH: jmp   9  (0000001BH)              labWHILE0:
0012H: mov   ebx,-4[ebp]                 labStatement0:
0015H: add   ebx,1                       LEA EDI,-4[EBP]
0018H: mov   -4[ebp],ebx                 INC [EDI]
001BH: mov   bl,1                        JMP labWHILE0
001DH: cmp   bl,1                        labExitWHILE0:
0020H: jz    -16  (00000012H)
```

In the above example the **LF** compiler actually outperforms the **Oberon** compiler. The **LF** specially optimizes the `WHILE TRUE DO` construct, as it occurs frequently in **LF** programs.

```
PROCEDURE P;
VAR
 i : LONGINT;
BEGIN
 i := 0;
 IF i = 0 THEN
  i := 1
 ELSIF i = 2 THEN
  i := 3
 ELSE
  i := 0
 END
END P;
```

```
PROCESS P;
VAR
 i : UINT32;
BEGIN
 i := 0;
 IF i = 0 THEN
  i := 1
 ELSIF i = 2 THEN
  i := 3
 ELSE
  i := 0
 END
END P;
```

```
000DH: mov  ebx,0
0012H: mov  -4[ebp],ebx
0015H: mov  ebx,-4[ebp]
0018H: cmp  ebx,0
001BH: jnz  13  (0000002EH)
0021H: mov  ebx,1
0026H: mov  -4[ebp],ebx
0029H: jmp  33  (0000004FH)
002EH: mov  ebx,-4[ebp]
0031H: cmp  ebx,2
0034H: jnz  13  (00000047H)
003AH: mov  ebx,3
003FH: mov  -4[ebp],ebx
0042H: jmp  8  (0000004FH)
0047H: mov  ebx,0
004CH: mov  -4[ebp],ebx
```

```
labP0:
LEA EDI,-4[EBP]
MOV [EDI],DWORD 0
MOV EBX,-4[EBP]
CMP EBX,0
JNE labnextGuard1
labStatement1:
labshort2:
LEA EDI,-4[EBP]
MOV [EDI],DWORD 1
JMP labexitGuard1
labnextGuard1:
MOV EBX,-4[EBP]
CMP EBX,2
JNE labnextGuard3
labStatement3:
labshort4:
LEA EDI,-4[EBP]
MOV [EDI],DWORD 3
JMP labexitGuard1
labnextGuard3:
labshort5:
LEA EDI,-4[EBP]
MOV [EDI],DWORD 0
labexitGuard1:
```

Again the **LF** and **Oberon** generated code is virtually identical. Code to traverse a linked list is shown below, in both **Oberon** (left) and **LF** (right):

```
PROCEDURE T;
TYPE
 Node = POINTER TO TNode;
 TNode = RECORD
  next : Node
 END;
VAR
 n : Node;
BEGIN
 n := NIL;
 IF n # NIL THEN
   n := n.next
 ELSE
  HALT( 80 )
 END
END T;
```

```
PROCESS T;
TYPE
 Node = POINTER TO TNode;
 TNode = RECORD
  next : Node;
 END;
VAR
 n : Node;
BEGIN
 IF n # NIL THEN
  n := n^.next
 ELSE
  HALT( 80 )
 END
END T;
```

As is evident from the listings below, the manipulation of pointers compares favourably with the code generated by the **Oberon** compiler. Essentially the only difference is the extra load instruction that the **LF** compiler generates (LEA) during assignments.

```
PROCEDURE T
000DH: mov     ebx,0
0012H: mov     -4[ebp],ebx
0015H: mov     ebx,-4[ebp]
0018H: cmp     ebx,0
001BH: jz      13  (0000002EH)
0021H: mov     ebx,-4[ebp]
0024H: mov     ebx,0[ebx]
0026H: mov     -4[ebp],ebx
0029H: jmp     3  (00000031H)
002EH: push    80
0030H: int     3
0031H: mov     esp,ebp
```

```
labT0:
MOV EBX,-4[EBP]
CMP EBX,0
JE labnextGuard0
labStatement0:
MOV EBX,-4[EBP]
MOV EBX,[EBX]
labshort1:
LEA EDI,-4[EBP]
MOV [EDI],EBX
JMP labexitGuard0
labnextGuard0:
PUSH 80
INT 3
labexitGuard0:
```

By default the **Oberon** compiler does not generate overflow checks. The reader will notice that the **LF** compiler generates extra instructions associated with the IMUL instruction when the code fragments shown below are compiled. The multiplication instructions for signed multiplication IMUL and unsigned multiplication MUL differ significantly in the addressing modes they allow, MUL being much more restrictive. The **LF** compiler generates code to conform to the restrictions of MUL for IMUL as well.

```
PROCEDURE T;                          PROCESS T;
VAR                                   VAR
 l,m : LONGINT;                        l,m : INT32;
 a,b : SHORTINT;                       a,b : INT8;
BEGIN                                 BEGIN
 l := m*a+b                            l := m*a+b
END T;                                END T;
```

Incidentally **Oberon** only supports signed data types. If **Oberon** were to support unsigned data types, I cannot foresee the code differing much from that generated by **LF** .

```
PROCEDURE T                           labT0:
000FH: movsx   ebx,byte ptr -9[ebp]   MOV EBX,-8[EBP]
0013H: mov     edx,-8[ebp]            MOV EAX,EBX
0016H: imul    edx,ebx                CDQ
0019H: movsx   ebx,byte ptr -10[ebp]  MOVSX ECX,BYTE -9[EBP]
001DH: add     edx,ebx                IMUL ECX
001FH: mov     -4[ebp],edx            INTO
0022H: mov     esp,ebp                MOV EBX,EAX
                                      MOVSX ECX,BYTE -10[EBP]
                                      ADD EBX,ECX
                                      INTO
                                      labshort0:
                                      LEA EDI,-4[EBP]
                                      MOV [EDI],EBX
```

When accessing array indices the **LF** compiler compares less favourably. The **LF** compiler generates fairly explicit code. It does not make use of the more efficient addressing mode used by the **Oberon** compiler.

```
PROCEDURE P;                          PROCESS P;
VAR                                   VAR
 a : ARRAY 100 OF RECORD               a : Array = ARRAY 100 OF RECORD
       a,b : LONGINT                         a,b : UINT32
     END;                                  END;
 i : LONGINT;                          i : UINT32;
BEGIN                                 BEGIN
 a[i].a := 20                          a[i].b := 20
END P;                                END P;
```

```
PROCEDURE Array                            PROCESS Array;
0023H:  mov  ebx,-804[ebp]                 MOV EBX,-804[EBP]
0029H:  cmp  ebx,100                       CMP EBX,100 ;Index check
002CH:  jb   3  (00000031H)                JB 3
002EH:  push 7                             PUSH 7
0030H:  int  3                             INT 3
0031H:  mov  edx,20                        SHL EBX,3
0036H:  mov  -800[ebp +  8* ebx],edx       LEA EDI,-800[EBP]
                                           ADD EDI,EBX
                                           ADD EDI,4
                                           labshort0:
                                           MOV [EDI],DWORD 20
```

2.6.3 Final remarks

The inadequacies of the compiler have been exposed in this section. They have been tolerated for the most part because of the fact that this is a first implementation. I however feel that even when considering the deficiencies of the compiler it fulfills the role of a suitable tool for conducting experiments.

Chapter 3

LF examples

In this chapter simple examples of **LF** programs are presented to demonstrate basic usage of the language. The initial examples are not meant to be of any practical value, but illustrate the basic concepts of the language. The subsequent examples are more relevant to the typical use of **LF**.

3.1 Simple

The example below illustrates the three most basic constructs of **LF**: The `WHILE`, assignment and the `IF-ELSIF-ELSE` constructs. Assignment should look familiar to **Pascal** programmers. The `WHILE` construct is identical to that of **Oberon** and is also equivalent to the **Pascal** `WHILE` except for the mandatory `END` (Line 17). The `IF-ELSIF-ELSE` construct has no direct equivalent in **Pascal**, but is identical to the **Oberon** construct. In **Pascal**, a nested `IF-ELSE` tree would be equivalent. Note the use of the **#** as the not equal operator instead if the **<>** of **Pascal**.

```
00 PROGRAM Simple;

01 PROCESS Basic;
02 VAR
03   x,y : UINT32;

04 BEGIN
05   x := 0
06   WHILE x < 10 DO
07     x := x+1;
08     IF x >= 8 THEN
09        y := 0
10     ELSIF x <= 4 THEN
```

```
       y := 1
    ELSIF x # 0 THEN
       y := 2
    ELSE
      y := 3
    END
  END
END Basic;

BEGIN
  Basic
END Simple.
```

3.2 Generate

All of the following examples, up to Example 3.7, are adaptations of **Joyce** examples presented by Brinch Hansen [7]. When comparing the **Joyce** and **LF** code, it is apparent that the two languages are indeed very similar.

In this example, the alphabet of the channel type `Stream` consists of the symbols `item` and `eos` (end of stream). The symbol `item` is of the type `INT32` [1] and the symbol `eos` is a signal.

```
TYPE
  Stream = [item(INT32),eos];

PROCESS Generate( OUT out : Stream; a,b,n : INT32 );
(* Generates a stream of INT32's *)
VAR i : INT32;
BEGIN
  i := 0;
  WHILE i < n DO
    (* an item (of type INT32) is sent via port "out" over a channel *)
    out!item( a+i*b); INC( i )
  END
END Generate;
```

The line `out!item(a+i*b)` evaluates the expression `a+i*b` and then sends this value over the channel associated with port `out`. Notice the type correspondence between symbol `item` and the expression. The process will go through `n` such iterations before terminating. In all of the following examples, it is assumed that the type `Stream` is defined as in example *Generate*.

[1]Recall that `INT32` is a 32-bit signed integer

3.3 Copy

The example named *Copy*, illustrates the use of the SELECT construct. Notice the use of two guards in the SELECT. The first guard reads the value of x over the channel linked to port in. The second guard reads the signal eos from the same channel. The semantics of the SELECT construct is such that a non-deterministic choice is made between the guards. If none of the guards are TRUE then the construct as a whole blocks execution of the process until such time as a guard does evaluate to TRUE.

Process Copy will accept any number of 32-bit signed integer values from its input channel in and transmit them via its output channel out until an eos signal is received.

```
(* A process to copy INT32 values from one stream to another *)
PROCESS Copy( IN in : Stream; OUT out : Stream );
VAR
  more : BOOLEAN; x : INT32;
BEGIN
  more := TRUE;
  WHILE more DO
    SELECT
      in?item(x) THEN out!item(x)
      []in?eos THEN more := FALSE
    END;
    out!eos
  END
END Copy;
```

3.4 Merge

The example *Merge* again makes use of the non-deterministic nature of the SELECT to create an arbitrary interleaving of two streams. The process waits for an eos signal from each of the channels before terminating.

```
(* A process that outputs an arbitrary interleaving of two streams *)
PROCESS Merge( IN in1,in2 : Stream; OUT : Stream );
VAR
  n,x : INT32;
BEGIN
  n := 0;
  WHILE n < 2 DO
    SELECT
      in1?item(x) THEN out!item(x)
      []in2?item(x) THEN out!item(x)
      []in1?eos THEN INC(n)
      []in2?eos THEN INC(n)
    END
  END
END Merge;
```

3.5 Suppress

This example removes duplicate values from its input stream and outputs the result to its output stream.

```
(*  Removes duplicates from a stream *)
PROCESS Suppress( IN in : Stream; OUT out : Stream );
VAR
  more : BOOLEAN; x,y : INT32;
BEGIN
  SELECT
    in?item(x) THEN more := TRUE
    []in?eos THEN more := FALSE
  END;
  WHILE more DO
    SELECT
      in?item(y) THEN IF x # y THEN out!item(x); x := y END
      []in?eos THEN out!item(x); more := FALSE
    END
  END;
  out!eos
END Suppress;
```

The first of the SELECT statements (Lines 01 to 04) handles the first item sent over the channel. Duplicates are detected by comparing the value read from the channel in the previous iteration to the one read in the current iteration. An eos causes the process to terminate.

3.6 Buffer

Example *Buffer*, is more complex. It implements a buffer as a sequence of *Copy* processes as defined in Example 3.3. The buffer has a maximum size of 3 elements as indicated by constant n. The predefined procedure NEW initializes a new instance of a channel type or allocates space on the heap for a pointer variable, depending on whether a port or pointer is passed to it as parameter. This example introduces the concept of an array of channels. It is important to remember that ports are references to channels, in much the same way as file handles are used to access files. When a NEW call is invoked, a channel is created.

```
(* Implements a buffer as a pipeline of Copy processes *)
CONST
  n = 3;
TYPE
  Net = ARRAY n OF Stream;

PROCESS Buffer( IN in : Stream; OUT out : Stream );
VAR
  a : Net; i : INT32;
BEGIN
  NEW( a[0] ); Copy( in,a[0] ); i := 1;
  WHILE  i <= n DO
    NEW( a[i] ); Copy( a[i-1], a[i]  ); INC( i )
  END;
  Copy( a[n-1], out )
END Buffer;
```

Upon examination of the above algorithm and the *Copy* example (Example 3.3), one finds that items propagate through the Buffer process in much the same way as values in the bubble sort algorithm. When the first item is sent to the Buffer process, this item is copied to channel a[0]. Similarly later values sent to Buffer are *hooked* by additional Copy processes. The input stream of the following process always being the output of the preceding process.

3.7 Recursive instantiation

This example demonstrates recursive instantiation of processes. It computes the final value in a Fibonacci series. The result is stored in the variable `i` as instantiated in the process `Caller`. Notice how communication replaces the use of reference parameters. The return value of each recursive instantiation is transmitted via channels. A new set of channels —represented by ports `g` and `h`— is created for each instance. These new ports are passed as parameters to each child instance. As the recursion unwinds, the values are returned via port `func`. Port `func` was instantiated as either `g` or `h` in the parent instance.

```
PROGRAM Ex012;
(*
   Calculates the n-th number in a Fibonacci series
*)
TYPE
  CfuncVal = [ f(UINT32) ];

PROCESS Fib( OUT func : CfuncVal; x : UINT32 );
VAR
  IN g,IN h : CfuncVal;
  y,z : UINT32;
BEGIN
  IF x <= 1  THEN func ! f( x )
  ELSE
    NEW( g ); Fib( g,x-1 );
    NEW( h ); Fib( h,x-2 );
    g ? f( y );
    h ? f( z );
    func ! f( y+z )
  END
END Fib;

PROCESS Caller;
VAR
  IN result : CfuncVal;
  i : UINT32;

BEGIN
  NEW( result );
  Fib( result,10 );
  result ? f( i )
```

```
END Caller;

BEGIN
  Caller
END Ex012.
```

3.8 Simple device driver

This example demonstrates some of the basic ways in which to access hardware devices using LF. It implements a simple process which displays the up-time of the computer on which it is executing. To do this it needs access to the system timer interrupt, some I/O ports and the memory mapped display. Some global declarations are shown below. A type which encompasses the logical structure of an IBM PC, memory mapped display pages, is declared. A display page on an IBM PC is logically an array of 2000 character and attribute pairs. The attributes refer to the intensity or colour of the characters depending on the display type (colour or monochrome).

```
01 TYPE
02  DisplayPage = ARRAY 2000 OF RECORD
03   char,attrib : UINT8
04  END;
```

The next fragment (Lines 5 to 21) covers the initialisation code and variable declarations of the process which does the actual work in the program. The process is called `Handler`. The only significant code in this section is the initialisation of the channel associated with interrupt 32. This happens in line 15. In line 21 the PC programmable interval timer is programmed to generate interrupts at a rate of 100Hz. The timer hardware is designed to generate an interrupt 32 upon expiry of a programmed interval. The IBM PC has a separate address space of `64kb`, which provides access to hardware devices via so-called *ports*. These ports are accessed via the `PORTIN` and `PORTOUT` predefined procedures. The first parameter specifies the port and the second parameter the value to read or write respectively. The actual values written to the ports are not pertinent to this discussion.

```
05 PROCESS Handler;
06 VAR
07   vp : DisplayPage AT $b8000; (* video page *)
08   i,pulse : UINT32;
09   h,m,s,d : UINT8;
10 BEGIN
11  i := 0;
12  WHILE i < 2000 DO
```

```
13  vp[i].char := 32; vp[i].attrib := 7; INC( i,1 )
14 END;

  (* initialize channel associated with interrupt 32 *)
15 NEW( IntChannels[32] );
16 pulse := 0; s := 0; m := 0; h := 0; d := 0;

  (* There is no string type *)
17 vp[0].char := ORD('u'); vp[1].char := ORD('p');
18 vp[2].char := ORD('-');vp[3].char := ORD('T');
19 vp[4].char := ORD('I');vp[5].char := ORD('M');
20 vp[6].char := ORD('E'); vp[7].char  := ORD('|');

  (* set Programmable interval timer to 100Hz  *)
21 PORTOUT( 67,54 ); PORTOUT( 64,156 );PORTOUT( 64,46 );
```

Lines 22 to 35 form the core of process `Handler's` functionality. It contains a non-terminating loop, which continuously waits on interrupt 32 to occur. This is done by blocking on the reception of a message over the relevant channel (Line 23). After the reception of the message the relevant variables to keep track of the elapsed time are updated (Lines 27 to 35). The `INC` predefined procedure generates more efficient code than would a simple add or subtract statement, such as `x := x+1`.

```
22 WHILE TRUE DO
    (* block on interrupt msg *)
23   IntChannels[32] ? sig;

24   INC( pulse,1 );
25   IF pulse = 100 THEN
26     pulse := 0; INC( s,1 );

27     IF s = 60 THEN
28       INC( m,1 ); s := 0;
29       IF m = 60 THEN
30         m := 0; INC( h,1 );
31         IF h = 24 THEN
32           INC( d,1 ); h := 0
33         END
34       END
35     END;
```

The final section does the menial work of outputting the uptime on the display (Lines 36-52). The main process is also listed (Lines 65-67). Since the array **vp** is declared at the address of the actual memory mapped display page (Line 7), assigning values to the fields of the appropriate record results in the display of the desired characters. For example the assignment `vp[8].char := ORD('0' )` will write the character '0' at display position 8 (coordinates (1,8)).

```
      (* display the up-time *)
36      IF s > 9 THEN
37        vp[8].char := ORD('0')+( s DIV 10 ); vp[9].char := ORD('0')+( s MOD 10 )
38      ELSE
39        vp[8].char := ORD('0')+0; vp[9].char := ORD('0')+ s
40      END;
41      vp[10].char := ORD(':');
42      IF m > 9 THEN
43        vp[11].char := ORD('0')+( m DIV 10 ); vp[12].char := ORD('0')+( m MOD 10 )
44      ELSE
45        vp[11].char := ORD('0')+0; vp[12].char := ORD('0')+ m
46      END;
47      vp[13].char := ORD(':');
48      IF h > 9 THEN
49        vp[14].char := ORD('0')+( h DIV 10 ); vp[15].char := ORD('0')+( h MOD 10 )
50      ELSE
51        vp[14].char := ORD('0')+0; vp[15].char := ORD('0')+ h
52      END;
53      vp[16].char := ORD(':');
54      IF d > 9 THEN
55        vp[17].char := ORD('0')+( d DIV 100 ); vp[18].char := ORD('0')+( d DIV 10 );
56        vp[19].char := ORD('0')+( d MOD 10 )
57      ELSE
58        vp[17].char := ORD('0')+0; vp[18].char := ORD('0')+0;
60        vp[19].char := ORD('0')+ d
61      END

62    END
63  END
64 END Handler;
(*------------------------------------------------------------*)
65 BEGIN
66  Handler
67 END Ex016.
```

3.9 A Linked List

The final example is that of a linked list. It shows how data encapsulation is achieved by using processes. The process `Q` provides the encapsulation for a linked list. Operations on the list are performed by sending messages to process `Q`, thereby providing a functionality similar to methods. Note that a new list is created for each instance of the process. This is similar to multiple instances of an object. Unfortunately no analogue for inheritance exists.

The example will again be split into sections. Lines 0 to 6 are just type definitions. The type `Node` is the data type which is manipulated.

```
00 PROGRAM List;
01 TYPE
02   Node = POINTER TO NodeDesc;
03   NodeDesc = RECORD
04     next : Node;
05     b : UINT8
06   END;

07 QDesc =  [ node(Node) ];
```

The process `Q` provides two operations on lists:

- Insertion is provided for by the first guard (Line 15).
- Removal is provided by the second guard (Line 22).

The initialisation of the list is implicit as it takes place immediately after the instantiation of an instance of process `Q` (Line 12). Notice the use of a boolean expression as part of the second guard (Line 22). This provides a convenient and efficient mechanism to test boundary conditions with. In this case whether the list is empty or not.

```
08 PROCESS Q( IN enQ : QDesc; OUT deQ : QDesc );
09 VAR
10   new,root : Node;
11 BEGIN
12   root := NIL;
13   WHILE TRUE DO
14     SELECT
15       enQ ? node( new ) THEN   (* insert operation *)
16         IF root = NIL THEN
17           root := new
```

```
     ELSE
      new.next := root;
      root := new
     END

   [] deQ ! node( root ) & root # NIL THEN (* remove operation *)
     root  := root^.next
   END
 END
END Q;

PROCESS Control;
VAR
 i : UINT8;
 n : Node;
 OUT enQ : QDesc;
 IN deQ : QDesc;

BEGIN
 i := 0; NEW( enQ ); NEW( deQ ); Q( enQ,deQ );
 WHILE i < 20 DO
  NEW( n );  n^.b := i; n^.next := NIL;
  enQ ! node( n );3 INC( i,1 )
 END;

 deQ ? node(n);
 WHILE n # NIL DO deQ ? node(n) END
END Control;

BEGIN
 Control
END List.
```

3.10 Remarks

It is anticipated that the clear and concise syntax of **LF** would appeal to those programmers familiar to **Pascal** and derived languages, such as **Oberon**. I hope that after examining the examples presented in this chapter the reader will have an intuitive understanding of the nature of **LF**. Syntactically and semantically **LF** is essentially an extension of **Joyce**. The extensions are for the most part designed to access hardware devices. These extensions include the ability to associate interrupts with channels, and the declaration of variables at absolute addresses.

occam has a more complex syntax than does **LF/Joyce**. Many of its constructs are however found in **LF** and **Joyce**. The `SELECT` construct for instance has an analogue in **occam** —the `ALT` construct. One of the most significant differences between **occam** and **LF/Joyce** is the fact that **occam** supports parallelism at statement level. In **LF/Joyce** parallelism is at process level. Moreover **LF/Joyce** does not offer the ability to sequentially execute processes. The channel typing mechanism of **LF** and **Joyce** is the same, but is much more rigorous than that of **occam**. **LF** and **Joyce** support recursive process instantiation, **occam** does not. **LF** does not implement device drivers as part of its runtime system. Given the intended use of **LF** (embedded work) this is to be expected. **Joyce** and **occam** integrate drivers into their runtime systems.

Chapter 4

Runtime System

The most challenging part of this project was the development of the runtime system. The need to implement a fast, efficient runtime system was offset by the necessity that the system be small and relatively maintainable so as to make it easier to modify when testing new ideas or conducting performance measurements. These and other problems will be discussed in this chapter. The chapter begins with an overview of the design criteria for the system. This is followed by a brief discussion of the structure of the runtime system and a design overview. Memory management is examined next. In the section on process management, the mechanism of context switches without hardware mechanisms is described. A discussion of scheduling is followed by a discussion of communication. Amongst others, two different implementations of the `SELECT` construct are discussed. This chapter concludes with a platform specific analysis of the overhead associated with runtime checks, as well as the performance of the interrupt subsystem.

4.1 Design Criteria

A number of factors were taken into account during the design of the system:

1. Execution speed
2. Memory usage
3. Security
4. Maintainability and portability

Fast execution is preferable in any software system. When implementing a useful runtime system it becomes essential. The efficiency of the runtime system greatly influences the efficiency of any program written in the language which it supports. It is also required that the runtime system be small in size and use little **memory**. A small runtime system improves execution speed by improving the cache hit ratio. More importantly, the less memory the runtime system uses, the more memory is available to the programmer. This is of special importance in embedded applications. When implementing a secure language, a **secure** runtime system is indispensable.

Since this is an experimental implementation —which was likely to change significantly during the evolution of the project— it was considered prudent to make the code as maintainable as possible. Most of the runtime system is therefore implemented in a high level language. Only speed critical portions of the runtime system are implemented in assembly language. Portability arose as a beneficial side effect. The runtime system as a whole is of course not portable, but significant portions (including a complete module) are.

The language **Oberon** [36] was selected for the implementation for the following reasons:

- It is a secure language. This makes it easier to implement a secure runtime system.
- A large base of software in **Oberon**, due to previous operating system projects, was available. Many portions of code in the runtime system have their origin in this code.
- **Oberon** is well known to the author.
- **Oberon** compilers are available for many popular architectures. This is not of much value in this experimental version of the system, but will of course make porting the system to other architectures much easier.

4.2 The modular structure of the runtime system

The runtime system is divided into three modules: *LFRuntime*, *LFHeap* and *LFProcess*. *LFRuntime* forms the basis for the rest of the runtime system. This module initialises the processor and programmable interrupt controllers. It also contains the primitives to facilitate interrupt and exception handling as well as the primitives for producing debugging information. This module is very hardware dependent. However, it successfully abstracts the intricacies of the hardware from the rest of the runtime system —promoting maintainability and portability in the rest of the runtime system.

LFHeap, the heap manager, is entirely portable, as it contains only **Oberon** code and uses no architecture specific constructs or features. The heap manager serves a dual purpose. It supports the dynamic memory allocation primitives defined within **LF** and also supports the dynamic memory primitives as used by **Oberon** in the runtime system. This pooling of resources, would be dangerous were it not for the fact that both **Oberon** and **LF** are secure languages.

LFProcess is at the highest level of the runtime system. It implements the scheduler and the communication primitives. All of the code is portable except for the speed critical code that handles context switches and the copying of messages. A version of *LFProcess* that implements the communication procedures in assembler and uses registers for parameter passing, was implemented to measure performance.

4.3 Design overview

The system is tailored towards embedded systems. As such the runtime system must run on the bare hardware and implement some of the functionality that one would find in an operating system. Scheduling and interprocess communication (*IPC*) are the most noteworthy of these functions. The entire **LF** execution environment (runtime system, program code, activation records and heap) resides in a single linear address space. The use of paging was considered to be inefficient and unnecessary. The **LF** language itself guarantees the memory security of each process. Page tables take up memory which could be put to better use in embedded systems. Moreover, some popular embedded processors, such as some **ARM** processors, do not support hardware memory protection, or only implement primitive protection [28]. It is also unlikely that a need for virtual memory would arise in an embedded application.

The target platform for the **LF** system is the *i386EX* embedded processor, which is essentially a specialised version of the *i386SX* core. Instruction privilege levels are not used. The entire **LF** system executes at the highest privilege level. **LF** does not allow the programmer to execute privileged instructions. Most of these instructions are intended for memory and privilege level management in any case. The notable exception is access to I/O ports. The need for low level access to devices makes access control to ports impractical. The Intel *i386* allows the programmer to define a bitmap for each process which allows hardware access control to I/O ports. This is not practical as the use of this bitmap makes the execution of hardware I/O instructions in the order of **four** times slower. Moreover it forces the programmer to use the *i386* task structure [20]. The use of tasks would have complicated the runtime system significantly and is less efficient than the current implementation. Such hardware control of I/O access is also not available on many embedded controllers, and would make the system less portable.

Procedure calls, rather than traps are used to access the runtime system. This is highly efficient, as context switches are avoided in many cases.

4.4 Memory allocation

The memory map of the **LF** system is divided into three parts (as illustrated in Figure 19). The first part (at the lowest offset in memory) contains the static image of the runtime system and program code. This is loaded by the boot loader as discussed in Appendix B. The remaining physical memory is divided into a heap area and stack area. The heap is allowed to grow from the static image towards the top of memory. The stack grows from the top of memory towards the heap. The stack contains the process and channel *activation records (ARs)* as well as the value

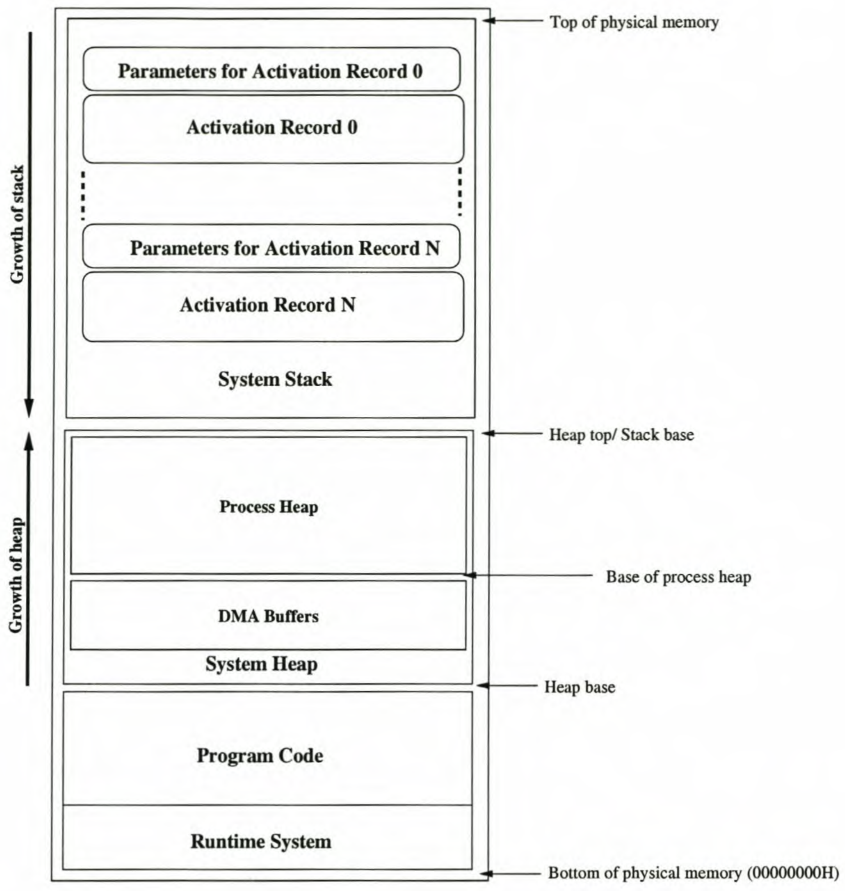

Figure 19: System memory layout

parameters passed to the processes upon instantiation. Processes are activated in a hierarchical fashion. Each process that instantiates another process becomes the parent of the new process. This logical tree structure is represented as a stack of activation records, one for each process. The value parameters passed to the process upon instantiation immediately precedes its activation record. The details of process management are explained in the next section. Section 4.9 covers the details of channels and communication.

The heap manager is implemented as part of the runtime system. The manager is invoked by the use of the seven pseudo procedures defined in Section 2.5. The heap is divided into two parts: a lower and an upper half. The lower half is reserved for DMA buffers to accommodate the *ISA* bus standard, which requires DMA buffers to be below the `16MB` memory offset. The upper half is used for dynamic allocation of variables. When the need arises some of the lower memory reserved for DMA may be released for use as part of the upper heap. This is done with the `RELEASEDMA` system call (Section 2.5).

A **first fit** strategy is employed because it is simple and reasonably efficient. A free block descriptor is located at the start of each free block of memory on the heap. The well known technique of chaining block descriptors in a FIFO list is employed [2].

The use of a chain of free blocks provides for a convenient and elegant way of overcoming the unfortunate physical memory layout of the IBM compatible series of PC's (on which the prototype is implemented). These machines map hardware devices into memory between the offsets of `640 KB` and `1MB`. The various display adapters are a good example of this. All fragments of free memory between `640 KB` and `1MB` and indeed any memory not occupied by the static image below `640 KB` are inserted into this list. Blocks of physical memory below the bottom of the start of the heap may therefore be logically handled as though they were part of the continuous heap portion of the memory.

It is important to note that the use of the `AT` construct (for defining variables at specific addresses) may lead to interference between the runtime system and the **LF** processes. This may occur when defining a variable within a portion of memory which forms part of the heap or runtime system itself. This problem is avoided by only allowing `AT` declarations within the range of addresses that is used by memory mapped devices. (`640KB` to `1MB` for the PC). This is justified by the fact that the `AT` construct is intended only to access memory mapped devices, although it introduces some hardware dependence into **LF**.

4.5 Process Management

Processes divide the memory into separate, secure, address spaces and form the basis for implementing concurrency. In addition, processes form the basis for abstraction in **LF** as was illustrated in Chapter 3.

4.5.1 Process creation

An instance of a process is represented by its *Activation Record (AR)*. The structure of an *AR* is shown in Figure 20, a '↑' indicating a pointer.

The **`mstate`** part of the *AR* is used to save the context of the process, whenever a context switch is performed. The **`size`** field is the size in bytes of the *AR*. The **`BufAdr`**, **`NextQ`** and **`expr`** fields are used during communication and will be discussed in Section 4.9. A process may be in one of four states as indicated by the **`status`** field:

1. `READY` - It is ready for execution.
2. `DONE` - The process has terminated.
3. `BLOCKED_ON_READ` - The process is waiting to receive a message.
4. `BLOCKED_ON_WRITE` - The process is waiting to send a message.

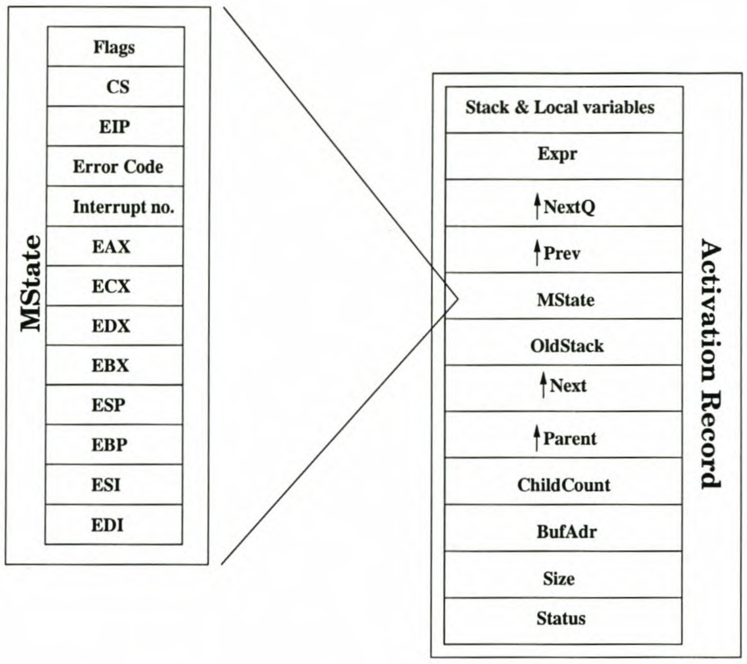

Figure 20: Internal representation of processes

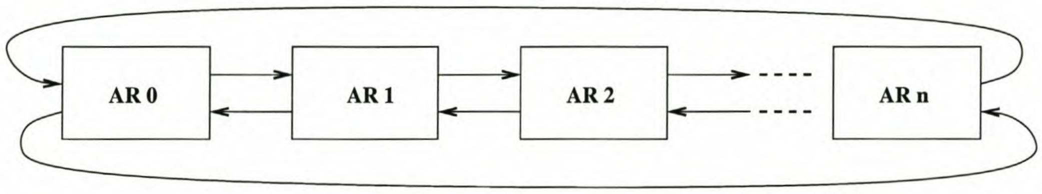

Figure 21: Ring of process ARs

For the purposes of scheduling, *ARs* are placed in a doubly linked ring (Figure 21). The `next(prev)` field points to the next(previous) *AR* within the linked list of *ARs*. A double link is used to allow efficient removal of processes from the ring. The `parent` field points to the *AR* of the parent of the current *AR*. Recall that processes are activated in a parent/child hierarchy. Note that an execution stack as well as space for all the local variables are allocated within the *AR* of a process instance. As **LF** does not implement procedures, a stack is not necessary. This issue is addressed in Section 4.14.

New instances of processes are created in the following manner:

1. The (execution) stack is switched from the process wishing to create the new instance, to the *AR* stack.

2. The parameters for the process are pushed onto the stack.

3. The address of the process to be instantiated is pushed.

4. Space is allocated for the *AR* on the *AR* stack by decrementing the stack pointer. The `oldStack` value of the *AR* is set to the old value of the stack pointer; that is the value before the execution of step 2. This allows for an efficient way of deallocating the memory reserved for the new *AR*.

5. The appropriate values, such as the instruction pointer (`EIP`) are set in the `mstate` part of the *AR*.

6. The *AR* is inserted into the ring of *ARs* used for scheduling.

7. The `ChildCount` field of the parent process is incremented.

8. The stack is switched back to the process which requested the instantiation. This process remains the active process.

The new process instantiation will only begin executing once it is scheduled.

4.5.2 Context switches

Context switches are performed in software. Hardware mechanisms in terms of '`task`-structures' are available on the *i386*, but are significantly slower than the software implementation outlined here. The *i386* task structure is also about 20% larger than the *AR* structure used in **LF** [20].

In order to understand the operation of these procedures some background is required as to how interrupts are handled by the runtime system. Interrupts are handled in two phases. A generic mechanism traps hardware and software interrupts as well as machine exceptions. This mechanism allows an interrupt handler written in **Oberon** to be associated with each of the possible interrupts or machine exceptions. The low-level handler passes control to this high level-handler (an **Oberon** procedure) after doing some initial processing to interface with the high level code.

Whenever an interrupt is invoked either by the hardware or software, the processor pushes three values on the execution stack: the `flags`, `CS`(code segment) register and the instruction pointer (`EIP` register). In the event of certain machine exceptions the processor will also push an error code onto the stack. The low-level interrupt handler now pushes the interrupt number and in the case where an error code was not supplied pushes a zero on the stack to maintain alignment. Next all the general purpose registers are pushed onto the stack. Finally, a `CALL` instruction is issued to whichever high-level interrupt-handler was installed by the higher layers of the runtime system. This extra instruction adds about `4.8` percent of overhead to the servicing of interrupts, when assuming an empty handler. This overhead is not much when one considers the benefits of the ability to interface with high level code.

```
56 EFLAGS      Pushed by INT XX
52 CS          Pushed by INT XX
48 EIP         Pushed by INT XX

44 Error code  Pushed by low level handler
40 Int no      Pushed by low level handler

36 EAX         General purpose registers pushed by low level handler
32 ECX
28 EDX
24 EBX
20 ESP
16 EBP
12 ESI
08 EDI

04 EIP         return address pushed by call to installed high level handler
00 EBP         pushed by entry code to procedure
-x             local variables of Oberon procedure
```

Figure 22: State of stack during context switch

```
PROCEDURE Handler( m : MState );
BEGIN
  ⋮
  m := some_other_mstate
  Continue( m.esp-32 )
END Handler;
```

Figure 23: The skeleton of an interrupt handler.

The contents of the stack upon entry to the high level interrupt handler is listed in Figure 22. The offsets are numbered relative to the EBP register which indicates the base of the procedure activation record.

Notice the equivalence of the structure in Figure 22 and the mstate structure in Figure 20. This allows the interrupt handler to access the context information of the interrupted process as a parameter. The skeleton of an interrupt handler is presented in Figure 23. The parameter m is used to access the context information of the process. Interrupt handlers are not allowed to return with a normal procedure return instruction (as generated by the **Oberon** compiler). Instead a special procedure, Continue, must be called, which uses the IRETD instruction. The EFLAGS, CS and EIP values are popped from the stack and the processor then jumps to the address specified by EIP. A context switch to process P is achieved by calling Continue with the stack pointer value associated with P as parameter. This stack pointer value is contained in the mstate that was saved when the process was interrupted. Note that the '-' before the procedure name in Figure 24 indicates to the **Oberon** compiler not to generate a procedure activation record, but

```
PROCEDURE -Continue( stackPointer : LONGINT );
  POP ESP          ; set ESP register to parameter
  POPAD            ; restore general registers
  ADD ESP,8        ; pops the error code and intno values
  IRETD
END Continue;
```

Figure 24: Continue

rather to insert the code of the procedure body at the point where the procedure would have been called. All hardware and software interrupts are handled by the scheduler. Machine exceptions are handled by a special exception handler procedure that produces debugging output and then halts execution.

4.5.3 Process termination

A process may terminate only when all its child processes have terminated. The last sequence of machine code for each process invokes the `ProcDone` system call. This sets the state of the process to `DONE` and decrements the `ChildCount` field of the parent *AR*. When the process is again scheduled its `ChildCount` field is compared to `0`. An activation record with no children and a state of `DONE` is removed from the scheduling ring, and its process record destroyed, thereby releasing the memory associated with it. The release of memory is accomplished by simply setting the `freemem` value to the `oldstack` value contained in the *AR* of the process.

The parent/child structure of the process activation tree guarantees that the above protocol will not destroy a channel that is in use. Processes can only share channels when a port variable relating to the channel is passed as a parameter. Since only value parameters are allowed, the parent process is forced to initialise the port and instantiate the child with this port as parameter. The parent is therefore always the owner of the port and as noted above cannot terminate before its children.

4.6 Interrupt handling and Devices

Since **LF** is intended for embedded systems, efficient access to hardware devices is essential. Most peripheral devices are interrupt driven. Tanenbaum [31] noted that:

> 'Interrupts are an unpleasant fact of life. They should be hidden away in the bowels of the system, so that as little as possible of the system knows about them.'

This advice was followed in the design of the **LF** interrupt mechanism. An interrupt handler is a process like any other. The runtime system (more specifically the scheduler) converts interrupts to messages that may be received by the appropriate interrupt handler (Figure 25). This makes the system as consistent as possible and also ensures a high degree of abstraction. An array of pointers to channels is kept, one channel per interrupt. This array is made visible to the user as a predefined array of ports called `IntChannels` (Section 2.5). This interface is platform independent, except for the size of the array.

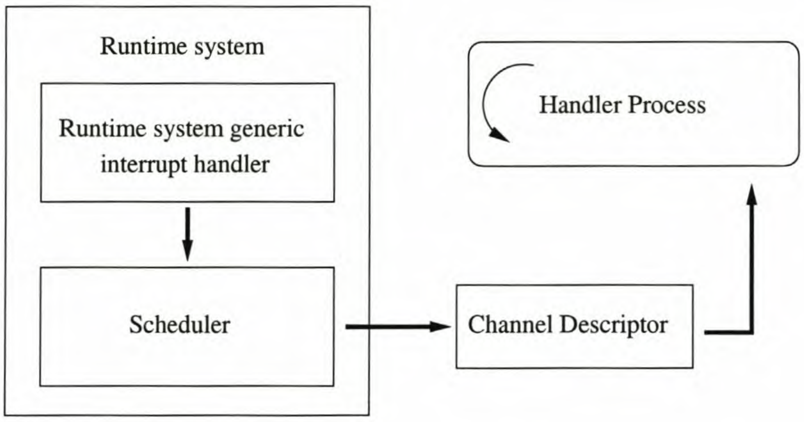

Figure 25: The interrupt mechanism

Note that the actual channel descriptor is only created once a `NEW` call is issued so that the memory overhead of the above is minimal. Only `MaxInt` pointers are needed as opposed to `MaxInt` instances of channel descriptors. To ensure acceptable performance, the scheduler immediately passes control to the appropriate interrupt handling process, provided that it is blocking on a *Hook* statement. If no process is ready to receive the interrupt, then the next `READY` process is selected. Scheduling will be discussed in the next section.

Up to one interrupt is buffered (by the runtime system) per channel. This helps to alleviate the loss of interrupts during periods of heavy processing loads, and is essential in the implementation of the `SELECT` construct, as will be explained in Section 4.9.2. It was decided to limit the amount of buffering because unbounded buffering could cause the system to run out of memory. Additionally, any appreciable amount of buffering causes a latency between the occurrence and the servicing of an interrupt.

4.7 The scheduler

Scheduling takes places on a round robin basis. This is a simple strategy with minimal overhead. With each invocation of the scheduler the next process in the `READY` state, i.e. not waiting for communication, is scheduled. Interrupts are an exception to this rule. Whenever an interrupt occurs, the scheduler will activate the associated handler if possible. The only time when this will not be possible is when the system is busy servicing another interrupt and therefore not currently blocking on the channel associated with the interrupt. Time slicing is not employed.

The scheduler is invoked upon each communication statement, or when a hardware interrupt occurs. This means that after each *Bang/Hook*, a context switch occurs. A `SELECT` statement is treated as an indivisible unit. When all guards have been evaluated without success a context switch occurs. Alternately a context switch occurs after the successful evaluation of a guard and the completion of its command sequence.

```
00 PROCEDURE ReqHandler( s : MState );
(* context is passed as parameter 's' *)
01 BEGIN
02  currentProcess.mstate := s;

03  IF s.intno # 42 THEN (* field IRQs & other interrupts *)
04   <ACKNOWLEDGE INTERRUPT CONTROLLER HARDWARE >

05   IF (IntChannels[i] # NIL ) & (IntChannels[i].symbols[0].readerFirst # NIL) THEN
06    currentProcess := IntChannels[i].symbols[0].readerFirst;
07    <REMOVE READER FROM QUEUE>
08    currentProcess.status := ready
09   ELSIF IntChannels[i] # NIL & (IntChannels[i].symbols[0].writerFirst = NIL) THEN
10    <ENQUEUE A DUMMY PROCESS INTO THE QUEUE ASSOCIATED WITH THE CHANNEL>
11   END;
12
13   Continue( currentProcess.mstate.esp-32 )
14  END;

15  IF currentProcess.status = done THEN RemoveProcess( currentProcess ) END;
16  REPEAT
17   currentProcess := currentProcess.next
18  UNTIL currentProcess.status = 0;

19  Continue( currentProcess.mstate.esp-32 )
20 END ReqHandler;
```

Figure 26: The scheduler

When no processes are in the `READY` state, the system is not necessarily in deadlock, as a process may be waiting on an interrupt. The runtime system therefore creates an additional dummy process upon startup of the system. The scheduling of a dummy process was the easiest way of circumventing this problem.

The current scheduler is very simple, but may be easily extended to include facilities such as priorities and time-slicing. An outline of the scheduler is shown in Figure 26. Phrases enclosed within a `'<'` `'>'` pair are descriptions of pieces of code that were omitted in order to simplify this discussion.

In line 2 the state of the currently executing process is saved. The variable `currentProcess` is a pointer to the AR of this process. In line 3 the number of the interrupt which invoked the scheduler is examined to determine whether it was triggered by communication (‘`s.intno = 42`’) or not. If communication primitives did not trigger the scheduler, lines 4 to 14 are executed: In line 5 a test is performed to determine whether a process is blocking on the relevant interrupt. If one is, it is directly scheduled (lines 6 to 8). If not then a dummy process is enqueued in the channel to represent the interrupt. Control is then passed to the interrupted process in line 13. The constant `32` in lines `13` and `19` is used to adjust the stack pointer for the stack being switched to, for the correct execution of the `POPAD` instruction in `Continue`.

The other scenario occurs when the scheduler was invoked due to a communication primitive. In this case lines 15 to 20 are relevant. In line 15 a test is performed to see whether the process should be destroyed or not. In lines 16 to 18 the next process eligible for execution is selected. Control is transferred to this process in line 19.

4.8 Fairness

Fairness (in the context of scheduling) refers to the guarantee that a process P which is eligible for scheduling, will indeed eventually be scheduled regardless of the behaviour of the other processes running concurrently with P [3]. Two forms of fairness are: *Weak* fairness and *Strong* fairness. *A scheduling policy is* ***weakly fair*** *if it will eventually schedule a process which remains eligible for execution indefinitely. Remaining eligible for execution indefinitely means that once it becomes eligible for execution, the process never again enters a state where it is no longer eligible for execution. A scheduling policy is* ***strongly fair*** *if it will eventually schedule a process which is eligible for execution infinitely often.*

LF provides no guarantees as to the fair scheduling of processes. A FIFO based policy is adhered to. The FIFO (FCFS[1]) scheduling policy is weakly fair by construction. The non-determinism introduced by interrupts may lead to the degradation of weak fairness. Moreover in the unlikely event of the system being swamped by interrupts, only processes waiting on interrupts will be scheduled. The other processes in the system are subject to *starvation* in this case. Starvation can also be introduced by the interaction of processes not related to interrupts. If communication between two specific processes remains feasible most of the time, other processes may be starved as well. An additional complication arises when a process is locked in a non-terminating loop which does no communication. When no interrupt handlers are defined, such a system will not proceed in its execution.

Note that weak fairness does not guarantee ‘quality of service’. Processes, such as device drivers that rely on hardware polling, would generally not run effectively as they are unlikely to be scheduled often enough. Moreover, they might be scheduled erratically. Also, in the current implementation, `SELECT` statements have an implicit priority order built into them. Guards are

[1]First come first served

evaluated from the top. It is not known how to improve fairness in this case without sacrificing efficiency.

4.9 Communication

Processes in **LF** communicate by using synchronous message passing. Since this is the only means of inter-process communication, an efficient implementation is essential. Communication is achieved by manipulating channel and process activation records. Channels are implemented by associating two FIFO queues with each symbol within the channel's alphabet, one queue for writers to, and one queue for readers from the channel. Note that the semantics of **LF** makes it impossible for both queues to be in use at the same time. The `NextQ` field of the AR is used to link processes inserted into the FIFO queue together. Operations on FIFO queues are more expensive (in terms of memory requirements and execution time) than those for LIFO queues, but a FIFO implementation does promote a measure of fairness. The use of two queues might seem wasteful, but it saves the effort and runtime overhead of explicitly tracking which queue contains readers and which contains writers. Only one word (32-bits) of extra memory is used [2].

4.9.1 Simple Communication

Simple communication is implemented by the `BangIt` and `HookIt` system calls. `Bangit` is executed by a process wishing to send a message. If no process is ready to receive the message, the sending process is blocked and its AR is inserted into the queue corresponding to the symbol which it wants to send. If however, a process is ready to receive the data, i.e. it is contained in the appropriate queue, the data is copied directly from the sender to the receiver. The sender is unblocked and removed from the queue corresponding to the symbol sent.

The `bufAdr` field in the AR of a process is used to store the source or destination addresses of data that needs to be sent over the channel, depending on whether a `BangIt` or `HookIt` was executed. The buffer address of the process executing the `BangIt/HookIt` is passed as a parameter to these procedures. When a process is blocked this value is stored in `bufAdr`. Therefore upon feasible communication the buffer address of one process is stored in its activation record (the blocked process) and the buffer address of the complementary process is available as a parameter. Figure 27 shows a channel which contains two symbols in its alphabet (`Symbol_0` and `Symbol_1`). `Symbol_1` is buffering three processes that wish to read this symbol from the channel. `Symbol_0` is buffering two processes that wish to send this symbol over the channel. `HookIt` essentially does the opposite of `BangIt`. A simplified version of `HookIt` is listed in Figure 28. In line 04 the queue corresponding to the appropriate symbol is examined to determine whether it contains a blocked *writer*. The parameter `symbol` serves to identify the correct symbol from the alphabet. If a *writer*

[2]The Oberon compiler pads its record fields to 32-bit boundaries, so a boolean field would also use 32-bits of memory

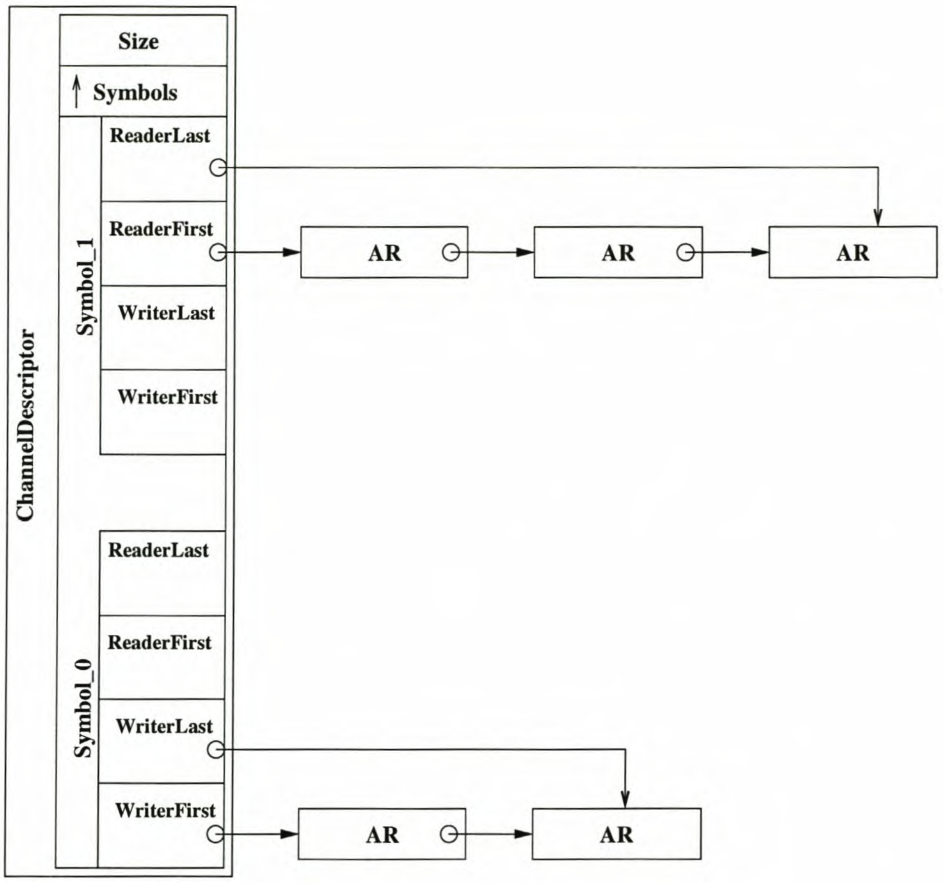

Figure 27: Communications channels

is blocked, then the data is copied (line 05), the *writer* is unblocked (line 06) and removed from the queue (line 07). If a *writer* is available, then the process executing the *HookIt* —denoted by `currentProcess`— is inserted into the appropriate queue (lines 09 to 17).

An interesting aspect of the queue management associated with the channels, is that it takes approximately 5 times as long to insert a process into a queue (when it blocks) and again remove it when it is allowed to continue, than it does to compare whether a process is ready for execution i.e. compare the `status` field of the process to `READY`. This implies that for a small amount of processes it is more efficient **not** to remove them from the scheduling queue, when they are inserted into the channel queues. 'A small number of processes' is of course not a precise definition. A more quantitative argument entails the following:

As a first approximation assume that no hardware interrupts are enabled. This means that the scheduler is only invoked when the currently executing process is blocked due to the infeasibility of communication. In order to avoid deadlock at least one other process must be in the `READY` state. In order for 5 successive unsuccessful comparisons to occur at least 7 processes must exist

```
PROCEDURE HookIt(  ch : Channel; bufAdr,sz,offset : LONGINT );
BEGIN
 ASSERT( ch # NIL );
 IF ch.symbols[offset].writerFirst # NIL  THEN
  MoveN( ch.symbols[offset].writerFirst.bufAdr,bufAdr,sz );
  ch.symbols[offset].writerFirst.status := ready;
  ch.symbols[offset].writerFirst := ch.symbols[offset].writerFirst.nextQ
 ELSE
  currentProcess.bufAdr := bufAdr;
  currentProcess.nextQ := NIL;
  currentProcess.status := blockedOnRead;
  IF ch.symbols[offset].readerFirst = NIL THEN
   ch.symbols[offset].readerFirst := currentProcess
  ELSE
   ch.symbols[offset].readerLast.nextQ := currentProcess
  END;
  ch.symbols[offset].readerLast  := currentProcess
 END;
END HookIt;
```

Figure 28: HookIt

in the system. If we assume that at least a third of processes are ready for execution then, in the worst case, 21 processes are needed to encounter 5 successive blocked processes. The likelihood that 5 successive processes will all be blocked is also low. This likelihood clearly increases with the number of processes, but it is unclear how this 'cross over point' can be determined without an extensive analysis of the program that is to be implemented. When one takes the occurrence of hardware interrupts into account the only significant change to the model is the introduction of some of randomness in the order of scheduling. The relative density of `READY` processes is not expected to change. The choice of queue management is left to the implementer as it is not part of the language definition. I chose the simpler and —for systems with relatively few processes— more efficient implementation, and therefore do not remove blocked processes from the scheduling queue.

4.9.2 Implementing the SELECT

The `SELECT` construct is implemented using the `ReadPoll`/`WritePoll` and `CommitReadPoll`/`CommitWritePoll` system calls. `SELECT` statements are not allowed to synchronise. The rationale behind this decision was explained in Section 2.1.2. This constraint facilitates a relatively simple implementation.

Implementation One

A process wishing to perform a `SELECT` calls `ReadPoll` or `WritePoll` for each guard, one after the other, depending on whether a *Hook* or *Bang* is encountered. These procedures interrogate the appropriate queue in the channel corresponding to the referenced symbol. If communication is feasible, i.e. another process is blocked in the relevant queue, the message is copied and the value `TRUE` is returned. If communication is not feasible then `FALSE` is returned. The optional boolean expression of the guard is then evaluated, using short circuit evaluation. If the guard evaluates to `TRUE` the appropriate `Commit` call is issued. `CommitWritePoll/CommitReadPoll` does the necessary housekeeping to unblock the communications partner. If the guard evaluates to `FALSE` the states of the queues and processes are not changed, and the following guard is processed. If all the guards in a particular `SELECT` fails the scheduler is invoked as the current process is blocked.

Implementation Two

The implementation outlined above is simple, but has some deficiencies. The most obvious inefficiency is the need to copy messages for every evaluation of every guard. It makes the implementation much easier as the values needed to evaluate the boolean expression in the guard are available as a local variable to the receiving process. For small messages (a few bytes) the overhead is relatively small. For larger messages the overhead becomes prohibitive. Clearly a more efficient mechanism is needed.

The boolean expression needs to be evaluated before any copying is done. This is achieved by modifying `ReadPoll` and `WritePoll` to return a reference to the AR at the head of the appropriate queue —a `NIL` pointer indicates an empty queue and therefore infeasible communication. In the case of a `ReadPoll` the returned pointer is used to access the variables in the address space of the sending process. For `WritePoll` the needed data is of course already in the local address space. The actual copying of data is performed by the appropriate `Commit` primitive.

Another —less obvious— problem with *Implementation One* occurs when hardware interrupts are encountered. A normal *Hook* statement operating on a channel coupled to an interrupt works fine as it will block while waiting for the interrupt or alternatively eventually service a buffered interrupt. This was explained in Section 4.6. However, a `SELECT` containing a guard with similar functionality will not work as efficiently, and significantly increases the likelihood of missing interrupts. The problem arises because `SELECT` constructs do not block on a specific guard, and insert the process record in the appropriate queue. The only way a `SELECT` can accept an interrupt is if it were previously buffered by the runtime system. This introduces a time delay between the occurrence of the interrupt and the actual servicing by the appropriate process. The unpredictable scheduling behaviour of the system compounds the problem.

This appears to be an inherent problem with the `SELECT` construct. One possibility would be to keep track of each process that services interrupts, as well as the guards in these processes that refer to the channel associated with the interrupt. The scheduler would then be able to directly schedule the pertinent process. Only one process can service a specific interrupt. The problem with this mechanism, however, is to determine whether the process is indeed busy processing the `SELECT` at the time the interrupt occurs. Clearly control can only be passed to the process if this is the case. An obvious solution to the problem would be to add an additional status value to the activation records. `SELECT_INT` could indicate that a process is currently processing a `SELECT` containing a reference to a channel associated with an interrupt. When an interrupt occurs the runtime system could then pass control directly to where the relevant guard is evaluated. A significant constraint then needs to be imposed on the use of `SELECTS` and interrupts. Only one guard may be associated with any specific interrupt within a given process, otherwise it is unclear to where control should be passed. The proposed solution should solve the problem, but has not been implemented. In **occam** a similar constraint to the one above is imposed on the `ALT` construct —the equivalent of the `SELECT`. The same channel may not be referenced by multiple guards in the same `ALT` construct.

4.10 Communications performance

The speed of communication is an essential criterion for the evaluation of the **LF** system, as it has a profound impact on the performance of the system as a whole. An experiment was conducted in which **Oberon** code executing `10000` procedure calls with parameters of varying size was compared to the communication between two **LF** processes using messages of corresponding size. In this experiment simple *bangs* and *hooks* were used. Message/parameter sizes varying between `1` and `512` bytes were used. Two sets of measurements were taken. In one set the parameters for the communication primitives were pushed onto the execution stack, and in the other set, these parameters were passed in registers.

Rather than concentrating on absolute values, this section tries to compare the use of IPC as paradigm to that of procedure calls. Please note that the values presented in this section should not be taken as precise measurements. Values are only accurate to the nearest millisecond, as this is the granularity of the timer used.

Three graphs are presented:

- Figure 29 shows the effect of using registers as opposed to the stack to pass parameters to the communication procedures.
- In Figure 30 the efficiency of IPC is compared to procedure calls.
- In Figure 31 the ratio of IPC execution times to procedure call execution times for both implementations is presented.

The graphs shows nothing unexpected. The values for both implementations converge for large buffers (Figure 29 and Figure 31), because the relative advantage of passing values in registers

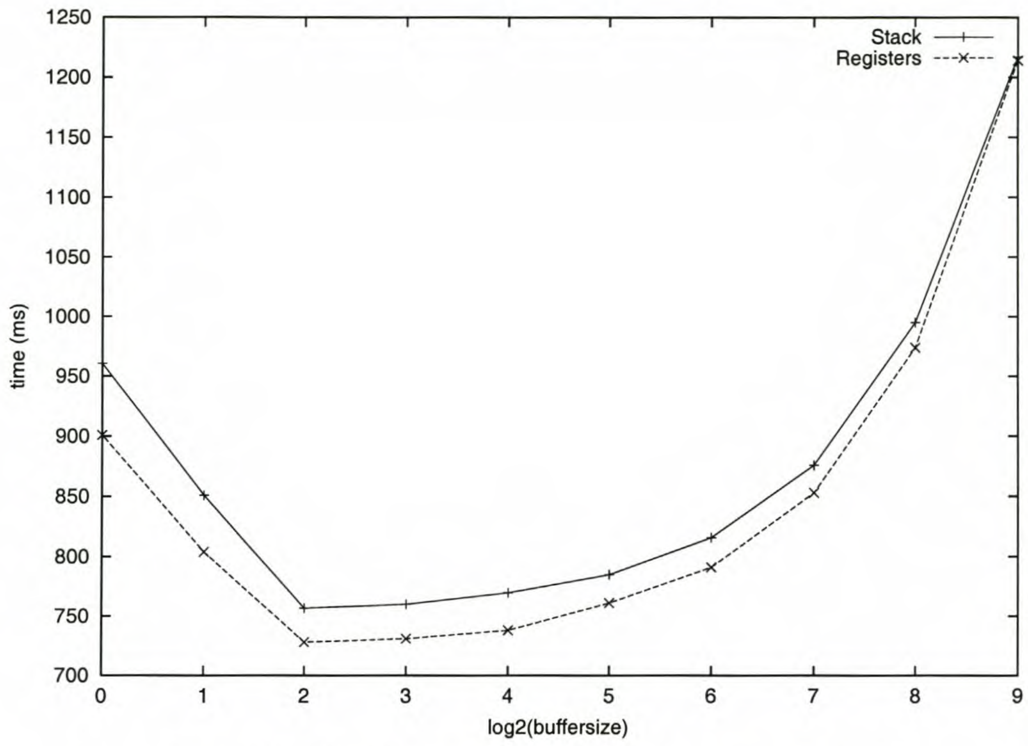

Figure 29: IPC with and without registers to pass parameters

becomes negligible relative to the time consumed to actually copy a large message. Similarly, IPC performance compares more favourably to procedure call times for large buffers, as again the relative overhead of IPC become less significant when compared to the time to copy the data. From Figure 30 it is also clear that a significant relative penalty is incurred for very small messages —1 or 2 bytes. The reason is that significantly more time is spent setting up the copying of data than is spent on copying. A contributing factor to the poor performance is the fact that the code which performs the copying of data during IPC is less efficient for such small messages. In Section 4.12.7 issues relating to message sizes are discussed. The most important observation that may be made from Figure 30 is not that that IPC is significantly slower than procedure call —this is inevitable—, but that difference in speed is acceptable. As far as simple I/O is concerned, it was found that with buffer sizes of 16 and 32 bytes, IPC is only about 9 time slower than procedure calls. The ratio improves for larger values. One has to bear in mind that communication is an inherently expensive operation, which not only passes data between processes, but also synchronises them. To achieve this while still not being an order of magnitude slower than a basic conventional programming construct is quite difficult.

The above experiment was repeated using a SELECT, with a single guard. Multiple guards will be dealt with shortly. The results are listed in Figure 32. Considering the flexibility and complexity of the SELECT construct, it performs surprisingly well when compared to simple communication. In Figure 33, the relative overhead of a SELECT containing a single guard to a simple *Hook* is

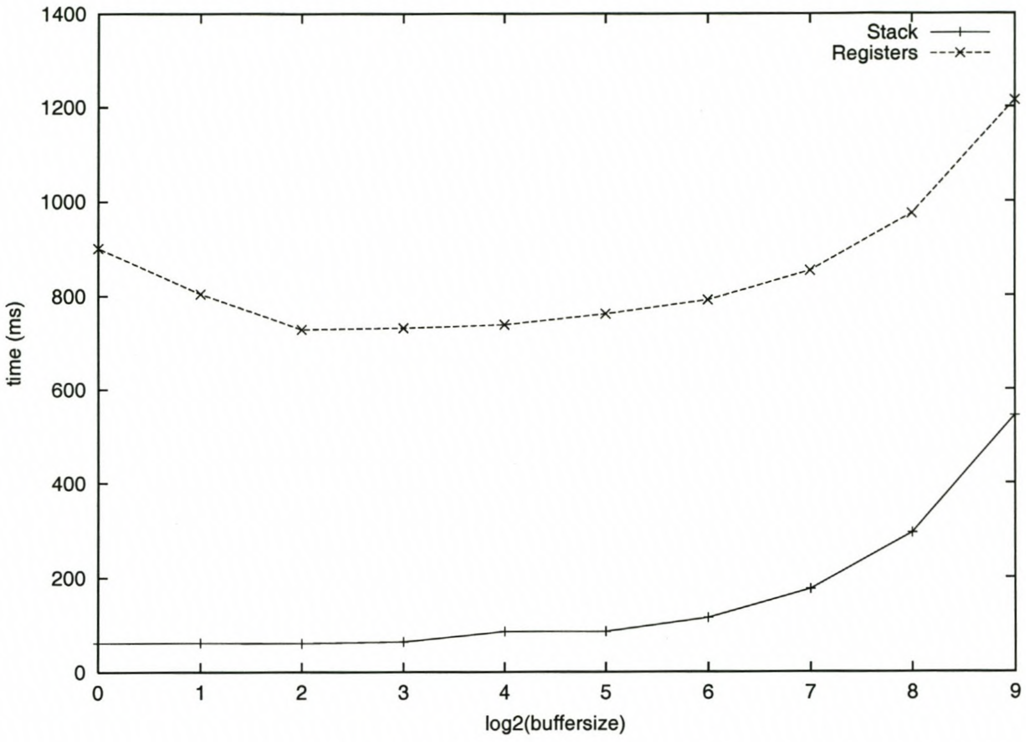

Figure 30: IPC vs. Procedure Call

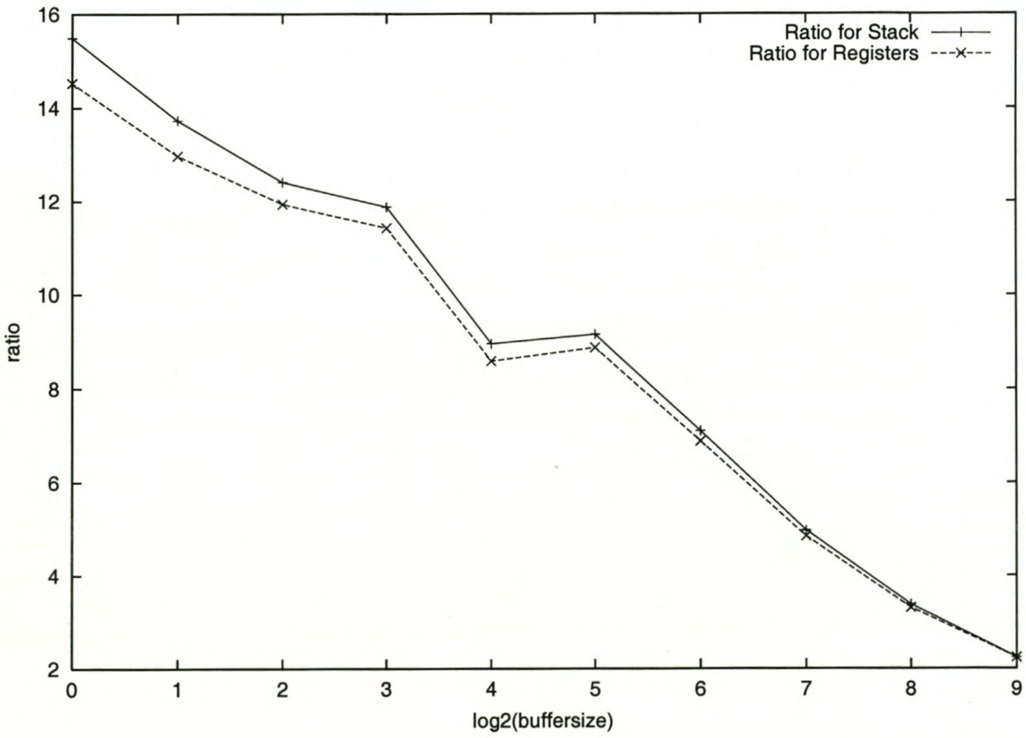

Figure 31: IPC to Procedure call ratio

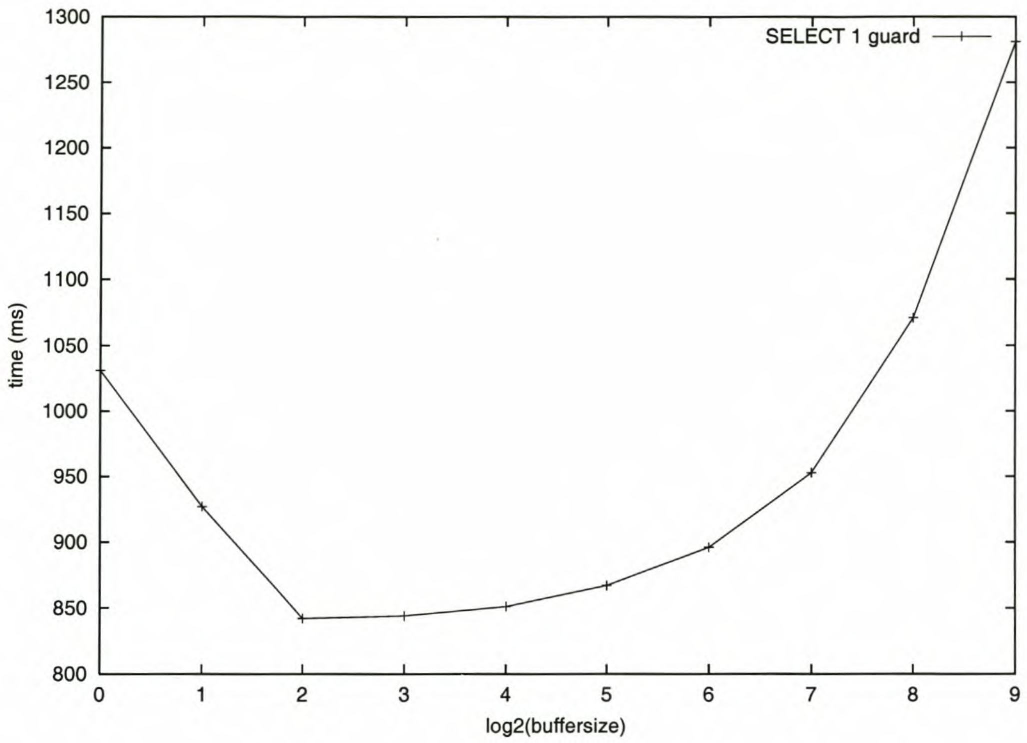

Figure 32: Times for a SELECT with one guard

plotted with respect to the size of the data copied (as in the above graphs). The overhead varies between about 5 and 13 percent. As is to be expected the penalty for larger buffers is significantly less than that for small buffers.

The final experiment in this section sheds some light onto the performance of a `SELECT` with multiple guards. The code in Figure 34 was used to test this performance. The message `t` was varied to force the selection of the different guards. As is to be expected performance is a linear function of the number of guards. This evident from Figure 35.

The linear scaling of execution times for the `SELECT` with the number of guards make estimates of the relative impact of the construct in programs easy to assess. This is good news in performance critical code. An optimising compiler can also help to improve these times. At the moment the stack is still used for some parameters, and even the passing of values in registers which is implemented incurs some unnecessary overhead as a lot of redundant register shuffling is done. Refers to Section 4.14 for a discussion of stackless execution of **LF** processes.

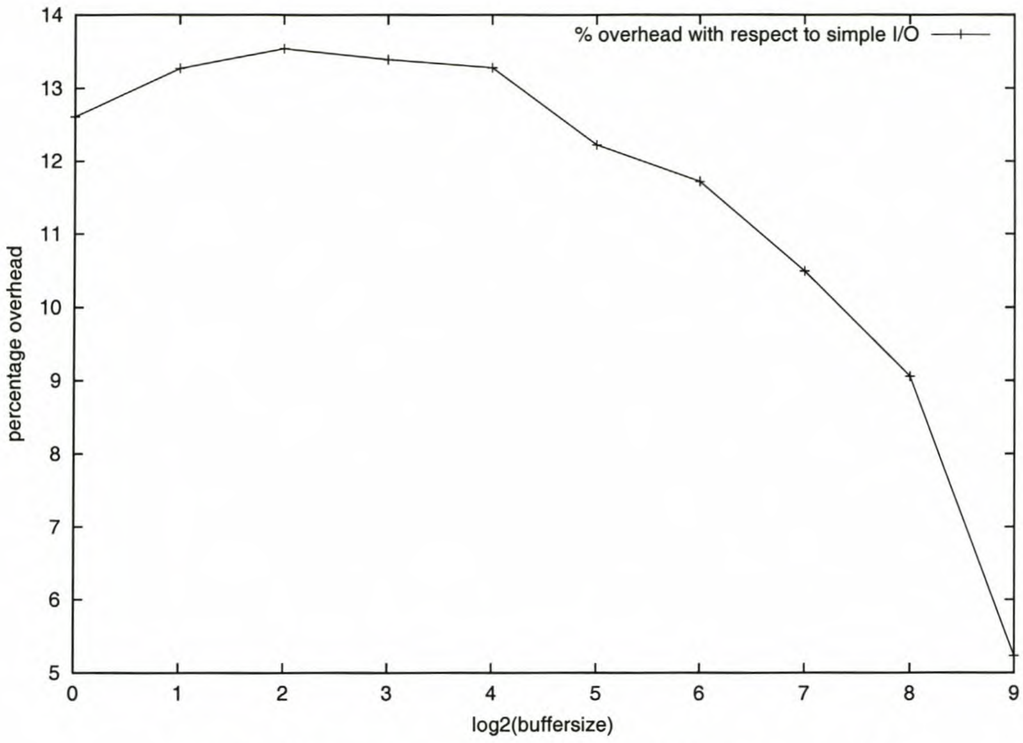

Figure 33: Percentage overhead of select relative to simple I/O

4.11 Runtime checks

Runtime checks form an integral part of the **LF** system. This section provides some background to the clock cycle analysis of the checks that are performed in Section 4.12

- Deadlock detection is **not** performed, because it would be impractical. A process waiting on an interrupt would not be ready to be scheduled. It is conceivable that no other processes can advance until the process waiting on the interrupt proceeds. The possibility therefore exists that periodically no processes are ready for execution. Clearly taking this scenario into account will complicate deadlock detection significantly.

- Processes containing variables defined at absolute addresses (`"AT"` construct) are not allowed to be invoked more than once. A runtime check is done whenever such a process is invoked. The overhead associated with this may be ignored, as it is done only once for each instantiation of a limited number of processes.

- Memory checks are also done. Essentially when a new process is created, or when memory is allocated on the heap, the system checks whether enough memory is available. This amounts to checking whether the allocation of a new entity would lead to a stack and heap collision. The overhead associated with this is negligible when compared to the overhead of actually

```
TYPE
 Msg = [n(UINT32)];

PROCESS Sender( OUT m : Msg );
VAR
  i :  UINT32;
  t : UINT32;
BEGIN
  i := 0; t := 1;
  WHILE i < 10000 DO
    m ! n(t );
   INC( i,1 )
  END;
  HALT( 3 )
END Sender;

PROCESS Receiver( IN m : Msg );
VAR
  t : UINT32;
BEGIN
  WHILE TRUE DO
    SELECT
      m ? n( t ) & t = 1 THEN
      [] m ? n( t ) & t = 2 THEN
      [] m ? n( t ) & t = 3 THEN
      [] m ? n( t ) & t = 4 THEN
      [] m ? n( t ) & t = 5 THEN
    END
  END
END Receiver;
```

Figure 34: Test program for multiple-guard SELECT

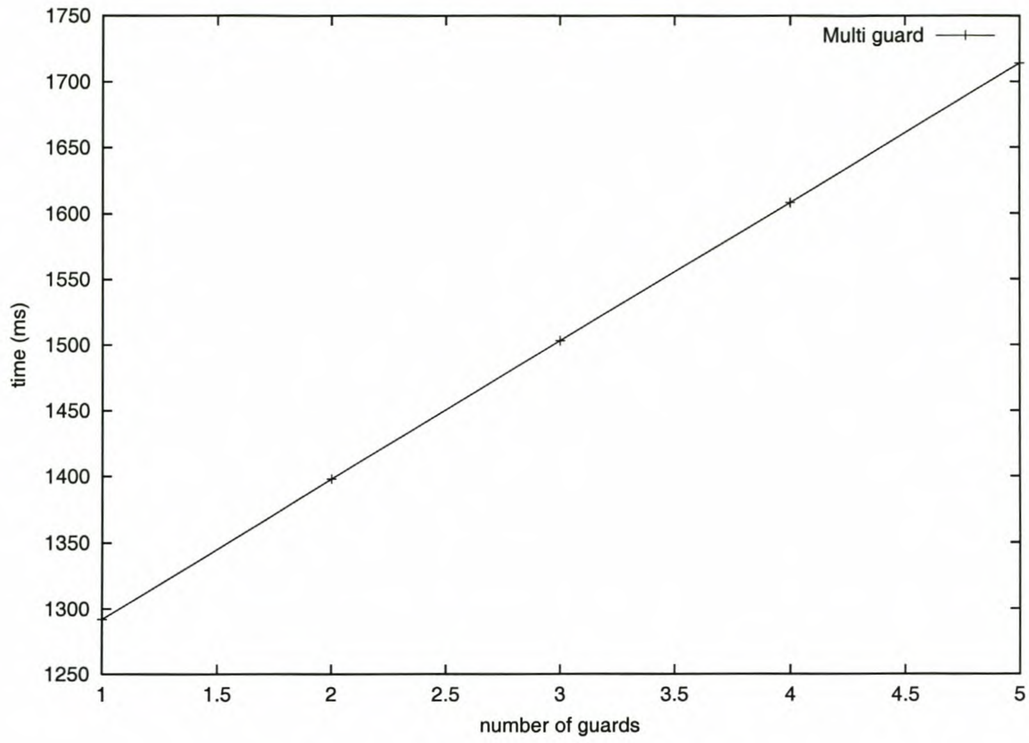

Figure 35: Select with multiple guards

creating a process. It does become more prominent when allocating heap memory, but is still of little consequence.

- Overflow checks on arithmetic operations (addition, subtraction, multiplication and division) may lead to significant overhead depending on the frequency of such operations.

- Array index checking similarly incurs significant runtime overhead. Unfortunately, as with overflow checking, this test is indispensable in guaranteeing the integrity of the system.

- A runtime check for references to uninitialised ports is done. This represents substantial overhead as a test is done for each reference to the port, but is necessary to ensure security. A possible optimisation would be to only perform the test for the first reference to the port within each process. There is no way that a previously initialised port can be de-initialised. This would eliminate most of the overhead, but would complicate the compiler significantly.

- A similar check is done for dereferencing of `NIL` pointers. The overhead is identical to that of checks for uninitialised ports.

4.12 Clock cycle analysis

In this section the execution of pertinent instruction sequences are discussed in terms of the Intel *i386* architecture. This is done to determine the effect of the required runtime checks on the systems performance. Other hardware specific topics such as the determining the maximum interrupt throughput is also examined. Specifically data is presented as pertaining to the *386EX* embedded controller, which represents a typical target platform for **LF** [19]. Table 6 gives the number of clock cycles required for a number of important instructions on the Intel *386EX* embedded processor.

The list of clock cycles are subject to the following assumptions (found on page E-1 of the manual)[19]:

- The instruction has been prefetched, decoded and is ready for execution
- Bus cycles do not require wait states
- There are no local bus HOLD requests, delaying processor access to the bus
- No exceptions are detected during instruction executions
- Add one clock cycle when an effective address calculation requires more than one general register component e.g `[EAX+2*EDX]`

The values also do not reflect the effect of misaligned data (page E-1 in the Intel manual[19]). The above assumptions represent a significant simplification of the execution environment and is in no way a realistic representation of actual execution times. They do however represent a consistent way of determining the **relative** costs associated with instruction sequences. Real world complications to the model are incorporated as of Section 4.12.6.

Please note that only addressing modes and instruction modes relevant to **LF** are listed. Cycles specified as a base value, n, plus an m value (e.g $7 + m$), indicates that the instruction requires n cycles plus one cycle for each component of the next instruction. It is unclear why the format of a following instruction influences the current instruction. It is suspected that it has something to do with the alignment of instructions in the prefetch buffer or execution pipeline. Components of an operand are defined as follows:

- An immediate operand constitutes a component e.g the `32` in `MOV EDX,DWORD 32`.
- A displacement constitutes a component e.g. the `123` in `MOV EAX,123[EDX]`.
- Every other byte of an instruction constitutes an additional component.

Instruction	386EX	Notes
`ADD`	2	Register to Register
	7	Register to Memory
	6	Memory to Register
	2/7	Immediate to Register/Memory
`CALL`	9+m	Call direct
`CLI`	8	
`CMP`	2	Register to Register
	5	Memory to Register
	6	Register to Memory
	2/5	Immediate to Register/Memory
	2	Immediate to accumulator
`INT n`	71	Via Interrupt of Trap gate, Same privilege level
`INT 3`	71	ditto
`INTO`	3	No overflow
	71	Overflow. Via Interrupt or Trap gate
`IRET`	42	Return to same privilege level
`JB`	3	Not taken
	7+m	Taken
`JMP`	7+m	Short or Direct within segment
	9+m	Direct Register/Memory within segment
`LEA`	2	Load address
`MOV`	2	Register to Register
	2	Register to Memory
	4	Memory to Register
	2	Immediate to Register
`MUL`	12-17	Multiplier is 8 bit Register
	15-20	Multiplier is 8 bit Memory
	12-25	Multiplier is 16 bit Register
	15-28	Multiplier is 16 bit Memory
	12-41	Multiplier is 32 bit Register
	17-46	Multiplier is 32 bit Memory
`POP`	5/7	Pop into Register/Memory
	7	Pop into Memory
	6	Pop Register (short form)
`POPAD`	29	Pop General purpose Registers
`PUSH`	5/7	Push Register/Memory
	2	Register (short form)
`PUSHAD`	18	Push General purpose Registers
`RET`	12+m	Within segment
	12+m	Within segment adding Immediate to ESP
`SHL`	3/7	Arithmetic shift Register/Memory by Immediate
`STI`	8	

Table 6: Instructions and clock cycles on the i386EX

4.12.1 Effective overhead of runtime checks

The effective overhead of the most prevalent runtime checks (Index, overflow and NIL checks) are discussed in this section. The less frequently encountered checks such as the check for the availability of memory are not examined here. This is motivated by the fact that due to the **presumed** infrequency of these tests, they are expected to have a limited impact on the **general** efficiency of the program. The fact that they appear sporadically in programs also makes it very difficult to determine their actual relevance in performance evaluation.

4.12.2 Index, overflow and NIL-dereference checks

Index checks have the following general form:

```
CMP register,immediate  ; compare index register/immediate
JB 3                    ; jump over the following two instructions
PUSH 7                  ; push 7 to the stack
INT 3                   ; force an interrupt 3
```

Figure 36: Index checks

When comparing this code sequence with **Table 6** one finds that it requires 5 cycles when the index is within bounds and 78 cycles when not. The cycles consumed when a runtime assertion is violated is of no consequence to the performance of the system and will therefore no longer form part of this discussion. **NIL-dereference** checks are similar:

```
CMP register,0          ; compare index register/immediate
JNE 3                   ; jump over the following two instructions
PUSH 84                 ; push 84 to the stack
INT 3                   ; force an interrupt 3
```

Figure 37: `NIL` checks

A clock cycle analysis yields the same results as for index checking. **Overflow** checking consists of a single `INTO` instruction which translates to 3 cycles when no overflow occurs [3]. Three to five clock cycles may not amount to much in most programs. However, if programs make frequent use of arrays or pointers, such checks may severely impact on the performance of a program. An example should illustrate the point. The code listed below, produces the assembler code listed in Figure 39.

[3]This is the case for unsigned data. Tests on signed data results in code requiring the same number of cycles. It has been omitted for the sake of brevity

```
VAR
  A : ARRAY 10 OF UINT32;
  j,z: UINT32;

BEGIN
  z := A[j]
```

Figure 38: A simple array reference

```
MOV EBX,-44[EBP]      ; value of j
CMP EBX,10            ; index check (3 cycles if passed)
JB 3
PUSH 7
INT 3

MOV EAX,EBX           ; multiply by size of array element
MOV EBX,4
MUL EBX

LEA ECX,-4[EBP]       ; address of A
ADD ECX,EBX           ; add offset
MOV ECX,[ECX]         ; get value
LEA EDI,-48[EBP]
MOV [EDI],ECX         ; assign
```

Figure 39: Index checks

The instruction sequence requires 64 clock cycles in total, implying that the overhead of range checking is in the order of 5 percent. When the multiplication is replaced by a shift operation, as is done by the compiler, the instruction sequence requires 26 cycles. The overhead increases to about 12 percent, which is significant.

The effect of **overflow checking** is even more pronounced. An `INTO` instruction would typically be emitted after each `ADD` and `SUB` [4] instruction and also after each `MUL` instruction. Table 6 indicated that this instruction requires 3 cycles to execute. This compares favourably to a `MUL` instruction, but *executes in more cycles than is required by an ADD* in most cases.

4.12.3 Overhead of variable initialisation

The zero initialisation of variables plays a prominent part in the security of the **LF** system. This issue was addressed in Sections 2.1.3 and 4.11. The overhead associated may be relatively high in terms of instruction counts, but one must see this as part of process instantiation. When seen in this light these instructions have relatively little impact on execution times. Moreover, they only execute once for each process instantiation. When viewed in terms of the size of the process as a

[4]The number of cycles required by `ADD` and `SUB` are identical

whole, these few additional instructions are insignificant.

4.12.4 Cost of a system call

A *minimal system call* may be defined as the least number of operations performed to successfully change from the calling process to the runtime system and back again. Let us assume that this is the null system call i.e. it does nothing. The processing needed to accommodate such a system call would then be that of the context switches to and from the runtime system. System calls are typically implemented with the aid of interrupts. This general form of system call will be compared to the approach taken by **LF**, that of implementing system calls with procedure calls.

```
calling process:
  MOV EAX,functionValue    ; pass request id in EAX
  INT 80h                  ; assume 80h refers to system call handler
```

Upon execution of the interrupt the processor **automatically** executes a number of operations to save the state of the current process. The operations performed vary dramatically with the addressing mode used and the setup of the processor. We will assume the simplest of system setups: hardware protection disabled and an unsegmented linear address space.

```
The following actions are implicitly executed by processor:
PUSH EIP     ; push program counter
PUSH CS      ; push value of CS segment selector
PUSH EFLAGS  ; push value of processor flags
```

Two steps must now be taken by the interrupt handler. In order to resume execution of the interrupted process, the interrupt handler is forced to save the contents of the general purpose registers. A stack switch must be executed to the stack of the runtime system. The order in which this is done depends on the implementation. For the sake of this example assume that the registers are saved before the stack switch.

```
PUSHAD                ; push general purpose registers to stack
MOV ESP,rtStkvalue    ; initialise runtime stack to some value
```

As a general rule, a system call does not return to the calling process. Therefore at the very least the original stack pointer value needs to be recorded by the interrupt handler. For the sake of argument we shall assume that this is done by a single `MOV` instruction with undisclosed operands. In most cases a process is resumed with an interrupt return (`IRET`) instruction. This undoes the operations performed by a `INT nn` instruction. Before the `IRET` is executed the stack must be switched back to the process to resume.

```
calling process:
  MOV EAX,functionValue    ; pass request id in EAX
  INT 80h                  ; assume 80h refers to system call handler
  POPAD
```

Figure 40: System call restoring machine state

```
calling process:
  MOV EAX,functionValue          (2)
  INT 80h                        (71)
  POPAD                          (29)

handler:
  PUSHAD                         (18)
  MOV ESP,rtStkvalue             (2)
  ; save stack pointer of process (2)
  ; do absolutely nothing
  MOV ESP,resStack               (2)
  IRETD                          (42)
```

Figure 41: Overhead of a null system call

```
MOV ESP,resStack
```

The values of the general purpose registers must be restored with a `POPAD` instruction. In our example the easiest way to accomplish this would be to let the compiler insert this instruction after the system call thus augmenting the code as in Figure 40.

The full sequence of instructions with their associated clock cycles are listed in Figure 41: The result is an overhead of 166 clock cycles without actually doing anything. A ***procedure call*** on the other hand firstly executes on the same stack and secondly does not require the saving of context information as it executes in the same context as the calling process. A procedure is instantiated with the `CALL` instruction. This instruction implicitly executes a `PUSH EIP` to store the return address of the procedure call. An assembly listing of a procedure skeleton is shown in Figure 42.

A procedure call only requires 34+n clock cycles (where n refers to the number of components of the instructions following the `RET` instruction). A minimal system call is therefore in the order of 5 times slower than a similar procedure call. It should be noted that the above example states the best case performance of a system call. There is usually additional overhead to interface with high level code within the runtime system. The actual figures for **LF** context switches are presented in the next sections.

```
CALL address (9+m)
; m = 1
PUSH EBP     (2)
MOV EBP,ESP  (2)

MOV ESP,EBP (2)
POP ESP   (6)
RET (12+n)
```

Figure 42: An empty procedure

4.12.5 Interrupt latency in the LF Runtime system

Interrupts must be serviced as quickly as possible. The runtime system must therefore invoke the appropriate process to handle the interrupt as quickly as possible.

The sequence of events in the handling of an interrupt is described in Section 4.5.2. The occurrence of an interrupt triggers the execution of the code in Figure 43. The scheduler has been left empty. Much of the discussion of Section 4.12.4 is also applicable here. Although the code is not complicated, an understanding of Section 4.5.2 is a prerequisite. Lines 01 to 05 constitute the low level interrupt handler. Line 01 is only executed in cases where the processor does not supply an error code. In line 02 the number of the interrupt that occurred is pushed. Line 03 stores the values of the general purpose registers on the stack. In lines 04 and 05 a call is made to the high-level handler. Lines 06 and 07 create an activation record for the **Oberon** procedure (Compare this to Figure 42.). In line 08 a local variable is instantiated. Lines 10 to 14 constitutes the termination of the interrupt handler. The procedure `Continue` was discussed in Section 4.5.2. This represents a total of 227 clock cycles (assuming a value of `4` for `m`) which translates into about 6,89μs for a context switch for a 33MHz processor.

4.12.6 Actual Performance

The analysis in the above section indicates exceptional performance. A similar operation in the QNX real-time micro kernel takes about 54μs on the same machine [14] [5]. QNX is a successful commercial product. Actual performance is however much worse. In fact the code in Figure 43 took almost **twice** as long to execute as was calculated above. The performance figures were gathered by executing one **LF** process which continuously forces a context switch. The number of cycles required by the scheduler upon a software interrupt is `170` cycles. This adds up to `405` cycles when merged with the code in Figure 43 and the minimal code of the process. This should execute in about `12,03`μs, but the actual performance is in the order of 22μs.

[5]Strictly speaking a micro-kernel and the less intricate architecture of **LF** cannot be directly compared as the operations of the micro-kernel are more complex, but this comparison does place the figures presented into some sort of context.

```
; processor needs 71 cycles to process interrupt
; low level handler
01 PUSH 0                    (4)
02 PUSH intno                (4)
03 PUSHAD                    (34)
04 MOV EAX,address_handler (2)
05 CALL EAX                  (9+m)
; activation of Scheduler (Oberon)
06 PUSH EBP                  (4)
07 MOV EBP,ESP               (2)
08 SUB ESP,4                 (2)
09 ; scheduler code
:

; call to continue
10 PUSH register             (7) ; register contains new ESP value
;Continue
11 POP ESP                   (6)
12 POPAD                     (35)
13 ADD ESP,8                 (2)
14 IRETD                     (42)
```

Figure 43: Interrupt overhead

The disparity between expected and measured performance can be accounted for when one considers the architecture of the *386EX* processor. Internally it is a full 32-bit machine, but it has a 16-bit bus interface. Both code and data use the same bus to access memory. Reading 32-bits from memory therefore requires an additional fetch from memory for each instance. Reading 32-bits of memory then requires 4 cycles instead of 2.

The processor prefetches code 16-bytes at a time, using a prefetch buffer from which the execution unit executes the code. When the prefetch buffer is not empty prefetches only occur when the bus is idle i.e. not used to access data. When the buffer does become empty the processor stalls and a prefetch is forced. The prefetch buffer is flushed whenever a branching instruction takes place. A prefetch fetches 16-bytes as noted above, and therefore occurs within 2 clock cycles. The prefetch buffer is 6 bytes in size.

The code which constitutes the interrupt handling and scheduling contain a large amount of branching instructions relative to other instructions —on average a branching instruction every 8 instructions— Most of the other instructions reference memory and therefore the prefetch buffer is not allowed to fill once it has been flushed. Effectively a memory access needs to be done for every second byte of code.

When adding up the effect of all of these considerations:

Number of switches	Time per switch (μs)
428 199	23,35
426 981	23,42
428 047	23,36
428 565	23,33
428 570	23,33
428 270	**23,34**

Table 7: Actual context switch times

405	$(cycles\ as\ per\ manuals)$
132	$(cycles\ because\ of\ 32-bit\ references)$
174	$(cycles\ for\ code\ fetches)$
35	$(DRAM\ refresh)$
747	

The last complication is the refresh overhead of refreshing DRAM on the computer. This has an impact of about 5% overhead [1]. This amounts to 747 cycles or 22,6μs.

The actual execution times measured are listed in Table 7. The number of switches were measured over a 5 minute interval and then converted to the number of switches per second (rounded to the nearest whole number). The average values are listed at the bottom of the table. This represents a 3.1% deviation from the expected value.

In the event of a hardware interrupt, the shortest path through the scheduler is the instance where a handler is blocking on the interrupt. If one follows a similar analysis to the above, a time of around 27μs is obtained. This suggests an upper bound of 37000 serviceable interrupts per second. A more pragmatic approach would be the assumption that an interrupt handler would execute at least 3 times as many cycles as the overhead associated with the interrupt. The value three is purely a heuristic and was decided upon after examination of small to medium size interrupt handlers. This suggests a practical upper bound of in the order of 12000 serviceable interrupts.

At an interrupt frequency of around 60kHz the interrupt subsystem collapses. The collapse manifests itself in the form of the spurious occurrence of interrupt 39. Although a software error has not been ruled out completely, an inherent problem with the interrupt controller hardware is suspected. Of course the actual throughput of the system is nowhere near 60kHz. The test program listed in Figure 44 was used to determine the maximal throughput for interrupts. It was found that interrupt events started to be dropped at around 13.8kHz.

```
PROGRAM PerfInts;
PROCESS Handler;
VAR
  snd : UINT32;
BEGIN
  NEW( IntChannels[32] );
  PORTOUT( 67,54 );
  PORTOUT( 64,$56 );PORTOUT( 64,$0 );  (* ~13,9 kHz *)
  snd := 0;
  WHILE TRUE DO
   IntChannels[32] ? sig; INC( snd,1 );
  END
END Handler;

BEGIN
  Handler
END PerfInts.
```

Figure 44: Test code for interrupts.

4.12.7 Copying messages

It is of critical importance that messages be copied as fast as possible. The problem with optimising copies lies is the fact that messages may have any length. The implications are:

- Messages are not necessarily aligned on word boundaries
- A general mechanism that operates as efficiently as possible on both short and long messages is needed. A compromise therefore needs to be reached.

Accessing unaligned data is exceedingly expensive on the *i386*. Even in the best case a penalty of 2 or 4 extra clock cycles is incurred for each access to a misaligned 16 or 32 bit operand on an even address. When the addresses are not even, the penalty is **doubled**. In most cases the penalty is even worse because of the heightened memory contention with the prefetcher which is caused by the additional clock cycles needed to access memory. Unfortunately this is something which has to be lived with.

4.13 The size of the runtime system

The runtime system is implemented in the high level language **Oberon**. Despite this fact, it remains noticeably compact. The sizes of the modules comprising the runtime system are listed in Table 8. Note that about 124 lines of `LFRuntime.Mod` is debugging code. The memory requirements of the runtime system may be found in Table 9. The static image contains the runtime system code as well as global variables. An activation record (AR) is created for each instance of

Module	Lines	Code Size (bytes)
LFRuntime	443	4399
LFHeap	224	1702
Assembler code	66	273
LFProcess (Oberon code)	295	2076
LFProcess (Assembler code)	852	1868

Table 8: Size of the Runtime System

Static image	
Process AR	92
Channel AR	4+8n

Table 9: Runtime system memory overhead (bytes)

a process or channel. Two sets of values are listed for *LFProcess* in Table 8. The first set refers to the version of the module which implements IPC in high level code. The second set refers to the version of the module implementing IPC in assembler.

The `n` in table 9 refers to the number of symbols in the alphabet associated with the channel. A typical value for the size of the runtime system static image is about `24kB`.

4.14 Stackless execution model

It is perfectly feasible for processes in **LF** to execute without a stack. Stackless execution is highly desirable since a significant amount of memory is saved. Operations on the stack are also notoriously expensive on the *i386* architecture. Although this may be less of a concern on other architectures, it is expected that a `PUSH` or `POP` operation will always be more expensive than a direct memory access. At the moment memory is reserved within each activation record for the allocation of the variables of a process, as well as an expression stack. The size of the block of memory is set to the maximum amount needed by the process, as determined at compile time (Section 2.4). The current version of the compiler does not make use of this stack for the evaluation of expressions. Effectively the stack is only used to pass parameters to the procedures of the runtime system and by the procedures of the runtime system themselves. The calls to the runtime system execute in the same stack frame as the calling process. A version of the compiler has been implemented that passes most of its parameters in registers in any case. With little additional effort, the use of registers can completely replace the use of the stack for the passing of parameters to the runtime system procedures. The use of temporary variables, may replace the use of the stack as supplementary storage space during the evaluation of complex expressions. As was noted in Section 2.6.1, the current version of the compiler is limited in the complexity of expressions that it accepts.

When one examines the runtime system closely, a major problem with stackless execution become apparent: the runtime system is implemented in a high level language, which requires a stack for the correct execution of its code.

Possible solutions to this problem are:

1. Re-code the entire runtime system in assembler. This is of course perfectly feasible, but negates the maintenance and portability benefits of a high level implementation.
2. Rewrite the runtime system code to make use of global temporary variables, rather than allocating local variables upon each instantiation of a process. This may well be faster than the current implementation because there is no need to instantiate local variables. This is poor programming practice and may lead to a very buggy implementation.
3. Essentially keep the runtime system in its current form, but create a stack for the runtime system. This will of course require a stack switch for each call to the runtime system. This is the easiest solution, but also incurs a significant performance penalty. In the current implementation, stack switches are only required when creating a new channel or process instance, or in the case of a context switch (in which case only a single stack switch is executed). Having a runtime system stack will require two stack switches for every call to the runtime system.

The passing of strings and other (potentially) large data structures as parameters complicates solution 1, but especially solution 2. In my opinion the best compromise would be to combine solutions 2 and 3. This would amount to the following: Code all procedures in the runtime system that are not related to the servicing of interrupts to make use of temporary variables. These procedures typically require in the order of 12 bytes of stack space. In addition to this allocate a small stack to the runtime system that is used by the interrupt servicing routines in the runtime system. The net effect is that an additional (with respect to the current implementation) stack switch is only forced in the event of an interrupt or exception occurring. When exiting the runtime system after the servicing of an interrupt, a full context switch to the next process scheduled is executed in any case, so this does not represent any additional overhead.

4.15 Conclusion

In this chapter the design and implementation of the **LF** runtime system has been discussed. Many performance measures have also been listed. Even when given the relatively simple design of the system, communication performance and interrupt response times are more than adequate and are in fact far superior to my initial estimates.

The overhead resulting from runtime checks is acceptable, although this may be arguable in the case of overflow checks. Searching for a more efficient way to implement overflow checks, may be worth while. Similarly a more efficient way to check for uninitialised channels should be implemented. These checks, do however, demonstrate the feasibility of software implemented memory protection.

With respect to the design criteria noted at the start of this chapter: I believe that a very compact runtime system has been implemented. This has a direct positive influence on memory usage and the maintainability of the system. It also —generally speaking— leads to faster execution. Security was never compromised to improve execution speeds.

One of the most interesting aspects relating to the runtime system was encountered when the clock cycle analysis of the system's interrupt response was done. It again became apparent that the only way to attain reasonably accurate performance figures is by measurement. My initial estimates, based solely on processor documentation, and the true values differed by approximately 100%. It was only through additional research **after** the disparity had been discovered, that I came to understand the true dynamics of the execution environment.

Chapter 5

Evaluation and Conclusion

This chapter presents my evaluation of the **LF** system, specifically the degree to which the objectives mentioned in Chapter 1 have been attained. Some thoughts that I have with regard to future extensions are also presented.

5.1 Evaluation of the LF language

At the time of completion of this thesis, a usable version of the **LF** system had been implemented. **LF** is being used as implementation language for a number of medium-sized projects related to protocol implementation. These projects will be the first real test of the capabilities of **LF** as embedded programming language. Some modifications will have to be made as **LF** matures. These changes will in all likelihood affect all aspects of **LF** i.e. the language definition as well as the compiler and runtime system. **LF** was kept as simple as possible. By only defining the most crucial elements of the language at this early stage in its development and keeping the runtime system as simple as possible, it is hoped that the runtime system can be modified to cater for special needs rather than complicating the language. **LF** can, at the very least, be used as a teaching language for concurrency. The concise and compact nature of the language, when combined with the strictness of its semantics, is well suited to a teaching language. I also believe that after an appropriate gestation period, **LF** will evolve into a viable programming language for embedded applications.

One of the primary motivations for the development of **LF** was the desire to support model checking at the code level. To this end another project revolves around the automated conversion of **LF** code to **Promela** [9]. Initial results are promising: a simple implementation of an alternating bit protocol in **LF** was converted and model checked at code level. This gives some indication that **LF** can be used to implement and verify protocols. A full listing of the **LF** code and corresponding **Promela** specification may be found in Appendix C.

Readers familiar with **CSP** would have noticed that several of the features of **CSP** are not included in **LF**. This was done in the interest of creating a compact language. A more complete implementation of **CSP** would unavoidably have been more complex. It is important to bear in mind that for the purposes of this project compatibility with **Promela** is far more relevant than compatibility with **CSP** per se. A fairly direct mapping exists between the constructs of **LF** and **Promela**.

Many other issues have been raised during the evolution of this project. This project illustrates the viability of software enforced memory protection in embedded systems. It also makes a strong case for the practicality of compact languages (as does **Oberon**). Although no embedded programs of realistic size and complexity have been implemented in **LF**, I believe that the IPC/process paradigm can successfully replace the procedure paradigm; **occam** has paved the way in this respect. I feel that **LF** has a more congenial syntax and is somewhat more flexible than **occam**. Moreover **LF** was not designed with a specific hardware configuration in mind. Most importantly, **LF** offers the benefits of strict type checking. Fisher criticises **occam's** and **occam 2's** channel type system in [11]. Of course message passing can never be as efficient as (say) a combination of semaphores and shared memory, but this was never the objective. **LF** is intended to be a concise language that is easy to use (the use of semaphores is notoriously treacherous) and to be efficient *enough.* These goals seem to have been attained.

5.2 Evaluation of the Compiler and Runtime System

The **LF** compiler is very simple. This at times leads to the generation of inefficient code. The complexities of optimising compilers were however beyond the scope of this project. Despite its inadequacies the code produced was sufficient for the purposes of this thesis and is indeed reasonable in most cases. Related to the compiler is the assembler used. It too is very simple and at times produces less than optimal object code. This too is of little consequence in the scope of this thesis.

The runtime system is relatively compact. Functionality such as scheduling is very basic and may need to be enhanced as the need arises. At this stage it seems adequate. The modular structure of the runtime system should make extensions relatively easy. The compiler and runtime system manage to enforce memory protection at relatively low cost. This is especially significant when considering that many embedded platforms do not implement hardware memory protection.

Communication is much slower than procedure calls. This is to be expected. It is, however, significantly faster than I expected initially. Interrupt response times look equally encouraging, but it remains to be seen how the performance of interrupt handlers will be influenced by the need to communicate with other processes in complex systems.

LF was designed to be a secure system, but some hardware related factors undermine this security.

1. The problem relating to restricting access to I/O ports was discussed in Chapter 1.
2. Related to the problem of I/O port restrictions, is a caveat related to DMA on the IBM PC: DMA bypasses the memory protection hardware. This is a serious problem indeed, but there does not seem to be an effective solution.
3. The problems associated with redefining an existing exception handler were discussed in Section 2.4.

These problems compromise the security of **LF**. Not trying to remedy factors 1 and 3 has been motivated, while factor 2 is essentially the result of poor hardware design that cannot be fixed effectively in software. Moreover, these problems relate to very specific circumstances that are unlikely to cause problems in general programming practice.

5.3 Future work

The **LF** project, in its current state, offers much opportunity for future work. Some aspects of the system can and must be enhanced or rewritten for it to be of practical use. These aspects are discussed below.

- The expressiveness and usability of **LF** can be considerably enhanced by extending its repertoire of constructs.
  - A **CASE** statement should be considered to increase efficiency.
  - **FOR** loops are generally more efficient when the number of iterations are known at compile time.
  - Since there is no *string* type, the ability to assign literal strings to an `array of char` structure would make programming more convenient. The following syntax is proposed:

    ```
    VAR
      s : ARRAY 10 OF CHAR;
    BEGIN
      s := "hallo";
      ⋮
    ```

  - An important omission from both the **LF** language definition and the runtime system is support for **floating point** numbers. A practical implementation would in all likelihood need to add such functionality.
  - At the moment all processes are activated in parallel. It may be deemed necessary to introduce constructs that allow for either the sequential or the interleaved/parallel [1]

[1]Depending on whether a uni- or multiprocessor is involved

execution of processes, to mirror the constructs of **CSP**. This functionality is indeed part of **occam**, which contains a `PAR` and `SEQ` construct that indicates parallel and sequential execution of statements respectively [2]. A possible **LF** incarnation might be the following:

```
P1||P2||P3
```

to indicate the parallel execution of processes and

```
P1;P2;P3
```

to indicate the sequential execution of processes.

- In order to make the **LF** system viable an optimising compiler needs to be developed. This is of course a non-trivial exercise. The current scenario of hand optimisation is tedious and error prone.
- As explained in Chapter 2, **LF** does not support modularisation. Modularisation will not only remove the need for unwieldy program files, but also promote the use of libraries of processes.
- Given the concurrent nature of the language a multi-processor[3] implementation is the obvious next step. **LF** is based on **CSP**. As such the notion of disjoint address spaces as mirrored in the process concept is a natural model on which to build multi-processor extensions. A shared memory implementation is of course also possible.

  When a multi-processor implementation is created, the language definition would have to be extended or some other method found to specify which processes should run on which processor.

5.4 Final Remarks

LF is a secure language suited to the implementation of embedded software. It features a clear, concise syntax. I believe an amiable compromise between practicality —in the sense of performance and expressiveness— and security has been achieved. **LF** is designed to be model checked at code level. This further improves the ability of the language to aid the programmer in producing correct software.

[2]**occam** allows for parallelism at statement level

[3]Multi-processor is used in the broadest sense of the word, to includes distributed computing as well.

Appendix A

EBNF

⟨program⟩ ::= `PROGRAM` *identifier*; ⟨declarations⟩⟨processes⟩ ⟨body⟩ .

⟨processes⟩ ::= { ⟨process⟩ }

⟨process⟩ ::= `PROCESS` *identifier*[⟨parameter list⟩] ; ⟨declarations⟩ ⟨body⟩ ;

⟨body⟩ ::= `BEGIN` ⟨command list⟩ `END` *identifier*

A.0.1 Declarations

⟨parameter list ⟩ ::= (⟨parameter⟩ { ; ⟨parameter⟩ })

⟨parameter⟩ ::= ⟨variable definition⟩

⟨declarations⟩ ::= {[⟨constant part⟩][⟨type part⟩][⟨variable part⟩]}

⟨constant part⟩ ::= `CONST` ⟨constant definition⟩ { ⟨constant definition⟩ }

⟨constant definition⟩ ::= *identifier* = ⟨constant expression⟩ ;

⟨variable part⟩ ::= `VAR` ⟨variable definition⟩{ ⟨variable definition⟩} ;

⟨type part⟩ ::= `TYPE` ⟨type definition⟩ ; {⟨type definition⟩}

⟨type definition⟩ ::= *identifier* = `POINTER TO` ⟨type definition⟩|
  `ARRAY` ⟨expression⟩ `OF` ⟨type definition⟩|
  ⟨alphabet definition⟩|⟨record definition⟩|⟨type identifier⟩

⟨record definition⟩ ::= `RECORD` ⟨fieldlist⟩ `END`

⟨fieldlist⟩ ::= ⟨identlist⟩⟨type definition⟩{ ; ⟨identlist⟩⟨type definition⟩}

⟨alphabet definition⟩ ::= [⟨symbol⟩ , {⟨symbol⟩}]

⟨symbol⟩ ::= *identifier*[(⟨type identifier⟩)]

⟨variable definition⟩ ::= [`IN`|`OUT`]*identifier*{ , *identifier*} : ⟨type identifier⟩[⟨at def⟩]

⟨at def⟩ ::= `AT` *number*

⟨type identifier⟩ ::= ⟨simple type⟩|*identifier*

⟨simple type⟩ ::= ⟨integer type[1]⟩|`CHAR`|`BOOLEAN`|⟨set type[2]⟩

[1]Platform specific
[2]Platform specific

⟨integer type⟩ ::= ⟨signedint⟩ | ⟨unsignedint⟩
⟨signedint⟩ ::= `INT8`|`INT16`|`INT32`
⟨unsignedint⟩ ::= `UINT8`|`UINT16`|`UINT32`
⟨set type⟩ ::= `SET8`|`SET16`|`SET32`

A.0.2 Statements

⟨command list⟩ ::= [⟨command⟩{`;`⟨command⟩}].
⟨command⟩ ::= ⟨if⟩|⟨while⟩|⟨repeat⟩|⟨select⟩|⟨create⟩|⟨access⟩⟨new⟩
⟨access⟩ ::= ⟨variable access⟩(⟨assign⟩|⟨io⟩)
⟨io⟩ ::= ⟨bang⟩|⟨hook⟩
⟨assign⟩ ::= `:=` ⟨expression⟩
⟨bang⟩ ::= `!` *identifier*[`(`⟨expression⟩`)`]
⟨hook⟩ ::= `?` *identifier*[`(`⟨variable access⟩`)`]
⟨repeat⟩ ::= `REPEAT` ⟨command list⟩ `UNTIL` ⟨expression⟩
⟨while⟩ ::= `WHILE` ⟨expression⟩ `DO` ⟨command list⟩ `END`
⟨new⟩ ::= `NEW` `(` ⟨variable access⟩`)`
⟨create⟩ ::= *identifier*[`(`⟨expression⟩{`,`⟨expression⟩`)`}]
⟨if⟩ ::= `IF` ⟨expression⟩ `THEN` ⟨command list⟩
    {`ELSIF`⟨expression⟩`THEN`⟨command list⟩} [`ELSE` ⟨command list⟩]
    `END`
⟨select⟩ ::= `SELECT` ⟨select guard⟩ `THEN` ⟨command list⟩
    {`[]`⟨select guard⟩`THEN`⟨command list⟩
    `END`
⟨select guard⟩ ::= ⟨variable access⟩⟨io⟩[`&`⟨expression⟩]

A.0.3 Expressions

⟨selector⟩ ::= {`[`⟨expression⟩`]`|`.`*identifier*|↑}
⟨variable access⟩ ::= ⟨identifier⟩[⟨selector⟩]
⟨expression⟩ ::= ⟨normal expression⟩|⟨typecast⟩
⟨constant expression⟩ ::= ⟨expression⟩
⟨typecast⟩ ::= `CAST` `(`⟨type identifier⟩`,`⟨expression⟩`)`
⟨normal expression⟩ ::= ⟨primary⟩[⟨primary operator⟩⟨expression⟩]
⟨primary operator⟩ ::= `&` | `|`
⟨primary⟩ ::= ⟨secondary⟩[⟨secondary operator⟩ ⟨secondary⟩]
⟨secondary operator⟩ ::= `<`|`<=`|`>`|`>=`|`=`|`#`
⟨secondary⟩ ::= ⟨term⟩[⟨adding operator⟩ ⟨secondary⟩]

$\langle$adding operator$\rangle$ ::= `+`|`-`

$\langle$term$\rangle$ ::= $\langle$factor$\rangle$[$\langle$multiplying operator$\rangle$$\langle$term$\rangle$]

$\langle$multiplying operator$\rangle$ ::= `*` | `DIV` | `MOD`

$\langle$factor$\rangle$ ::=*number*|`TRUE`|`FALSE`|~$\langle$expression$\rangle$|`-`$\langle$expression$\rangle$|`(`$\langle$expression$\rangle$`)`|
$\langle$variable access$\rangle$|`LONG(`$\langle$expression$\rangle$`)`|`SHORT(`$\langle$expression$\rangle$`)`|*char constant*

A.0.4 Tokens

$\langle$identifier$\rangle$ ::= $\langle$letter$\rangle${$\langle$letter$\rangle$|$\langle$digit$\rangle$}

$\langle$number$\rangle$ ::= [`$`]($\langle$hexdigit$\rangle$|$\langle$digit$\rangle$)|{$\langle$hexdigit$\rangle$$\langle$digit$\rangle$}

$\langle$letter$\rangle$ ::= `a`|...|`z`|`A`|...|`Z`

$\langle$hexdigit$\rangle$ ::= `a`|...|`f`

$\langle$digit$\rangle$ ::= `0`|...|`9`

$\langle$char constant$\rangle$ ::= '*ascii character*'

Appendix B

The boot loader

The boot loader used for the **LF** system was originally developed for the Gneiss micro-kernel [24, 33]. Developing a new loader is not worth the effort in an experimental system. The boot loader loads an executable image from a diskette, and starts to execute the code in the image[1]. Before executing the code it disables all interrupts and does all of the housekeeping to switch to protected mode. After this switch, control is transferred to the executable image. It is up to the executable image to appropriately initialise the hardware. The layout of the memory after the boot loader has finished its execution is illustrated in Figure 45.

The boot loader also sets up a table called the *boottable* which contains the memory layout of the system after booting, as well as the hard drive parameters from the CMOS settings. Two `4kB` segments of memory on both sides of the image are left free. (This is due to historical reasons. In the Gneiss system, the pages on both sides of the kernel were left unmapped to catch `NIL` pointer references.) The next chunk of memory is used during the boot process and contains the boottable and a provisional stack for the image. Another segment of free memory follows the boot memory and stretches up to `09FFFFh`[2]. The segment between `0A0000h` to `01000000h (1MB)` is used for memory mapped devices (such as the display) and ROMs. The memory above `1MB` up to the physical memory limit is free.

[1]The boot loader operates via DOS, which is later expelled from memory after loading the image.
[2]The old Microsoft/IBM glass ceiling

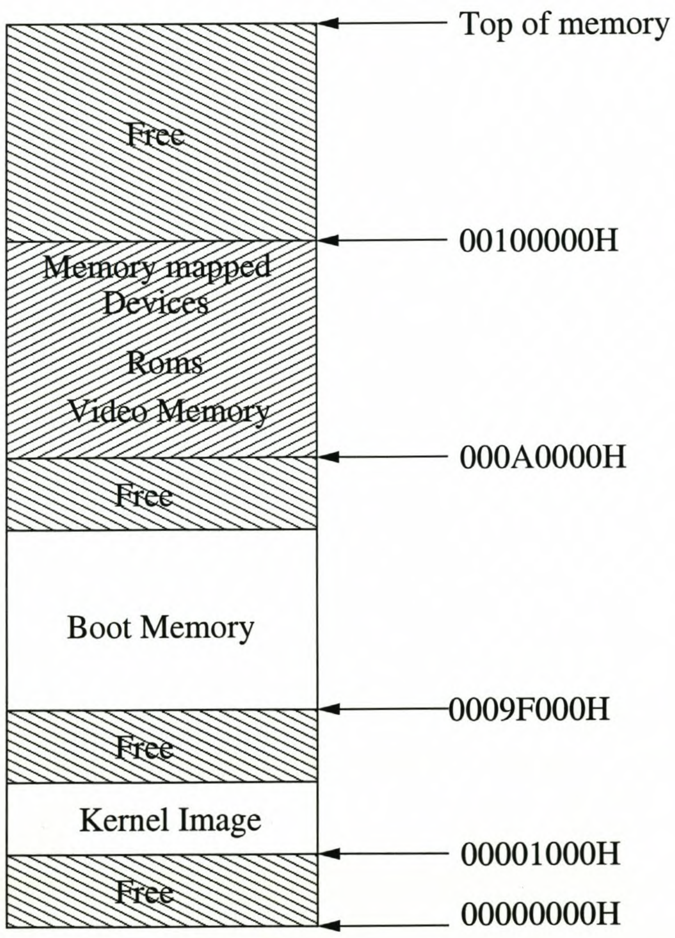

Figure 45: System memory layout as created by the boot loader

Appendix C

LF vs. Promela

Promela is a process oriented language which serves as input to the **SPIN** model checker [18]. For the purposes of this discussion a working knowledge of **Promela** is assumed. An implementation of the Alternating Bit protocol implemented in **LF** is presented along with a **Promela** model which was generated from it. The derivation was done with the use of a tool developed by Dirk Bull[1][9]. The aim of this section is not to describe the use or implementation of the tool, but merely to demonstrate the feasibility of deriving **Promela** from **LF**. It should be clear from the below that a fairly direct mapping exists from **LF** to **Promela**. Some aspects of the mapping are:

- **Promela** does not support record types, therefore the individual components of a record are declared as discrete variables in the **Promela** model.
- Expressions are mapped directly with only syntactic changes from the **Pascal**-like syntax of **LF** to the **C**-like syntax of **Promela**.
- The mapping of control structures such as the `REPEAT` is direct, albeit somewhat convoluted, owing to the explicit non-determinism of **Promela**.
- Communication in **LF** and **Promela** both owe much to **CSP**, the only difference being that structured variables, such as records, are transmitted in terms of their constituent variables in **Promela**.

[1]The **LF** implementation is also by Dirk.

```
PROGRAM AlternatingBit;
TYPE
 MsgType = RECORD
  o : UINT32;
  s : UINT8
 END;
 MesgMsg = [a(MsgType)];
 AckMsg = [a(UINT8)];

PROCESS Receiver(IN mesg:MesgMsg;OUT ack:AckMsg);
VAR
 NewMsg : MsgType;
 e, a : UINT8;
BEGIN
 e := 0; a := 0;
 WHILE TRUE DO
  REPEAT
   mesg ? a(NewMsg);
   a := NewMsg.s;
   ack ! a(a)
  UNTIL a = e;
  e := 1-e
 END
END Receiver;
```

```
proctype Receiver(chan mesg;chan ack)
{
    int NewMsg_o;
    byte NewMsg_s;
    byte e, a;

    e = 0;
    a = 0;
    do
    :: 1 ->
        bool MyRepeatTest1 = 1;
        do
        :: (MyRepeatTest1 == 1) || (a == e) ->
            mesg ? NewMsg_o, NewMsg_s;
            a = NewMsg_s;
            ack ! a;
            MyRepeatTest1 = 0

        :: !(a == e) ->
            break
        od;
        e = 1-e;

    :: !(1) ->
        break
    od
}
```

```
PROCESS Sender(IN ack:AckMsg;OUT mesg:MesgMsg);
VAR
 NewMsg : MsgType;
 r, s : UINT8;
 Counter : UINT32;
BEGIN
 s := 0; r := 0; Counter := 0;
 WHILE TRUE DO
  NewMsg.o := Counter;
  NewMsg.s := s;
  REPEAT
   mesg ! a(NewMsg); ack ? a(r)
  UNTIL r = s;
  s := 1 - s;
  Counter := (Counter + 1) MOD 2000
 END
END Sender;
```

```
proctype Sender(chan ack;chan mesg)
{
    int NewMsg_o;
    byte NewMsg_s;
    byte r, s;
    int Counter;

    s = 0;
    r = 0;
    Counter = 0;
    do
    :: 1 ->
        NewMsg_o = Counter;
        NewMsg_s = s;
        bool MyRepeatTest2 = 1;
        do
        :: (MyRepeatTest2 == 1) || (r == s) ->
            mesg ! NewMsg_o, NewMsg_s;
            ack ? r;
            MyRepeatTest2 = 0

        :: !(r == s) ->
            break
        od;
        s = 1 - s;
        Counter = (Counter + 1) % 2000
    :: !(1) ->
        break
    od
}
```

```
PROCESS Run;
VAR
 IN ack : AckMsg;
 OUT mesg : MesgMsg;
BEGIN
 NEW( ack ); NEW( mesg );
 Sender( ack, mesg );
 Receiver( mesg, ack, I)
END Run;

BEGIN
 Run
END AlternatingBit.
```

```
proctype Run()
{
    chan ack = [0] of { byte };
    chan mesg = [0] of { int, byte };
    run Sender(ack, mesg);
    run Receiver( mesg, ack);
}
init
{
    run Run()
}
```

Bibliography

[1] M. Abrash. *Michael Abrash's Graphics Programming Black Book*. Coriolis Group, Inc., 1997.

[2] A.V. Aho et al. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] G.R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, 1991.

[4] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice*. The Benjamin/Cummings Publishing Company, 1993.

[5] E.J. Berglund. An Introduction to the V-system. *IEEE Micro*, pages 35–52, August 1986.

[6] A.J. Bernstein. Output Guards and Non-Determinism in CSP. *ACM Transactions on Programming Languages and System*, 2:234–238, April 1980.

[7] P. Brinch Hansen. Joyce - A Programming Language for Distributed Systems. *Software-Practice and Experience*, 17:29–50, January 1987.

[8] G.N. Buckley and A. Silberschatz. An Effective Implementation for the Generalized Input-Output Construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[9] J.D. Bull. Verification of software with automated tools (draft). Technical report, University of Stellenbosch, 2001.

[10] D. Cooper and M. Clancy. *Oh! Pascal!* W.W Norton & Company, second edition, 1985.

[11] A.J. Fisher. A Critique of occam Channel Types. *Computer Languages*, 13(2):95–105, February 1988.

[12] Formal Systems(Europe) Ltd. *FDR2 User Manual*, fifth edition, May 2000.

[13] N.H. Gehani and W.D. Roome. Concurrent C. *Software-Practice and Experience*, 16:821–844, September 1986.

[14] D. Hildebrand. An architectural overview of QNX. *Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures*, April 1992.