Christoph Regli

µOBERON

A Development System for Mcs51 Microcontrollers

Diploma Thesis 1996/97
Swiss Federal Institute of Technology
Supervised by Erwin Oertli, Hans Eberle

µOBERON
A Development System for Mcs51 Microcontrollers
Christoph Regli

Institute for Computer Systems
Department of Computer Science
Swiss Federal Institute of Technology, Zurich

Erwin Oertli, Prof. Hans Eberle

Acknowledgements

I would like to express my gratitude to Professor Hans Eberle who gave me the opportunity to do this interesting work. It was a great challenge which continually led to new hurdles that had to be taken. Erwin Oertli who attended me was always a great help and didn't hesitate to contribute new ideas and suggestions. It was exciting to discuss with him about what should be implemented and where the focal points of my interest should be, and his enthusiasm for simple but smart and clever solutions is very infectious. His knowledge about compiler construction seems to be inexhaustible. Jonas Kurth and Pascal Peng deserve special thanks for proofreading this text. They pointed out numerous grammatical and stylistic mistakes, and their corrections and suggestions led to an improved readability.

Finally I am greatly indebted to my parents who enabled me to do my studies, and to whom this work is dedicated.

Kurzfassung

Als aussergewöhnlich vielseitig einsetzbare Bauteile gewinnen Mikrocontroller immer mehr an Bedeutung. Verglichen mit mechanischen oder elektrischen Systemen erhöhen sie Funktionalität und Zuverlässigkeit und reduzieren gleichzeitig Grösse und Kosten. Zudem trägt die Wiederverwendbarkeit von Programm- und Bauteilen zusätzlich zur Reduktion der Entwicklungszeit und Kosten bei.

Mikrocontroller werden heutzutage aus Gründen der Geschwindigkeit häufig in Assembler und nicht in einer Hochsprache programmiert. In Assembler erstellte Programme weisen meist eine höhere Codedichte auf, was einen geringeren Speicherbedarf und eine kürzere Ausführungszeit zur Folge hat. Auf der anderen Seite bietet eine Hochsprache mehr Komfort und Effizienz beim Entwickeln von Programmen. Modulares Programmieren und Wiederverwendbarkeit von Programmteilen sind nur zwei Stichworte in diesem Zusammenhang.

µOberon ist ein Versuch, die Geschwindigkeit von Assembler mit der Modularität von Hochsprachen zu vereinen. Mit dem Inline-Assembler ist es möglich, neben den Anweisungen in der Hochsprache direkt auf Maschinenebene zu programmieren, wo alle Register des Mikrocontrollers direkt zugänglich sind.

Die Hauptschwierigkeit während der Entwicklung von µOberon lag darin, dass der Instruktionssatz der Mcs51-Mikrocontroller keine Befehle mit indirekter Adressierung mit Offset anbietet. Andere Crosscompiler berechnen daher die Zieladresse bei jedem Variablenzugriff, so dass wie gebräuchlich die lokalen Variablen auf dem Stack alloziert werden können. µOberon hingegen führt mit der statischen Speicherzuteilung einen neuen Ansatz ein. Die absoluten Adressen der lokalen Variablen werden hier statisch während dem Linken zugeteilt, was zu besserem Code führt, der sich selbst mit Assemblerprogrammen vergleichen lässt.

Abstract

Providing general purpose solutions, microcontrollers are becoming increasingly important in the world of electronic systems. Compared to mechanical or simple analog electrical control systems they dramatically improve functionality and reliability, while reducing size and cost. Furthermore the capability of reusing software and hardware components reduces overall design-in time and cost.

Nowadays microcontrollers usually are programmed in assembler and not in a high-level language for efficiency reasons. As a rule, programs written in assembler language have a higher code density, what results in a lower memory consumption and a shorter execution time. On the other side a high-level language provides more comfort and higher efficiency during program development. Modular programming and code reusing are only two points in that context.

µOberon is a an approach to unite the speed of assembler programs and the modularity of high-level programs. It allows high-level as well as low-level programming, i.e. all microcontroller resources are easily accessible. This implied the need of an inline assembler, and µOberon offers a comfortable embedding of assembler sections into the language.

The main problem while developing µOberon was the fact that Mcs51 microcontrollers don't provide indirect-offset addressing. Therefore other cross compilers calculate the effective address on each variable access, so local variables may be located on the stack and addressed relative to a frame pointer as usual. µOberon on the other hand introduces a new approach, the so-called Static Memory Assignment, where the absolute addresses of local variables are assigned statically while linking. This leads to better code that even competes with assembler programs.

Table of Contents

1 Introduction

1.1 Motivation

When programming microcontrollers in assembler language, first results may easily be obtained. In general the program works fine even if it is getting more and more complex. But if one has to add something or to make changes somewhat later on, the size of the source text reduces the overall view even if it is well documented, what is recommendable when writing assembler programs anyhow.

To counteract this unsatisfactory fact, high-level languages have been introduced. High-level languages not only are more intuitive and easier to get used to than assembler language, but also the modularity and therefore the possibility to divide a problem into several subproblems allows more efficient software engineering. Therefore, modern computers mostly are programmed in high-level languages.

The motivation for this work stems from the unsuitable fact that there is no simple but powerful development system for Mcs51 microcontrollers that allows high-level programming.

1.2 Goal

The advantages of programming processors in a high-level language are highly visible. It is easier to code than in assembler language, the error rate is lower, and it is simpler to reuse software components because it is modular, what speeds up any code development process considerably if there are libraries that may be reused.

A module in a modular language is a component that does a defined job. There are static data and code belonging to a module. Other modules importing that module, so-called clients, don't have to know the implementation, and the module on the other side doesn't have to know its clients; it is exchangeable therefore. This is known under the term *information hiding*. Later releases of the module, after changes of the underlying implementation, can be used without the need of rewriting or recompiling the clients, as far as the changes did not affect the common interface. On the other side the module doesn't have to know his clients because they communicate via that interface. So, the modularity reduces complexity; splitting of tasks between several modules is possible. Moreover it is conceivable that more than one programmer work at a sole project.

The main purpose of μOberon is the transfer of what is usual on modern computers to so-called single chip computers, the microcontrollers.

1.3 Used Terms

In order to reduce the interpretation scope, this section explains some of the fundamental terms used in this thesis. Note that the definitions do not lay claim to completeness.

Microcontrollers

A *computer* may be regarded as the assemblage of a processor, memory and input/output ports. Usually these are several physical devices mounted together. A microcontroller on the other side is a whole small computer system combined on one chip. There may be a

small RAM and/or ROM on the microcontroller chip, as well as peripheral functions like timer and counter hardware, interrupt logic and analog-to-digital converters. Therefore microcontrollers often are called one-chip microcomputers.

In today's day, microcontrollers can be found in nearly every electronic device like video recorders, phones, toys, model railway controls, washing machines and terminals, or in the wide field of car electronics like ABS, fuel injection, gearing control or airbag. This illustrates how different the tasks for microcontrollers may be. A microcontroller that controls a video recorder doesn't require as much memory as a controller for a terminal does, but instead of that there may be higher requirements for the real-time behaviour, for instance to control the read/write head.

The fact that the whole CPU and periphery is integrated on one chip reduces the power consumption. This may be an additional criteria that speaks for microcontrollers when designing battery powered systems. CMOS technology chips may consume less than 10 mA, moreover providing standby modes like idle or power-down mode in the Mcs51 family, that further reduce overall power consumption.

In short terms, the main advantages compared to discrete solutions are costs, reliability and space requirement.

Assembler

Microprocessors need their program code in a machine readable form, that is a sequence of bytes representing the instructions and their operands. Since for humans it is quite hard to code directly those numbers, one writes the program in a better readable assembler language, where the single instructions are represented by easily remembered mnemonics like MOV or ADD. A computer program, likewise called assembler, translates those symbols into the machine readable numbers.

Compiler

So-called high-level languages (HLL) introduce a higher level of abstraction for programming, being even more intuitive than the assembler mnemonics mentioned above, and it is much easier to write a program in a HLL like C or Oberon than in assembler language.

In general, a program called compiler transforms a program from one programming language into a semantically equivalent formulation in another language. In our case it reads the program source text, written in μOberon, and translates it into machine readable code, here the Mcs51 machine language. In fact, assemblers do quite the same. They also transform programs from the mnemonic notation to binary machine code. But assemblers are quite simple programs, e.g. they are executing a simple 1-to-1 translation from the mnemonics into the instruction opcodes. Compilers on the other side have to span the semantic gap between the HLL program and the machine code. An example:

```
Oberon:     sum := x + y

Assembler:  MOV A,x
            ADD A,y
            MOV sum,A
```

More of these code patterns can be found in the sections of chapter *Project μOberon*.

Development Systems

Microcontroller applications usually are cross-developed, i.e. the programmer writes and compiles the code on a host system that differs from the target system where the

application finally runs on. The development system on the host ordinarily provides an editor and a compiler, maybe a simulator. The final machine code, the linked modules, will then be loaded to the target system.

During development of a new application it is important that the modify-compile-load-execute cycle does not take too long. Although it is not the idea to develop programs by trial and error, there will always be a possibility for an error.

- One way to develop programs is to burn each version into an EPROM, plug the EPROM into the socket and then switch on the microcontroller system. If the program does not work correctly one has to erase the EPROM and restart the cycle.
- It is also possible to simulate the embedded system by a simulator running on the host. This doesn't require further hardware. But because embedded microcontroller systems usually are characterized by many connections to the environment, e.g. to latches, shift registers, multiplexers and so on, it may be hard to reflect this hardware by software in order to get satisfying results. Software simulators can hardly show problems in software-hardware communication, and timing problems aren't as trivial to be discovered by simulation as well. But for first tests and in order to understand the functions of the microcontroller and its instructions this may be a first approach.
- Another possibility is a so-called monitor program running on the target system. The monitor software serves the host-target communication on the microcontroller and loads and executes programs transmitted by the host. The machine code will be transmitted into the RAM of the target. After that, the monitor program gives control to the just loaded program. Only after the program has proven its correctness it will be burned into an EPROM, and the microcontroller system can be used autonomously.
- A further way to develop software for microcontrollers is the use of in-circuit emulators. This device is plugged into the target system instead of the controller chip. The whole program can be downloaded to the emulator, breakpoints may be set during runtime, and even step-for-step execution of the program is possible. Unfortunately these devices are quite expensive.
- Instead of simulating the whole controller hardware, it may be less expensive to use an EPROM emulator. This device pretends that an EPROM containing the microcontroller program is plugged into the microcontroller system. The host sends the data into the emulator RAM, and after a reset the microcontroller executes the program, believing that there is a true EPROM in the socket.

Even though the third solution with a monitor program seems to be the neatest one, there are several reasons why the monitor program on the target does not satisfy, especially in a Mcs51 environment:

- The monitor program requires memory. This will not be a problem on a larger computer system, but it may be necessary to deal economically with each byte on microcontroller systems with typically small memories.
- During the step from development to the autonomous system the monitor program will mostly be removed because it is not used any more and might cause some overhead if it still exists. This could be a further source of errors.
- Several microcontrollers, as the members of the Mcs51 family, require special data at fix addresses at the beginning or at the end of the code memory where the controller starts executing the program after a reset, or fix interrupt addresses; after the occurrence of an interrupt the program flow branches to a specific interrupt address or reads a vector at a predefined location. Since the monitor program is supposed to start after a reset, the ROM containing the monitor program has to be mapped to these locations. An

interrupt during execution of the program in development is first processed by the monitor and then handed over to the program. This causes an overhead.

- Finally, in order to allow downloading of code, the code bytes have to be written somewhere into a random access memory. But Mcs51 systems, providing separate code and data memory spaces, usually find their programs in a read only memory space. To run programs located in the data memory, some external hardware is necessary; refer to Appendix B. This hardware becomes superfluous after the development process has terminated.

The idea of µOberon is to avoid unnecessary overheads and to have full control over the resources of the microcontroller. So, the way I chose was that the linker produces binary data that may directly be loaded into an EPROM emulator or burned into an EPROM.

1.4 Outline

Chapter *Mcs51 Microcontrollers* gives a brief introduction into Mcs51 microcontrollers with special concern about compiler construction. A more detailed presentation of the Mcs51 microcontroller family can be found in Appendix B.

Chapter *Project µOberon* outlines the thesis, introduces the components of µOberon and points out the relations between them. The specific sections deal with the hurdles that had to be taken during the evolution of µOberon, and illustrate the chosen solutions.

A rough explanation on how to use µOberon can be found in chapter *Using µOberon*. It treats all components like compiler, linker, loader, browser and decoder.

Finally, chapter *Conclusion* summarizes what has been achieved, gives an outlook onto some areas for future work, and concludes the thesis.

The appendices may be used as quick references while working with µOberon. They contain detailed information about the µOberon system as well as miscellaneous information compiled from the books listed in the *Bibliography*.

Appendix A: The Implemented Language µOberon lists the differences between µOberon and the standard Oberon [1] and may serve as an overview for programmers that already know Oberon.

Appendix B: Mcs51 Hardware Overview contains a detailed presentation of the target system. It may be used as a reference when accessing the controller's resources.

Appendix C: Mcs51 Instruction Set Summary lists the entire Mcs51 instruction set and explains the possible addressing modes. µOberon allows code inlining, so this list may be useful for low-level programming.

At the end, *Appendix D: File Formats*, *Appendix E: Data Structures and Module Interfaces*, *Appendix F: Bibliography* and *Appendix G: Index* complete the documentation.

2 Mcs51 Microcontrollers

This chapter gives a brief introduction into Mcs51 microcontrollers with special concern about compiler construction. A more detailed presentation of the Mcs51 microcontroller family can be found in Appendix B.

2.1 Overview

The members of the Mcs51 microcontroller family are 8 bit microcontrollers designed for general steering and control applications. The I/O ports are memory mapped using special function registers located in the internal RAM, which are also used for additional resources like timers, counters, interrupts and so on. Like most Intel processors, the Mcs51 uses the little endian format, so the least significant byte is at the lower address. However, as an exception, the most significant byte of an absolute address follows the instruction byte and the least significant byte is at the end, and hence at the higher address.

The memory model of the Mcs51 family is rather complex since there are three different memory spaces, and one of them is even divided into two sections. So, unlike other microcontrollers that provide a linear address space, Mcs51 controllers force the compiler to distinguish what kind of data is in which space. Both external memory spaces, the code and the data memory, use 16 bit addresses, whereas the internal memory may either consist of 128 or 256 bytes depending on the microcontroller type and is accessed with 8 bit addresses therefore.

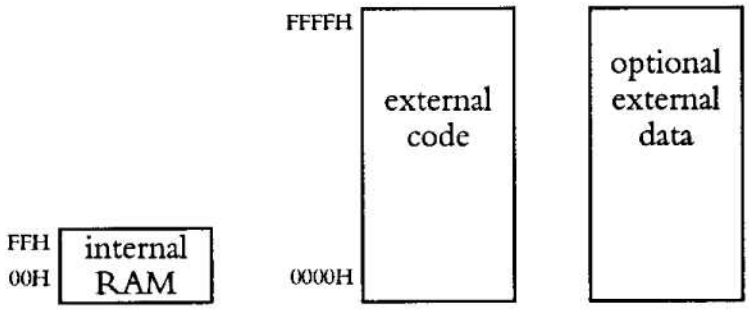

Figure 1 Memory spaces of Mcs51 microcontrollers.

There are some fix addresses in the external code memory. After reset, the microcontroller starts execution at address 0000H, and in case of an interrupt, it immediately branches to a predetermined address at the lower end of the code space. Note that on the external code memory space only read accesses are possible.

The internal RAM is divided into a lower and an upper half. The lower half contains the eight general purpose registers R0 to R7, and it is possible to switch among four register banks, so R0 to R7 may either be located at addresses 00H to 07H, 08H to 0FH, 10H to 17H, or 18H to 1FH. Addresses 20H to 2FH may be accessed bit by bit, providing 128 general purpose flags. Addresses 30H to 7FH are free and may be directly addressed.

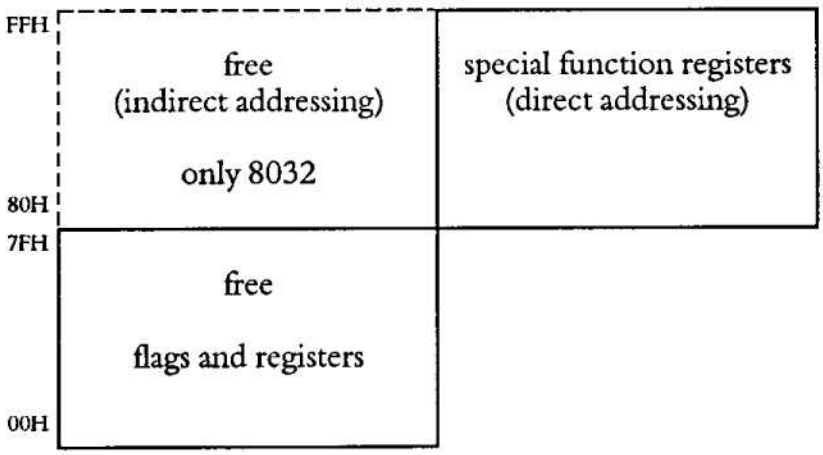

Figure 2 Internal memory organization of Mcs51 microcontrollers.

The upper half of the internal memory contains the mentioned special function registers that may also be addressed directly. But there are members of the Mcs51 family that provide 256 bytes of internal memory, and thus the upper half exists twice; when addressing indirectly, the upper 128 bytes are for free use, and direct addressing concerns the special function registers. Since stack operations use indirect addressing and the stack grows upward to higher addresses, models with 256 bytes of internal RAM allow deeper procedure nesting and recursion.

2.2 Programming Model

Beside a large number of special function registers, Mcs51 microcontrollers provide the following general purpose registers. Note that these registers, except R0 to R7, are located in the upper half of the internal RAM, as special function registers are.

Accumulator	A	8 bit	
B register	B	8 bit	
Registers	R0..R7	8 bit	x 4; four register banks
Stack pointer	SP	8 bit	
Program status word	PSW	8 bit	CY, AC, F0, RS1, RS0, OV, P
Program counter	PC	16 bit	
Data pointer	DPTR	16 bit	may also be accessed as 8 bit registers DPL, DPH

The accumulator is used for arithmetic and logical operations, for compares, and for data transfer from and to the external memory. The B register is an auxiliary arithmetic register for multiplications and divisions. Registers R0 to R7 may be used with several data transfer and compare instructions. R0 and R1 moreover may be used as 8 bit pointer registers for indirect addressing. By setting bits RS0 and RS1 in the program status word, one of the four register banks may be selected. This would allow fast context switches, e.g. in interrupt service routines. µOberon does not make use of that facility, but pushes all used registers onto the stack on the entry of an interrupt service procedure.

The program status word consists of several bits like the carry, an auxiliary carry, a general purpose flag F0, the register bank select bits, an overflow and a parity bit. µOberon uses F0 as sign flag for multiplications and divisions. The data pointer finally is necessary to address external memory locations.

The crucial fact is that the stack pointer is an eight bit register only and the stack is located in the internal RAM. This reduces the level of procedure nesting substantially since local variables are allocated on the stack to enable recursion.

2.3 Problems for Compiler Construction

Very non-orthogonal instruction set

This problem is common to 8 bit microcontrollers and implies some tricks to span the semantic gap between the high-level language and the instructions of the microcontroller. Sometimes it is not possible to avoid data transfer between the registers because some instructions expect their operands in particular registers. Likewise, registers R0 to R7 aren't real general purpose registers. For arithmetic and logical instructions as well as for external memory accesses, the accumulator is always one of the operands.

Only unsigned arithmetic operations

Because numbers in the high-level language usually are signed, the compiler has to insert sign checks for the operands, which blows up the code size. Especially divisions may imply large code sequences.

Various memory spaces

Since the memory space is not linearly addressable, the compiler has to distinguish what kind of data belongs to what memory space. µOberon uses the optional external RAM for dynamic memory allocation exclusively, global and local variables are located in the internal memory.

Interrupt vector table is located in the code memory space

This fact disallows dynamic installation of interrupt service procedures, i.e. they have to be linked to a specific interrupt on compilation time. Nevertheless it is possible to install them dynamically by using procedure variables, and the statically linked interrupt handler calls the current procedure.

No indirect-offset addressing

This was the main problem while developing µOberon. Local variables usually are located on the stack and addressed using a frame pointer. This substantial deficiency in the Mcs51 instruction set led to the innovation of µOberon, the so-called *Static Memory Assignment*. Refer to the corresponding section in the next chapter.

3 Project µOberon

This chapter outlines the thesis, introduces the components of µOberon and points out the relations between them. The specific sections deal with the hurdles that had to be taken during the evolution of µOberon, and illustrate the chosen solutions.

3.1 Aims

Before developing µOberon, the following aims had been defined:

- µOberon shall be a development system that allows full and easy access to all microcontroller resources, and implements efficiently a suitable subset of the vocabulary of the language Oberon. Its usefulness must be proven in practice.
- In contrast to existing Mcs51 compilers, µOberon shall also be able to generate code for systems without external data memory, and support multiple memory models like 128 or 256 bytes internal RAM and external RAM of any size. A large number of programs don't need a lot of variable memory space and a lot of applications may be made without external RAM. - Any external memory should be accessible nevertheless.
- A minimal runtime system for dynamic memory management and handling of runtime errors must be implemented. The programmer may define a procedure that will be activated in case of a runtime error. The number of the actual runtime error shall be passed as parameter.
- µOberon shall generate inline code for arithmetic functions.
- Built-in assembler. It shall be possible to write time-critical procedures in assembler language. The opcode notation of other Oberon implementations is only an interim solution. The ASM directive of µOberon may occur wherever a statement may be. Moreover it shall be possible that sections in assembler language may access µOberon variables.
- The linker shall produce a static core image that may directly be burned into an EPROM. Unlike standard Oberon systems, dynamic linking is impossible because the program is located in a read only memory.
- The SHORTINT basic type shall be implemented most efficiently since Mcs51 microcontrollers are 8 bit computers.
- The generated code shall be optimized with respect to execution time and shall make use of internal RAM economically. The generated code must compete with assembler programs. - Guideline: *„How would I solve the problem in assembler ?"*

The last point is an item to be discussed. [11] describes the implementation of an Oberon cross compiler for Motorola 68HC11 microcontrollers, where the generated code was optimized with respect to code length instead of execution time. The indicated reason was the fact that microcontrollers only provide limited memory resources. In my point of view the code memory, usually a ROM, is more than large enough in most cases - a program has to be pretty extensive to fill up even only a 4 kB ROM. But, especially when dealing with periodical interrupt procedures, the processor time is the crucial resource to be optimized. So, whenever there was a decision either to reduce code size or to reduce execution time, the latter way was taken.

3.2 T Diagrams

The development of compilers may be expressed using so-called T diagrams. A T diagram, as in figure 3 shows a translation process. Translating, another word for compiling, is the process of transforming a text from a source language into a target language. The program that does this translation is written in a third, not necessarily different language.

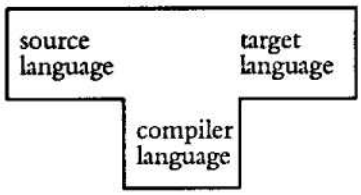

Figure 3 A T diagram.

Figure 4 shows the μOberon project. The most left T diagram represents the μOberon compiler written in Oberon, the subject of this thesis. In the middle there is the Oberon/F compiler that translates an Oberon program into machine executable code. At the right side finally there is the μOberon compiler running on Intel x86 processor or compatibles that produces code for Mcs51 microcontrollers.

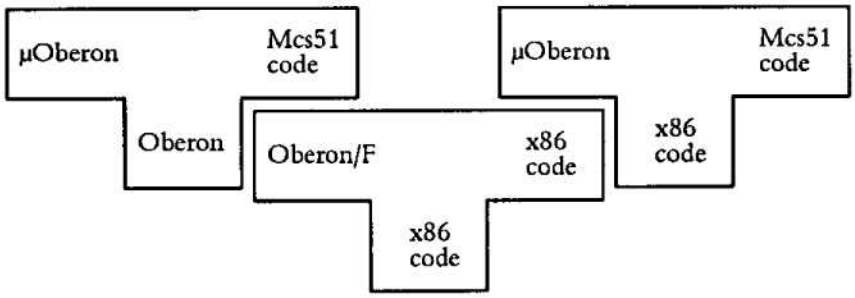

Figure 4 The steps of the μOberon project.

Cross compilers are programs that do not produce code for machines on which they are running. So, the right and the bottom part of the corresponding T diagram are not equal.

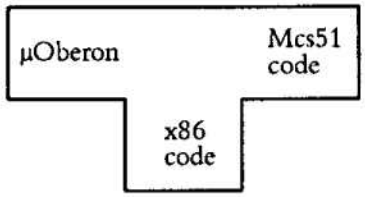

Figure 5 The μOberon cross compiler.

Note that the μOberon compiler source is written in standard Oberon-2 which is not compatible to the μOberon language. μOberon does not support program constructs like record extensions or open array parameters. For details refer to Appendix A.

3.3 Module Overview

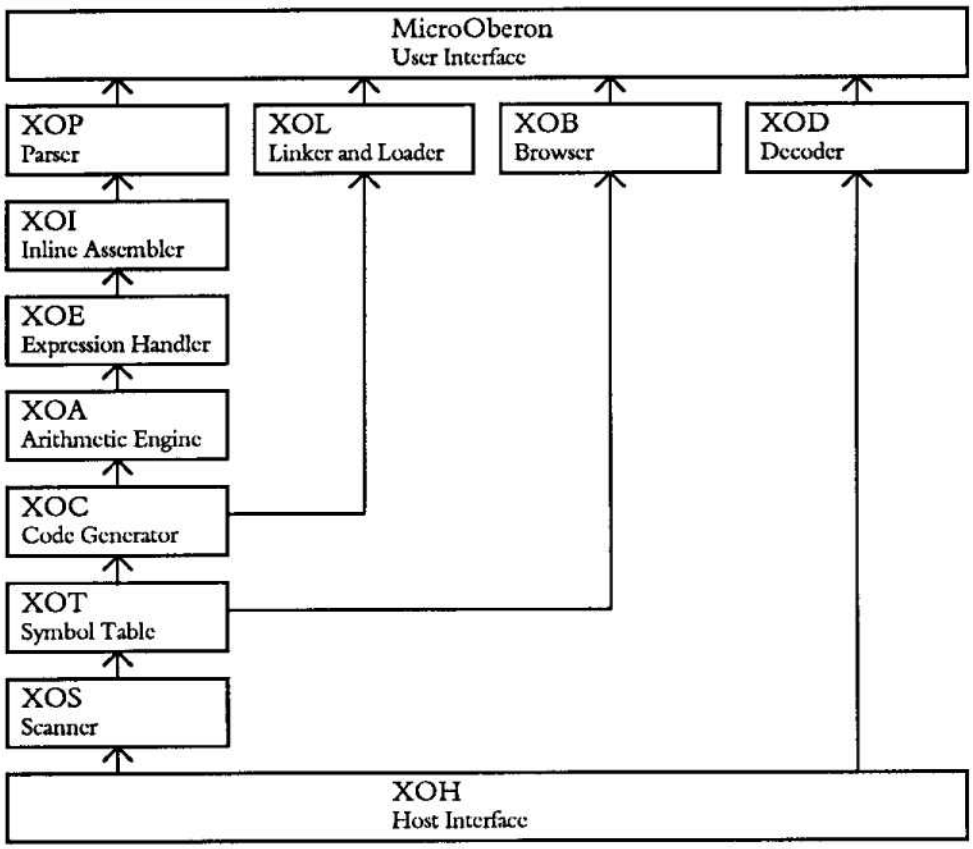

Figure 6 The module hierarchy of the μOberon system.

μOberon used the skeleton of the standard Oberon compiler frontend described in [1], completed with the Mcs51 backend. The modules of the whole application are shown in figure 6. This diagram is simplified by omitting direct import relationships if an indirect path also leads to the import. So, the parser for instance also imports the scanner or the host interface.

The host interfaces provides easy access to host resources like files or windows. The scanner translates the character stream into a token stream, the so-called lexical analysis, while the parser does a semantical analysis of that token stream. At the same time, since it is a one-pass compiler, the parser calls the procedures that generate the appropriate code. The symbol table manages the context information, e.g. the type of variables, and manages the symbol files. These three modules usually are called the frontend of the compiler, and they are independent of the target system. The code generator manages the memory and the stack of the target system and contains procedures to transfer memory, like loading a specific variable into a register. Moreover there are procedures for resolving fixup chains, for procedure calls and for generating the object file. The expression handler generates code for expressions, and the arithmetic engine generates code for single arithmetic operations and compares. The inline assembler module is used whenever inline code is to be generated, as it is in code procedures or assembler sections. Appendix A describes these two situations. The linker merges several object files from the compiler into a static core image that may be downloaded to the target system by the loader. The browser may be used for browsing the interface of a module, i.e. it decodes its symbol file. On the other side, the decoder reads object files or binary or hexadecimal linker output and shows the code in mnemonic notation.

3.4 Register Management

After a first glimpse at the programming model, one has the impression that the number of registers simplifies compiler construction. Compared to similar controllers like Motorola's 68HC11, it is true that the Mcs51 microcontroller family is well equipped. But due to the fact that they are something between register machines and accumulator machines, it is not trivial to set up a strategy for register managing. The following example may illustrate that point. In order to understand the example it is important to know the technique of delayed code generation; see [1] or [10].

```
a := b + c + d*e;
```

First the addition of variables b and c is executed and the result is in the accumulator. It would be wise to set the mode of the item containing the result to A, indicating that its content is stored in the accumulator. Then the term d*e is parsed. Since the MUL instruction expects the operands in the accumulator and the B register, and the accumulator is used at that time, the content of the result item of the first addition must be saved. There are three different strategies to do that:

- The first idea would be to store it in a general purpose register R0..R7. But this is not as easy as it sounds since that item is a local variable of the expression procedure, and the term procedure may not change the local variables of another procedure.
- The content of the accumulator could be pushed onto the stack. This way would lead to something between expression evaluation on the stack and in registers, and it is not wise to push too much onto the stack because of its limited size.
- Finally, and that is how the current version of µOberon solves the problem, it is possible to demand that the accumulator and the B register must always be free, i.e. the result of an operation must be stored in a general purpose register compellingly. So, e.g. after each addition the content of the accumulator is stored in a newly allocated register R0..R7.

There would be a possibility to enable the first way: A facility that manages all items that exist at the time being. So, if a procedure needs the accumulator, it announces its interest to that manager which saves the accumulator in a general purpose register if it was used. This *dynamic item manager* would be an extension for later releases of µOberon.

The main problem with the chosen solution of the current version is the following unsatisfying situation:

```
x := y + z;
```

would be compiled to

```
MOV A,y
ADD A,z
MOV R7,A
MOV x,R7
```

because the result of the addition must be stored in a general purpose register so the accumulator is released. In order to avoid the unnecessary use of R7 in the last operation, a peephole optimization has been implemented that reduces the last two instructions to

```
MOV x,A
```

This again is not fully satisfying because the register R7 that now is superfluous is allocated prior to the optimization and won't be free for further use in complex expressions therefore, i.e. one register is lost. But since the register allocation procedure first returns R7, then R6 and so on, the lost register is R0 that may be used as a pointer register.

3.5 Sets

Variables of the basic data type SET may contain elements numbered from 0 to MAX(SET). Since Mcs51 microcontrollers have 8 bit processors, the upper bound is 7, i.e. a set is a single byte. Operations on sets are efficient since they are translated into a single assembler instruction in most cases. However, the initialization of SET variables is not trivial. Consider the following assignment:

```
s := { b1, b2, b3 }
```

As long as the element are constants, the value is calculated on compilation time. But if they are variables, the compiler has to generate code to calculate the value. In the example above, s will be the result of

```
{ b1 } + { b2 } + { b3 }
```

so the single bits are calculated individually and the OR operator is applied. The calculation of a single bit is done by the following code sequence:

```
        CLR A
        SETB C          ; that bit will be rotated into the accumulator
        INC Rn          ; Rn contains the number of the bit
L0:     RLC A           ; rotate
        DJNZ Rn,L0      ; decrement and jump if not zero
                        ; the value is now in the accumulator
```

This sequence might look too extensive, but the Mcs51 instruction set disables any shorter sequence since there is no compare and jump if equal (CJE) instruction, or the rotation is only possible on the accumulator, which disables its use as loop counter. So, the accumulator must be rotated at least once even with bit number 0.

The situation even gets worse when not only single bits are defined, but also a range of bits. As long as the bounds of the range are constants, the calculation may be done at compilation time again, but if one bound or even both are variables, the generated code may become quite large. In general, the compiler calculates a range as follows:

```
val1 := LeftShift(-1, lower);
val2 := LeftShift(-2, upper);
value := val1 - val2
```

In val1, all bits are set except bit 0 to lower-1, and in val2, all bits are set except bits 0 to upper. So, the subtraction results in the desired value where bits lower to upper are set. If the bounds are variable, the compiler has to generate code that calculates val1 and/or val2 respectively. The missing shift instruction is simulated by rotate through carry (RLC), while the carry bit is reset prior to the rotation. Refer to the documented source text of module XOE for details.

Constant Table

Instead of shifting the bit to the right position, a constant table containing the eight powers of 2 would allow a faster generation of set values. The instruction

```
MOVC A,@A+DPTR
```

allows fast access to that table. It is worth to mention that the accumulator first should be masked by

```
ANL A,#7
```

so the MOVC instruction does not exceed the upper bound of the table. In order to get the values for a range of bits, the accumulator is incremented prior to the MOVC instruction, and decremented afterwards. So, bits 0 to the initial value are set.

3.6 Numbers

Mcs51 microcontrollers exclusively support unsigned byte numbers, i.e. from 0H to 0FFH. But high-level languages also provide signed numbers and larger ranges, so a compiler has to emulate the missing instructions. µOberon offers the basic signed integer types SHORTINT, INTEGER and LONGINT, which take 1, 2 or 4 bytes respectively. Arithmetic with real numbers is not possible with the current version, but 4 and 8 bytes IEEE real numbers arithmetic is planned. Most algorithms for arithmetic operations on numbers stem from [13].

µOberon does not generate overflow checks for efficiency reasons. So, the addition

```
VAR s1, s2, s3: SHORTINT;

s2 := 60;
s3 := 70;
s1 := s2 + s3
```

would assign 130 to the short integer s1, but 130 will be interpreted as -126 since short integers are signed. It is the programmer's response that the result of any calculation lies within the range of the actual type.

Compares

These overflows may be a real problem when comparing two numbers. A compare usually is implemented by a subtract instruction, and the carry bit indicates whether the first or the second number was greater. But subtractions inherently imply the possibility of overflows, what may lead to a wrong result when dealing with signed numbers:

-10	<	+120	subtract
-130	<	0	-130 is not in the range of SHORTINT
+2	<	0	error; misinterpretation

In µOberon, this problem is solved as follows: Prior to compare two signed numbers, their sign bit is inverted. This simple trick maps the positive numbers above the negative numbers, so that they may be interpreted as unsigned:

	before inverting the sign bit	after inverting	
127	0111'1111	1111'1111	255
126	0111'1110	1111'1110	254
1	0000'0001	1000'0001	129
0	0000'0000	1000'0000	128
-1	1111'1111	0111'1111	127
-127	1000'0001	0000'0001	1
-128	1000'0000	0000'0000	0

So the inversion of the sign bit will lead to the correct result. Of course the inversion of the sign bit has to be redone at the end when directly operating with registers that will be reused. As mentioned above, the accumulator is supposed to be free at any time, so if one operand has been loaded into the accumulator, its sign bit needs not to be reinverted. Only the sign bits of operands in the general purpose registers R0..R7 have to be reinverted.

All compares are done with the CJNE instruction: Compare and jump if not equal. This instruction affects the carry bit, which may be used to determine which operand was greater:

```
CJNE  A , B

          ↓
      ≥  if carry = 0. But if A=B, CJNE would have branched. So A > B.
      <  if carry = 1
```

There are four different addressing modes for the CJNE instruction, all using three code bytes and taking two clock cycles:

```
CJNE A,direct,rel
CJNE A,#data,rel
CJNE Rn,#data,rel
CJNE @Ri,#data,rel
```

The μOberon compiler uses the adequate addressing mode and avoids unnecessary transfers into registers. So, when both operands are variables, the first addressing mode is chosen and one operand has to be loaded into the accumulator. If the other operand is a variable that has to be accessed indirectly like intermediate variables, it will be loaded into the B register. If one operand is a constant, one out of the other three addressing modes is used. It may be that the other operand already is in a register, so the third addressing mode is used, or the fourth mode if the other operand has to be accessed indirectly. Otherwise, the operand that is not constant has to be loaded into the accumulator and the second addressing mode is taken.

3.7 Fixup Chains

One of the goals of μOberon was to implement the SHORTINT type most efficiently, and conditional expressions only consisting of SHORTINT compares must be rather complex in order to break the bounds of the limited range of relative jumps where the target address must be in the range of -128 to 127 relative to the address of the next

instruction. Following the mentioned guideline *„How would I solve the problem in assembler"*, relative 8 bit branches are used for TRUE and FALSE jumps. This avoids unnecessary insertions of LJMP instructions with 16 bit target addresses, since the conditional branch instructions are part of 8 bit fixup chains. So µOberon supports two fixup chains, namely 8 bit and 16 bit. 8 bit fixup chains are used for TRUE and FALSE jumps within expressions, and 16 bit chains are used on statement level, i.e. for while loops. Certainly, backward jumps always use the branch instruction that is appropriate to the known distance, i.e. short (relative) or long (absolute) jumps.

16 bit fixup chains are implemented as usual, i.e. the target address of the first forward jump is 0000H, and the target addresses of further forward jumps to the same label point to the previous address belonging to that fixup chain. So, after the effective target address has become known, the compiler follows the fixup chain and inserts that address on each location.

On 8 bit fixup chains, the relative target address is only an 8 bit value, so it is not possible to encode the whole address of the previous member of the chain. Instead, the inserted address is a negative number pointing to the previous location, relative to the current position. And again, a zero marks the end of the chain.

3.8 CASE Statement

The only exception where relative jumps are used on statement level is within CASE statements. CASE is a redundant statement that was introduced to speed up the selection of a statement sequence according to the value of an expression. In contrast to the IF statement where the expressions are evaluated on each ELSIF branch, the CASE statement evaluates the expression only once, and compares its value with a set of constants. However, the statement sequences belonging to the single cases usually are quite short, and in order to enable the use of the compare and jump if not equal (CJNE) instruction, the jumps over those statement sequences are relative jumps with 8 bit addresses. Note that the jumps to the end of the case statement are LJMP instructions. So, the statement

```
CASE expr OF
Value0:
   StatementSequence0
| Value1:
   StatementSequence1
| ...
END
```

is compiled into

```
        MOV A,expr              ; evaluate expression
        CJNE A,#Value0,L1
           StatementSequence0
        LJMP End
L1:     CJNE A,#Value1,L2
           StatementSequence1
        LJMP End
L2:     ...
Ln:     Trap(4)                 ; Invalid case in CASE statement
                                ; if there is no ELSE branch
End:    ...
```

It is important to know that the expression of a CASE statement must be of a one byte type, i.e. CHAR or SHORTINT. Its value is loaded into the accumulator, and

```
CJNE A,#data,rel
```

is used for the compares. If the last instruction was a transfer from the accumulator into a general purpose register as it is done with expressions a priori, see *Register Management*, this transfer instruction is canceled, and the result is taken directly from the accumulator.

However, if the type of the expression is SHORTINT, the most significant bit is inverted; see *Compares*. This is necessary if a case label specifies a range, e.g. 10H..20H, because the value is compared against the bounds with the following instruction sequence:

```
        CJNE A,#from,L0
        SJMP Ok                 ; A is equal to lower bound
L0:     JC False                ; A is less than lower bound
        CJNE A,#to,L1
        SJMP Ok                 ; A is equal to upper bound
L1:     JNC False               ; A is greater than upper bound
Ok:       StatementSequence
        LJMP End
False:  ...
```

If from+1 = to, the accumulator is compared with both values directly, and no range check is inserted. This only takes 10 code bytes and 8 cycles instead of 14 bytes and 12 cycles with a range check.

```
        CJNE A,#from,L0
        SJMP Ok
L0:     CJNE A,#to,False
Ok:       StatementSequence
        LJMP End
False:  ...
```

Jump Table

Another idea would have been a jump table for CASE statements. This would have been a further decision in favour of the execution time and to the disadvantage of code size.

```
        MOV DPTR,#Table
        MOV A,offset            ; offset is the number of the case
        RL A                    ; AJMP instructions take two code bytes
        JMP @A+DPTR             ; jump to [ACC] + [DPTR]
Table:  AJMP Case0              ; jump table
        AJMP Case1
        AJMP Case2
        ...
Case0:    StatementSequence0
        LJMP End
Case1:    StatementSequence1
        LJMP End
        ...
```

But the use of AJMP instructions may lead to problems since the target address must lie within the same 2 kB code segment, and only the linker knows in which segments the code will be. Moreover the rotation of the accumulator reduces the range, i.e. only values from 0 to 127 are possible instead of 0..255.

3.9 Dynamic Memory

As mentioned above, the external RAM is exclusively used for dynamic memory allocation. The heap is implemented rather simple but efficient: Two bytes above the first register bank, addresses 08H and 09H, are used as heap pointer if external RAM is present; this 16 bit pointer always points to the next free byte. The NEW standard procedure increments that pointer by the size of the allocated object, and compares the new size with the upper bound of the external RAM. In fact this implements a heap with a block size of one byte.

The external data memory is an own data space. Unlike other systems where address 0000H is reserved for special purposes, the external data of Mcs51 microcontrollers may start at this address. So the convention that NIL is a synonym for address 0000H is not possible here. µOberon uses the value 0FFFFH for NIL.

3.10 Procedure Variables

Mcs51 microcontrollers require constant target addresses for jump and call instructions. But the nature of procedure variables requires variable targets, so the following trick was necessary:

```
        PUSH return address     ; first push low byte, then high byte
        PUSH target address     ; value of the procedure variable
        RET                     ; misuse
return: ...
```

The RET instruction takes the target address from the stack and branches. The return instruction of the called procedure takes the previously pushed return address as target address, and the program execution continues after the misused RET.

When a procedure is assigned to a procedure variable, its 16 bit address is stored in the two bytes that belong to the variable. But the final address of the procedure within the static core image is not yet known, moreover it should be possible to assign procedure to procedure variables where only the forward declaration is known, and the assignment must be fixed up later on after the declaration of the procedure. But all fix up mechanisms require 16 bit addresses where the low byte immediately follows the high byte, whereas the assignment of 16 bit values is of the form

```
MOV Var_l,#address_l
MOV Var_h,#address_h
```

and it is not possible to use the common fix up mechanisms. One way would have been to write new procedures that handle this situation, and extend the object file. But in order not to overload the compiler another way has been chosen:

```
MOV DPTR,#address   ; so the low byte is immediately after the high byte
MOV Var_l,DPL
MOV Var_h,DPH
```

3.11 Inline Assembler

As stated above, applications for embedded systems inherently are more hardware oriented than other, portable programs. So the need for an inline assembler is highly visible, e.g. for procedures accessing on-chip or external hardware. µOberon is more than a cross compiler for Mcs51 microcontrollers, so one of the main focal points of this thesis was a comfortable embedding of an inline assembler into the language Oberon.

Assembler sections may occur wherever ordinary Oberon statements may be. Any assembler mnemonics and their operands written in the common Intel assembler notation may be inlined enclosed by keywords ASM and END. Furthermore, code procedures and even code module bodies may be defined using ASM instead of BEGIN. Refer to Appendix A for more details.

Branch instructions may contain labels as target addresses. As mentioned above, µOberon supports both fixup chains, namely relative (8 bit) and absolute (16 bit) addresses. If the target of an absolute jump instruction is a label, it is automatically inserted into the fixup list so the linker adds the module code offset to the label. On the other side, a constant number as target address will not be affected by the linker.

3.12 Static Memory Assignment

The problem with the missing indirect-offset addressing mode has been mentioned. One possible solution would have been to calculate the effective address and then to access indirectly on each access of a variable, i.e. to simulate an access of the form off[base] by [base+off]. But this would have been very inefficient:

```
MOV A,base
ADD A,#off
MOV Ri,A      ; Ri = R0 | R1
MOV A,@Ri
```

which takes 6 code bytes and 4 cycles; 4 µs using a 12 MHz crystal. Note that this sequence is necessary on each variable or parameter byte access. Beside that, a pointer register has to be free on each access, and since there are only two of them, R0 and R1, one should deal economically with that resource. So, following the mentioned guideline of µOberon, to search for the ideas in assembler programming, a different solution was chosen.

The principle of Static Memory Assignment bases on a constant frame pointer. So, absolute addresses for local variables are assigned statically while compiling or linking, and procedure activation frames are not located on the stack therefore. Global and local variables are assigned statically to the memory in order to be able to make use of direct addressing mode. All local variables share the same memory addresses and the procedure prologue pushes the memory spaces that will be used for the local variables onto the stack. So, within the procedure the addresses of all global and local variables are known during compilation.

Organization of the Internal Memory

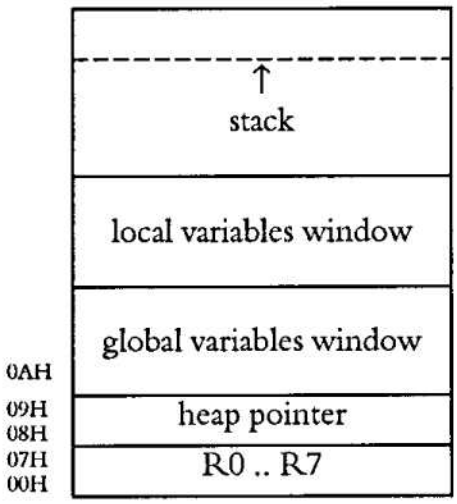

Figure 7 The organization of the internal memory.

Registers R0 to R7 are located at addresses 00H to 07H, so default register bank 0 is always used. At 08H to 09H, with the least significant byte at the lower address, there is the heap pointer that points to the next free byte in the heap in the external memory, i.e. there is a heap with a block size of one byte. The linker generates code that initializes this pointer corresponding to the selected memory model. Note that the heap pointer only exists if external RAM is present.

Starting at address 0AH are the global variables. Note that the compiler starts at address 0H for the global variables, and the linker fixes up these addresses so that all modules have their own space for global variables. Therefore the size of the global variables window is determined by the linker. The linker also sets the stack pointer above the local variables window. The easiest way would be to take the size of local variables of that procedure with the most local variables space as the local variables window. But as explained later, because of the intermediate variables in nested procedures, the user has to define the maximum size of local variables.

Parameter Passing

Since parameters should be as easily accessible as local variables are, namely directly addressed, the callee copies the parameter from the stack into the local variables space. The following sequence is necessary:

```
MOV Ri,SP
DEC Ri          ; jump over the return address

DEC Ri          ; copy parameters after saving local variables space
MOV loc_n,@Ri   ; for each parameter byte
```

This sequence initially takes 4 code bytes and 4 cycles plus 3 code bytes and 3 cycles for each parameter byte instead of 6 code bytes and 4 cycles on each variable access. - It is not advisable to misuse the POP instruction after changing the stack pointer for the memory transfer because of interrupt procedures that may occur when the stack pointer is changed.

Call-by-reference parameters are somewhat more difficult to implement. It is not possible to pass the address of the variable parameter because the parameter will be pushed

onto the stack and therefore the address changes. Earlier versions of μOberon implemented the copy-restore approach where variable parameters are passed as if they were call-by-value parameters, and restored by the calling procedure after procedure execution. But now the caller calculates the address of the parameter after the push operation, i.e. the offset is added to the current stack pointer value and that address is pushed onto the stack.

Calling Conventions

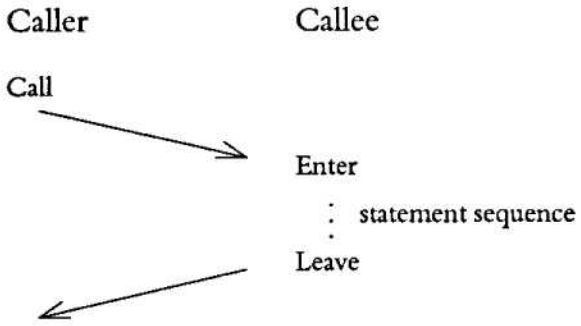

Figure 8 Procedure calls.

Call

The caller pushes all parameters onto the stack, and executes the procedure call with a LCALL instruction. Figure 9 shows the internal memory after the call.

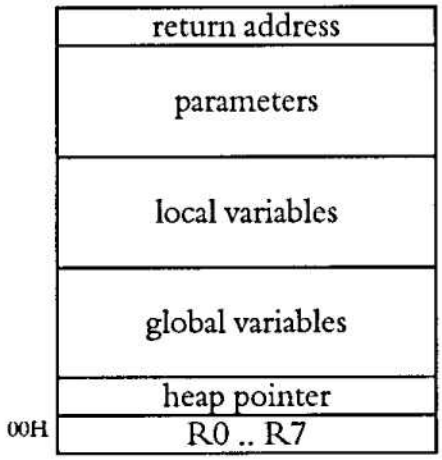

Figure 9 The internal memory after the call.

Enter (procedure prologue)

First the current stack pointer has to be stored in a pointer register, in R0 or R1. Since all registers are free on a procedure entry, there will always be a pointer register available.

After that the local variables space must be saved:

- Global procedures:
  Callee-saved parameters and local variables. The callee pushes the amount of bytes onto the stack that he will use for his parameters and local variables.

- Local or nested procedures:
  Callee-saved local variables window plus static link. The callee pushes the whole local variables window onto the stack and the address of the local variables of the surrounding procedure on the stack.

- Interrupt procedures that must be global:
  Callee-saved parameters and local variables plus accumulator A, B register, program status word PSW including user flag F0, R0..R7, and data pointer DPTR if external RAM is present. These registers may be changed during interrupt service and must be saved therefore.

This is the reason why the user has to define the size of the local variables window. The compiler has to know how many bytes have to be pushed onto the stack in the prologue of a nested procedure. The reason why nested procedures must save the whole local variable window is that all variables of the caller must be on the stack, especially when the nested procedure has a smaller local variable space than the surrounding caller.

Finally the parameters on the stack must be copied into the local variables space using the pointer register. So, the procedure prologue code looks as follows:

```
    MOV R0,SP          ; R0 points to the last byte pushed onto the stack
                       ; here: the MSB of the return address

IF (XOC.level > 1):    ; static link only necessary if it is a nested procedure
    MOV loc0,SP        ; in that case, loc0 is the static link
    INC loc0

    PUSH all locals

    DEC R0             ; R0 now points to the LSB
                       ; and to the last byte of the parameters
                       ; after the next decrement
    DEC R0             ; for all parameter bytes:
    MOV locn,@R0       ; copy parameter
```

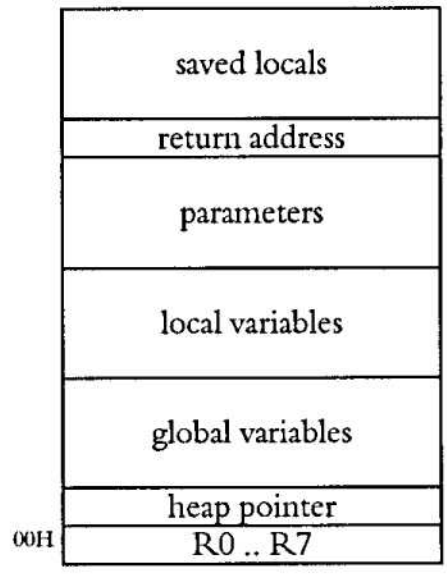

Figure 10 The internal memory after the procedure prologue.

Actually, procedures called from the module body don't have to save the local variables because in that case that space is not used prior to the call. But due to the fact that the local variable space is callee-saved, this situation is difficult to implement, i.e. it is not possible to detect this at compilation time except with a hidden parameter indicating if called from the module body. But since module bodies are executed only once on initialization it is not necessary to optimize that. It is not disastrous if the initialization sequence takes some microseconds more or less; the realtime behaviour won't be affected. Moreover the stack is empty at that time, so it won't aggravate the problem of the small stack size.

Leave (procedure epilogue)

The callee must copy back all VAR-parameters, and must restore the pushed local variables.

The code sequence of the procedure epilogue is somewhat larger than usual. If there is a RETURN statement, a new fixup chain is generated, and the program execution branches to the only procedure epilogue code sequence.

After the call

The caller decrements the stack pointer by the size of the parameters, while restoring all VAR-parameters. Adjusting the stack pointer is done by the following code sequence:

```
MOV A,SP
CLR C           ; Carry must be cleared prior to SUBB
SUBB A,#pSize   ; subtract the parameter size
MOV SP,A        ; write back
```

This sequence takes 7 code bytes and 4 cycles. If the size of the parameter is less or equal than 4 bytes, this is done directly by DEC SP.

Return Value

The return value of a function procedure is passed in R2 for CHAR, SHORTINT and SET values, R2+R3 for INTEGER or POINTER values, R2+R3+R4+R5 for LONGINT values. On a function call, these registers must be free. Otherwise the error *not enough registers: simplify expression* is indicated. If the compiler allocates a register, first R7 is returned, then R6, and so on. Remember that R0 and R1 are the only registers usable as pointer registers and should not be used as general purpose registers if possible. So, R2 is taken for the function return values.

Consequences

Compared to ordinary compilers that only have to increment the stack pointer and set the frame pointer on a procedure entry, static memory assignment implies less efficient procedure calls. So it is recommendable to use local variables only if the procedure may be called recursively, and to prefer global variables otherwise.

Restrictions

Using static memory assignment, it is difficult or even impossible to implement open array parameters and record extensions because the size of all parameters must be known at compilation time.

3.13 The Linker

Once the compiler has generated the necessary object files, the linker must merge them together. In contrast to the standard Oberon system, the Mcs51 linker plays a more important role. In common Oberon environments, a linking loader resolves references to other modules dynamically while loading a particular module. The Mcs51 linker statically generates a core image of all modules, and moreover even generates code for the reset and interrupt sequence. The reset sequence for example has to set the heap pointer appropriately if external RAM is present, and to initialize the stack pointer.

As described, Mcs51 microcontrollers read the instruction located at address 0 after reset, and in case of an interrupt, they branch to particular addresses and continue the execution after having pushed the return address onto the stack. Since interrupt service procedures may be defined in various modules, it is the task of the linker to generate the jumps to those procedures and to check that at most one service procedure is defined for each interrupt.

The common linker tasks include the recursive loading of all modules and checking that each module is loaded only once, and checking the keys of the modules to reveal any import inconsistencies. While loading, the μOberon linker compares the options in the object files, e.g. the internal and external RAM settings for the target system as well as the size of the local variables window. In contrast to other compiler options like stack overflow or array index checking, these settings must correspond.

After having loaded all imported modules, the linker resolves all open references; it *links* them together. Moreover it fixes the variable addresses that are known now, e.g. the linker decides where the global variables of each module are located. The local variables window is located above the global variables, so after the global variables are processed, the linker will increment the addresses by the local variable base at each local variable access. The object file contains information about all global and local variable accesses. The absolute 16 bit addresses have to be incremented by the code base address of the module as well. So there is also a list of all 16 bit addresses in the object file.

Finally the linker generates a chain of all module bodies that have to be executed prior to the execution of the command. Here all module bodies, namely the bodies of the imported modules as well as the bodies of the additionally indicated modules, are chained.

The following example shall illustrate the creation of a core image:

```
MODULE Lib;                 MODULE Main;

...                         IMPORT Lib;

BEGIN ...                   PROCEDURE Do*;
END Lib.                    BEGIN ...
                            END Do;

                            BEGIN ...
                            END Main.
```

Now the linker is started with parameter

```
Main.Do
```

and the core image showed in figure 11 will be generated.

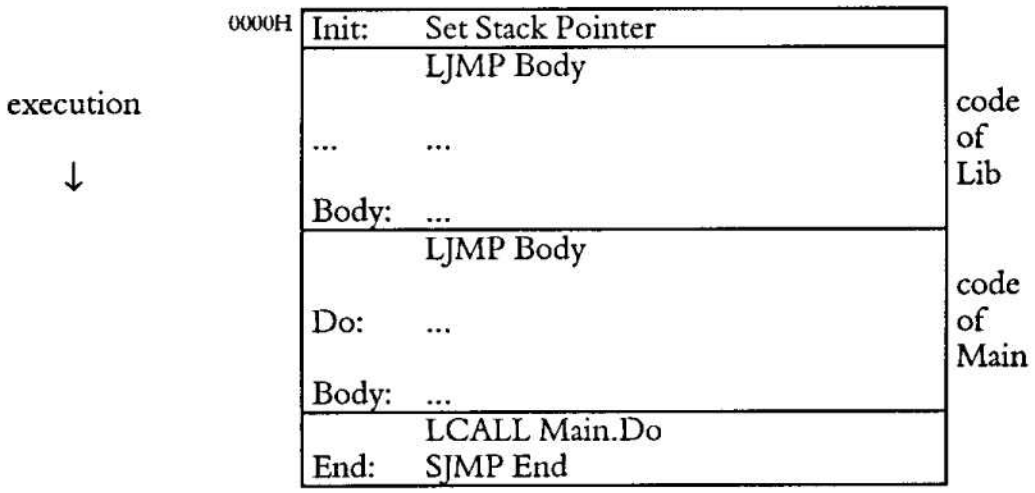

Figure 11 The core image generated by the linker.

This is the simplest case where no interrupt service procedures and no trap procedure are defined, and no external RAM is available. The command is called at the end of the module bodies chain, and after its termination, an endless loop lets the microcontroller stay in a defined state.

If interrupts are defined, LJMP instructions to the service procedures are inserted at the corresponding addresses, and a SJMP or LJMP instruction jumps from address 0000H, where the microcontroller starts execution after reset, over those instructions to Init. If external RAM is present, the heap pointer has to be set. Finally, if a trap procedure is defined, as explained later in Appendix A, the linker inserts a call to that procedure prior to the endless loop. So, if there are interrupt service procedures and a trap procedure and external RAM, the extended core image looks as in figure 12.

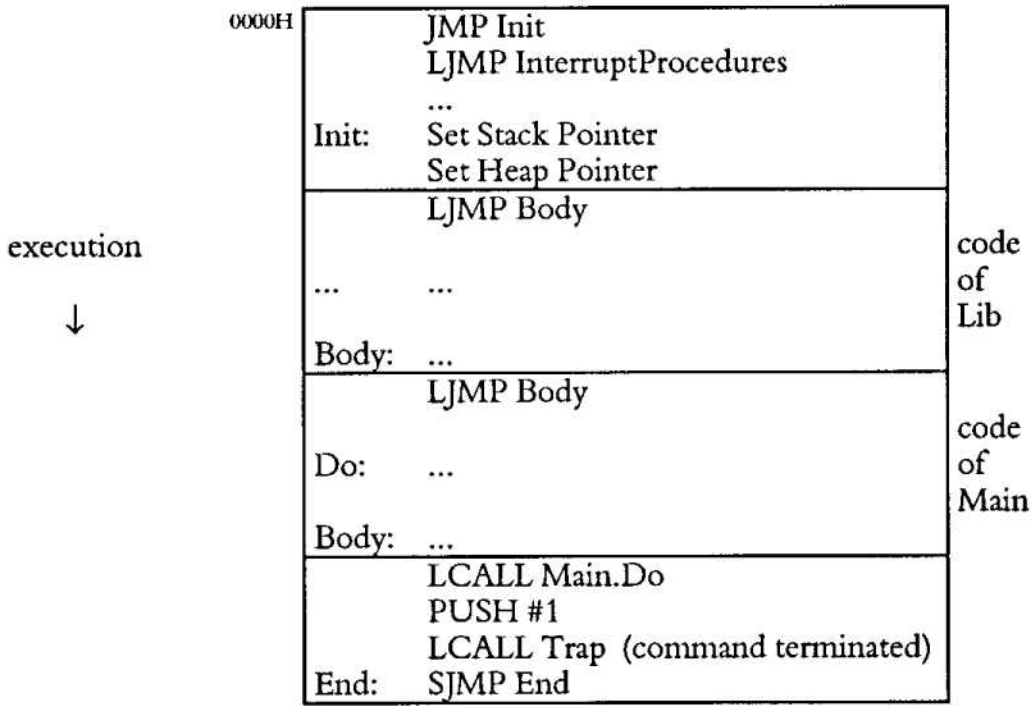

Figure 12 The extended core image generated by the linker.

Beside the simple binary output format where the bytes to be burned into the EPROM are ordered sequentially without any further information, the linker provides an alternative output format that is accepted by a large number of EPROM burner programs, namely the Intel-Hex format. Refer to Appendix D for a description of that format.

3.14 The Loader

Operating systems provide a high level of abstraction for the interfaces of the host computer. This simplifies development of programs that wish e.g. to print the content of a window. So, the programmer does not have to know specific details about connected printers since this is a task of the operating system.

But this was a problem while developing the µOberon loader that sends the code to be downloaded to the parallel interface byte by byte. It was not possible to make out a simple I/O procedure in the Oberon/F system that allows to send a single byte. So two code procedures were written that directly call DOS interrupt 17H to do this task. This way it was also possible to read the current state of the parallel port, i.e. if the port is ready or not. The current version of the µOberon loader only supports parallel interfaces LPT1 to LPT4. Other interfaces may be added by writing new I/O procedures. When porting µOberon to other operating systems where Oberon/F is available, these procedures in the host module XOH have to be rewritten.

4 Using µOberon

A rough explanation on how to use µOberon can be found in this chapter. It treats all components like compiler, linker, loader, browser and decoder.

4.1 Development Process

The compiler reads its input out of the current window and generates an object file with extension *obj* containing the code of the module plus some additional information, and a symbol file with extension *sym* that defines the interface of the module. The linker reads all necessary object file and creates either a binary output file with extension *bin* or an Intel-Hex file with extension *hex*, as selected in the µOberon options panel. The loader finally accepts those binary or Intel-Hex files to download to the target system. The browser may be used for decoding symbol files, i.e. for browsing in interfaces, and the decoder displays the contents of object files and binary or Intel-Hex core image files.

Messages

All µOberon components output status and information messages. After successful compilation, after decoding an object file and after linking, three numbers are written to the log window:

xcompiling "name" n1 n2 n3

where n1 is the code size, n2 the size of the module's global data, and n3 the maximum size of the local data; all values are measured in bytes. The last number may be useful to determine the size of the local variables window. Note that if there are nested procedures, the last number is equal to the size of the local variables window. After downloading and after decoding a core image, the code size is written to the log.

4.2 The µOberon Control Panel

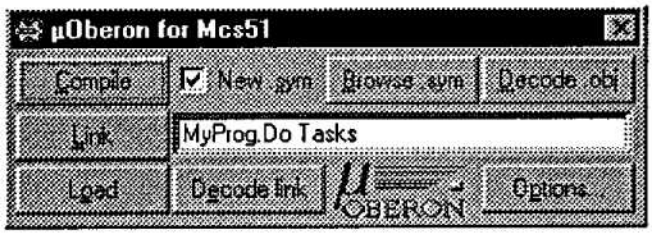

Figure 13 The µOberon control panel.

This panel provides access to all components and should always remain open while working with µOberon. The underlined letters represent the shortcut access, so the compilation process may for instance be evoked by pressing Alt-C. The compile button may only be activated if the current window is a text window, from where the compiler gets its input.

The check box *New .sym* indicates whether the creation of a new symbol file is allowed or not. As in every Oberon implementation, this must be enabled explicitly, otherwise the compiler marks an error if the interface has changed. To browse interfaces of existing modules, use the *Browse .sym* button. This opens a file selector box where the desired

symbol file may be chosen. The information in the generated object files on the other side may be displayed by pushing the *Decode .obj* button. The object file chosen in the file selector box will be displayed in a separate window.

Object files must be linked. The *Link* button links all object files used by the indicated command written in the text box, plus additional object files. The text in the box must be of the form

```
LinkParameter = Module "." Command {Module}.
```

The supplementary modules to be linked may be modules which install periodic interrupts or special tasks in their bodies and which are not directly or indirectly imported by the main module. However, after the linking process has terminated, the memory map of the just generated core image is written into the log window. The three sections inform about the division of the three memory spaces, namely the code memory space, the external and the internal RAM.

The core image may be loaded to the target system by pushing the *Load* button. Note that the download is executed only if the corresponding interface is ready.

Besides the linker output, i.e. the core image may be decoded as well. The *Decode link* button opens an new window with the code of the just selected file selected.

The *Options...* button finally opens the options panel which is subject of the next section, and additional information about µOberon finally can be retrieved by pushing the small button near the µOberon logo.

4.3 The µOberon Options Panel

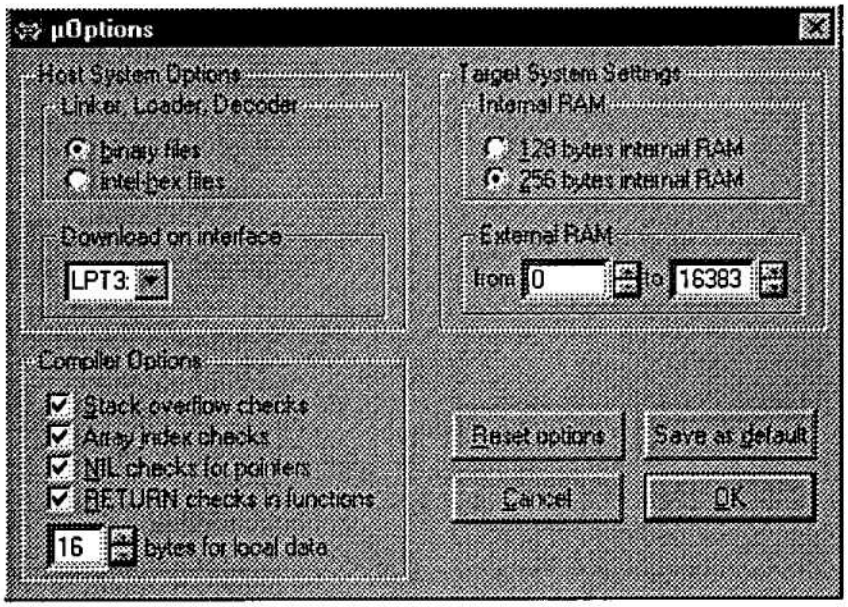

Figure 14 The µOberon options panel.

This panel is used to configure the system. Beside the compiler options the configuration of the target system or the interface used for downloading files to the target may be set as well.

Host System Options

This point comprises two settings concerning the host system. First it may be selected whether the linker, the loader and the core image decoder work with binary files where

the code bytes are ordered sequentially, or with Intel-Hex files that are accepted by a number of EPROM burner programs or simulators. The structure of Intel-Hex files can be found in Appendix D.

Secondly, the interface port for the download process may be specified here. The current version of µOberon only supports the parallel printer ports *LPT1:* to *LPT4:*, but later releases also may support serial interfaces.

Target System Settings

The target system settings finally define the kind of target system µOberon shall produce code for. The size of the internal RAM is used for the stack checks, and the external RAM is a coherent range in the 16 bit address space, i.e. the external RAM must be mapped into a linear range. The external RAM is used for dynamic memory allocation by the standard procedure NEW.

It is worth to mention that all target system settings plus the local data size of the compiler options of all object files to be linked together must match. The linker recognizes deviations and prompts an options inconsistency error.

Compiler Options

The four check boxes are used to enable and disable the generation of some additional checks. These checks will blow up the generated code and slow down the execution. But especially during the development phase stack overflow checks are recommended since stack overflows produce unpredictable behaviour of the target system, and it may be hard to identify them. If enabled, the stack pointer is checked after each PUSH instruction, and an overflow evokes trap 5.

Array index checks are inserted on each array access with a variable index. The actual index is compared with the legal upper bound of the array, and if it is exceeded, a trap 6 is called. Each time a pointer is dereferenced and NIL checks are enabled, the pointer is compared with NIL, and trap 7 is evoked if equal. RETURN checks are used in function procedures. These procedures are supposed to be left at the latest when the last statement is executed. If that check is enabled, a trap 8 call is inserted at the very end of the procedure, just before the procedure epilogue.

The following table lists the approximate overhead caused by those optional checks:

Check	*is inserted*	*additional code bytes*	*additional cycles*
Stack overflow	after each PUSH instruction	12	8
Array index	before each array access with a variable index	14	10
NIL	after each dereference	13	9
RETURN	at the end of each function procedure	7	5

The option ... *bytes for local data* is used to define the maximum size of local variables and parameters. This value is used to set the stack pointer just above the local variables. In fact the linker takes the maximum local data size of all modules to be linked to set the stack pointer, but as mentioned in the section about the static memory assignment, intermediate variable access implies that the size of the local variables window is known during compilation already.

For the programmer it is important to know the following:

- If there are no intermediate variable accesses, the local data size may be set to a high value without affecting the stack size. If the actual local data size is larger than the defined value, the compiler prompts an error.

- But if nested procedures exist, that value should not be too large since the stack size will be reduced, and more PUSH instructions are generated. In that case it would be wise to compile all modules with a large local data size first and then to set that value to the effective maximum local data size, the third number that the compiler writes to the log text.

It is not common that compilers allow disabling of those additional checks. In general, they are inserted by default. But especially in embedded systems where the application is tested, checks with known results are superfluous and only slowing down execution time and blowing up code size.

Options File

All settings of the options panel plus the state of the *New .sym* check box and the linker parameter text may be stored in the options file. If this file exists, it is read during initialization of the μOberon system, otherwise default values are loaded. Pushing the *Reset options* button forces the system to reload the values in the file, while *Save as default* writes the current state into the file. *Cancel* cancels the changed settings, and *OK* accepts the changes and closes the options panel. The structure of the options file is described in Appendix D.

5 Conclusion

This chapter summarizes what has been achieved, gives an outlook onto some areas for future work, and concludes the thesis.

5.1 Summary

µOberon has become an easy-to-use cross development system that allows comfortable access to all microcontroller resources. Compared to compilers that simulate the missing indirect-offset addressing, the static memory assignment technique is a great advantage and leads to better code, and especially when preferring 8 bit variables, the generated code even competes with assembler programs. The expenditure was worth it, even if it implied the complete change of the compiler skeleton. For instance all variable accesses had to be listed in the object file so that the linker would be able to fix the addresses.

The current version of µOberon has been used to implement some applications formerly written in assembler language and has proven its usefulness in practice as a result. The inline assembler is a helpful facility to write procedures that access internal or external hardware, e.g. that send a character to a liquid crystal display (LCD) interfaced by a shift register.

The main goals of µOberon have been achieved. But there are still points that may be completed or optimized. In the course of this thesis it was not possible to implement the whole Oberon system with a more sophisticated memory management supporting garbage collection. Later releases of µOberon will also support floating point arithmetic, maybe even provide multiprocessing facilities.

But I'm sure that it is possible to develop a large number of microcontroller applications with the current system. Always remember that Mcs51 microcontrollers are 8 bit processors only.

5.2 Benchmark

The following table lists the difference between µOberon and two other development systems for Mcs51 microcontrollers. 8031-Pascal is the compiler that comes along with [15]. None of the compared languages is object-oriented; µOberon does not support record extensions.

	Mcs51 µOberon	8031- Pascal	Keil C51 V5
Features			
Language	µOberon	NiliPascal	C
External RAM requirements	none	1 kB	none
Built-in assembler	✓		✓
Floating-point		✓	✓
Runtime libraries	inline	✓	✓
Decoder, Browser	✓		✓
Debugger, Simulator			✓
Optimizations			
Constant folding	✓		✓
Support AJMP/ACALL			✓
Support register banks			✓
Peephole optimizing	✓		✓
Register variables and parameters			✓
Common subexpression elimination			✓
Dead code elimination			✓
Shortest Program [instructions / bytes]	4 / 9	305 / 487	-
Cost		100 $	3000 $

The shortest possible program in µOberon consists of a jump over the module that only contains the RET instruction of the empty command, a call of that command, and an endless loop:

```
0000H   LJMP 0004H      ; jump over the empty command
0003H   RET             ; empty command
0004H   LCALL 0003H     ; call of the command
0007H   SJMP FEH        ; endless loop SJMP -2
```

This compact core image is achieved if no external RAM is present, so the initialization of the 16 bit heap pointer is superfluous, if no interrupt service procedure and no trap procedure is defined, and if no global and no local variables are used, so the stack may directly follow the first register bank, i.e. start at address 08H, which is default after reset; the stack pointer is preincremented on a PUSH, so its value after reset is 07H.

The other systems link runtime libraries on demand what considerably blows up the code size of an empty program. On the other side, code inlining implies larger code if numerous calls of the same sequence are required, because the sequence is inserted as a whole each time. Nevertheless the overhead of a procedure call is unnecessary what speeds up the execution, which was one of the goals of µOberon

5.3 Outlook

The process of developing µOberon has not yet terminated. I intend to write some further libraries, e.g. a multiprocessing extension. This will allow several processes running at the same time; see [8]. Coroutines are another idea.

Two stacks

Improvements of the current version of the µOberon compiler may be achieved by implementing two different stacks; an internal and an external one. This would allow a higher recursion depth. As mentioned, the external RAM is only used for dynamic

memory allocation at this time. Since there is no indirect-offset addressing for external accesses either, the static memory assignment technique is also offering for external variable accesses. This technique may be extended to the external memory, allowing more global and local variables. In that case, the internal stack would be used for return addresses only.

Dynamic item manager

As mentioned above, a dynamic item manager would solve the problem with the lost last register and making the peephole optimization superfluous. The dynamic item manager manages the accumulator and the B register, and before each use of them, they must be reserved. Since the accumulator is used for each arithmetic or logical operation, the content of the item that currently occupies the accumulator must be saved in a general purpose register. Compared with the benefits of a dynamic item manager, the expenditure to implement it is immensely large, so the current version doesn't provide it.

Tools

A symbolic debugger that allows the setting of breakpoints during runtime may be quite difficult to implement since the program code is located in a read-only memory. But it would be possible to write a post mortem debugger as provided by standard Oberon implementations, where the content of the stack and the procedure call sequences is displayed in case of a runtime error. Simulators that allow testing programs prior to burning EPROMs are already available and may be used in tandem with µOberon.

The µOberon user interface could be extended in order to simplify the management of several files as well as compiler and target system options belonging to a common application project.

Further points

Maybe some users need a more complex model of the external RAM, supporting several ranges instead of a single coherent one. There may be users that want to run a monitor program as well, in spite of its disadvantages mentioned above. In that case, it should be possible to force the linker to map the program to specific addresses. In the current version this has been kept deliberately simple.

The missing open array parameters or record extensions could be implemented by using the heap in the external RAM. So, an open array for instance is temporarily allocated in the heap, and the pointer is passed to the callee, whereby the size of the parameter is known on compilation time and the static memory assignment technique is applicable. This would require a garbage collector to collect the allocated block after the termination of the callee.

Certainly there are a lot of optimizations that could be integrated in µOberon:

- Use the ACALL instruction instead of LCALL, and AJMP instead of LJMP wherever possible. This would imply to check whether the new address lies in the same 2 kB block; only the linker knows where these boundaries are.
- Use DJNZ instruction for FOR statements with step -1.
- Parameter passing using registers instead of the stack.
- Register variables

plus all other well known optimization strategies.

One of the focal points when planning optimizations should be the prologue of interrupt service procedures. The current prologue pushes all registers that might be used onto the stack, so correct program execution is guaranteed even in case of interrupts. But the registers are saved a priori, without knowing which one will be used eventually, so it

would be better to save only the used registers. This is a further criterion that speaks for a two-pass compiler.

Other microcontrollers

Future work could include adaptations of µOberon to other, mainly 8 bit microcontrollers. In this case, only the backend has to be changed, i.e. the modules XOI, XOE, XOA and XOC, XOL, and XOD. If several adaptations are planned, one should play with the idea of implementing an intermediate code representation as in the Oberon-2 frontend-backend structure.

Derivatives

The Dallas 80C320 microcontroller is fully compatible to the 8032 and executes the same instructions in less cycles. Code generated by µOberon will run on these controllers.

[17] describes the 80C51XA architecture which is a 16 bit extension of the Mcs51 family. The XA CPU is compatible at source code level - the source code has to be reassembled using an XA assembler - and is 10 to 100 times faster than the 8 bit predecessor. Benchmark compares versus Motorola 68000 and other popular 16 bit architectures show even a speedup of 3 to 4. The reason why the µOberon compiler doesn't support the XA microcontrollers lies in the discrepancy between the two instruction sets. The XA provide much more sophisticated address calculations like the indirect-offset addressing which would force a completely different strategy in memory allocation and parameter passing, making the static memory assignment superfluous. XA is much more than a simple supplement to Mcs51 and has to be treated separately when building compilers hence.

Intel itself is producing successors of the Mcs51 family, the Mcs151 and Mcs251 families namely. Especially the first one seems to be interesting when programming in µOberon; these microcontrollers are fully binary code and pin compatible with the Mcs51 family, whereas being five times faster. This is achieved by pipelining. The Mcs252 on the other hand is somewhat more sophisticated by providing 16 bit instructions. Even they are still binary code compatible, so code produced by µOberon will run, but the new 16 bit instructions would remain unused.

Epilogue

µOberon has been developed from October 28th 1996 to February 27th 1997 in the course of my diploma work at the Swiss Federal Institute of Technology in Zurich. I first met the Mcs51 microcontroller family in 1990 and produced numerous assembler listings for hardware projects like colour organs and mobile robots. As mentioned above, the size of the sources really reduces the overall view, so it was hard to change or extend the programs later on. Now I began to rewrite my applications in µOberon.

I would appreciate if the µOberon users would give me some feedback, so suggestions for improvement can be taken into account for later versions. Thank you for your interest in µOberon, and I hope it does the same good turn in your projects !

Computer Scientists and Transparent Programming

During my studies of computer science I got the impression that programmers tend to keep their programs as their own secrets, not allowing other people to understand what they are doing and why they are doing it like that. There are few reasons not to comment a program. Maybe the memory is rare and one has to reduce the source text size. But this is hard to believe in the time of gigabyte discs. Or the programmer might take the view that it is *his* program and only *his* business. But it may become someone else's business later on. Perhaps the programmer had no time because the program had to run yesterday already... but consider the time that your successor will have to get into your code. Or possibly one is simply too lazy to write long identifiers...

It is common that a programmer has to update or to change another programmer's work. So, why not taking some time and comment the code ? Why using variable names like poT instead of ptrOffsetTable ?