

Programming in Micron

Rochus Keller

version 0.7, 2026-10-10

Table of Contents

Preface	8
How to read this book	9
Examples	9
I: The Language and Its Reasons	10
1. Why Micron?	12
2. A Tour of Micron	14
2.1. A first program	14
2.2. Two modules	16
2.3. The toolchain in one page	17
2.4. Why there is no extra optimization	18
2.5. Language levels at first sight	19
3. Notation, Types and Values	20
3.1. Notation	20
3.2. Number literals	20
3.3. Basic types	21
3.4. Signed and unsigned	22
3.5. Division	23
3.6. Bit operations	24
3.7. Sets	24
3.8. Enumerations	24
3.9. Constants and variables	25
4. Micron for C, Pascal and Go Programmers	26
4.1. If you come from C	26
4.2. If you come from Pascal, Modula-2 or Oberon	27
4.3. If you come from Go	28
4.4. Habits worth keeping, habits worth dropping	29
II: The Language in Depth	30
5. Program Structure: Modules, Procedures, Statements	32
5.1. Modules	32
5.2. Procedures	36
5.3. Statements	40

6. Structured Types	42
6.1. Arrays	42
6.2. Records	44
6.3. Pointers	47
6.4. Procedure types	48
6.5. Constructors	48
7. Abstraction without Object-Orientation	50
7.1. Procedures bound to record types	50
7.2. Delegates: behavior as a value	52
7.3. Interfaces: contracts satisfied structurally	53
7.4. One program, three ways: the benchmark suite	54
7.5. Choosing	55
8. Dynamic Memory	56
8.1. Three kinds of storage	56
8.2. new and dispose	57
8.3. Ownership without a collector	59
8.4. newgc, a preview	59
9. Object-Oriented Programming	62
9.1. Object types	62
9.2. Type tests, guards and type case	63
9.3. Two styles of object orientation	64
9.4. Objects and the heap	65
10. Generic Modules	66
10.1. Meta parameters	66
10.2. Instantiation at import	67
10.3. Constraints	68
10.4. What generics replace	69
11. Compile-Time Programming	70
12. Low-Level Programming	72
12.1. Bit manipulation	72
12.2. Reinterpretation: cast and usig	72
12.3. Addresses and memory-mapped devices	73
12.4. Registers, interrupts and level 0	74

12.5. The discipline of levels	75
13. Interfacing with C	76
13.1. Extern procedures	76
13.2. What maps, and what does not	76
13.3. Building against C code	78
13.4. Separate compilation	78
13.5. The moral	78
14. The Oakwood Modules	80
14.1. Interface modules and their implementations	81
14.2. Out: formatted output	81
14.3. Math and MathL	82
14.4. Args: the command line	82
14.5. Files: a C-like file interface	83
14.6. Time: measuring durations	85
14.7. What is deliberately missing	85
15. Graphics with Screen	88
15.1. The framebuffer	88
15.2. Window, dirty rectangles, and the loop	89
15.3. Events as packed integers	90
15.4. A sketch pad	90
15.5. The fractal fern	91
16. Stepwise Refinement and Data Abstraction	94
17. Error Handling, FINALLY and Exceptions	96
III: Systems Programming — The Micron System	98
18. The Machine and the Build	100
18.1. The machine	100
18.2. One command, one file	101
18.3. Main as the boot linker	101
18.4. Levels in the system	103
19. The Kernel: Memory, Heap, Collector	104
19.1. How new reaches the Kernel	104
19.2. The heap: Wirth's free lists	105
19.3. The conservative collector	106

19.4. Registers as roots	107
19.5. When collection runs	108
20. Disk, Files, Modules	110
20.1. Sectors	110
20.2. The file directory and the unsigned domain	110
20.3. Modules without a loader	111
20.4. What dynamic loading would take	112
21. Display, Input, Viewers	114
21.1. The frame buffer	114
21.2. Input	115
21.3. Viewers and the message hierarchy	116
21.4. The central loop	117
22. Texts, Fonts and the Editor	118
22.1. Texts: the piece list	118
22.2. Readers, Scanners, Writers	119
22.3. Fonts	120
22.4. TextFrames: the view	120
22.5. MenuViewers and Edit	122
23. Graphics and Applications	124
23.1. Graphics: objects with a method suite	124
23.2. Rectangles: a new class in one module	125
23.3. Draw: commands over a shared model	126
23.4. The small programs	126
24. Case Studies Beyond the System	128
24.1. S9: a Scheme system	128
24.2. P4: the portable Pascal compiler	129
24.3. A Smalltalk-80 virtual machine	129
24.4. Are We Fast Yet: one suite, four styles	130
24.5. What the studies say	131
IV: Appendices	132
Appendix A: The Tools	134
A.1. micc	134
A.2. The RV32 emulator	135

A.3. Building and running the Micron System	135
Appendix B: The Micron System: Build, Boot and Design Notes.....	136
B.1. Build and startup	136
B.2. The design decisions in summary	136
Appendix C: Wirth's RISC5 and Our RV32	138
Appendix D: Reading MIL.....	140
D.1. A module and its MIL	140
D.2. The instruction vocabulary.....	141
D.3. The two lower forms.....	141
D.4. Handwritten MIL	142
Appendix E: Syntax Summary.....	144
Appendix F: Predeclared Types and Procedures	148
F.1. Predeclared types.....	148
F.2. Predeclared function procedures.....	148
F.3. Predeclared proper procedures	149
Appendix G: Language Levels Quick Reference	152
Appendix H: Micron for C and Pascal Programmers.....	154
H.1. From C	154
H.2. From Pascal and Oberon	155
Bibliography	158

Preface

This book describes Micron, a systems programming language in the school of Niklaus Wirth, and shows how to use it: for ordinary programs, for libraries, and for an operating system that the reader can build and boot with one command.

Micron is not my first language design, and it did not appear out of nowhere. It is the result of a long journey, and the journey is perhaps worth telling.

I studied electrical engineering at ETH Zürich from mid eighties to early nineties, with a specialization in information technology and computer engineering. Wirth was famous and admired among the students, and I took the chance to attend his lectures in my fifth and sixth semester, including the compiler lectures, alongside a hundred of other students. We were taught Modula-2 on the Macintosh, and the compilers we studied were written in Modula-2. Oberon was developed while I was there, and Wirth might have mentioned it, but it took some more years until I encountered it.

My first close contact with Oberon came around 2000, during my doctorate, a thesis in molecular biology and biophysics on the NMR structure determination of biological macromolecules. An early idea was to design a programming language extending Oberon to implement the tools required by the molecular biologists, but after initial prototypes, that endeavor turned out to be too much for the project, which even so took more than five years to complete. I postponed the language for twenty years and implemented my scientific software in C++ with Qt instead, and with Lua as a scripting language. Even while pursuing my doctoral studies, I took the chance to take lectures at the computer science department. I was lucky that Bertrand Meyer arrived as Wirth's successor during those years, and I took all his lectures, studied the world of Eiffel, and attended the guest lectures of people like Adele Goldberg, Jean-Raymond Abrial, Dines Bjørner and Ivar Jacobson whom he invited.

The postponed idea did not go away. When I finally returned to it, I did not start with Micron. I first built Oberon+ [\[R1\]](#), a modernized and unified Oberon, in order to understand, by implementing it and by porting real systems to it, what Wirth's design actually gives and where it fails. The systems ported to Oberon+ included the 2013 Project Oberon System [\[R2\]](#) and different versions of Oberon System 3, and I also spent significant time with the Blackbox Framework and AOS/Bluebottle including the

corresponding language versions. I then explored the opposite direction with Luon, a higher-level language for managed application programming, with which I implemented yet another version of the Project Oberon System, and a Smalltalk-80 virtual machine [R3]. I also implemented parsers and analysis tools for Lisa Pascal, FreePascal, Simula 67, Algol 60, Mesa and Cedar, i.e. a whole bunch of languages along the Algol/Wirth lineage. Micron is the systems programming conclusion of these experiments. It is what remained when the lessons of Oberon+, the ideas that proved themselves in Luon and the other languages studied, together with thirty years of using C, C++ and their successors, including many embedded system, were applied to the question that Oberon left open: what should a small, safe and honest language for programming machines look like?

The answer, in one sentence: Oberon's clarity and leanness with the reach of C. The rest of this book is the long version.

How to read this book

The book assumes that you already program, whether in C, in Pascal or one of its descendants, in Go, in Oberon, or in a comparable language. It teaches Micron, not programming.

[Part I](#) explains what Micron is and why it exists, and introduces the notation and the type system. [Part II](#) covers the language feature by feature, TBD. [Part III](#) walks through the Micron System, TBD. The appendices contain everything needed to reproduce the examples, and reference material.

Examples

As I write this, I have already migrated a lot of decently large and complex applications and systems to Micron, including the obligatory Project Oberon System (both hosted and native on Risc-V), the Smalltalk-80 VM, the original P4 Pascal compiler [R4], different versions of the Are-we-fast-yet benchmark suite [R5], the Scheme 9 virtual machine [R6].

TBD

Part I: The Language and Its Reasons

This part explains why the Micron programming language exists, what using it looks like, and it lays down the vocabulary the rest of the book depends on. TBD

Chapter 1. Why Micron?

TBD

Chapter 2. A Tour of Micron

This chapter shows a complete Micron program, runs it, and walks through what the reader sees, in the manner of a guided tour: everything is shown once and explained briefly, and the details follow in later chapters. A reader who knows C, Pascal or Go should be able to read the programs of this chapter fluently thereafter.

2.1. A first program

The following module computes the prime numbers below 50 with the sieve of Eratosthenes.

```
module Primes

const
  N = 50

var
  sieve: array N of bool
  i, k, count: integer

  procedure clear()
  var i: integer
  begin
    for i := 0 to N-1 do sieve[i] := false end
  end clear

begin
  clear()
  count := 0
  for i := 2 to N-1 do
    if ~sieve[i] then
      inc(count)
      print(i); print(" ")
      k := i + i
      while k < N do
        sieve[k] := true
        k := k + i
      end
    end
  end
  end
  println("")
  print("primes below "); print(N); print(": "); println(count)
end Primes
```

Stored in a file `Primes.mic`, the program is run with the interpreter that is built into `micc`:

```
> micc --run Primes.mic
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
primes below 50: 15
```

The tour begins at the top.

Modules. A Micron compilation unit is a module. It opens with `module` and its name, and closes with `end` and the same name; the compiler checks the match. Everything in between belongs to the module: constants, types, variables, procedures, and finally an optional statement sequence between `begin` and `end`, the module body, which is executed when the module is loaded. There is no distinguished `main` function; the body of the main module is the program, and the main module is usually the module (or set of modules) on which no other modules depend. C programmers may read a module as the disciplined version of a translation unit: it has a name, its interface is checked, and its initialization is ordered. Pascal and Oberon programmers are at home, with one visible difference treated next.

No semicolons, no capitals. Micron inherits Oberon's syntax but relaxes two of its formalities. Statement separators are optional; statements are separated by white space, and semicolons may still be written where two statements share a line (as in `print(i); print(" ")` above). Reserved words and predeclared identifiers are recognized in all lower case or all upper case; `module` and `MODULE`, `integer` and `INTEGER` are the same word. This book writes lower case, which is easier on eyes trained by C, Pascal or Go; the Micron System sources of [Part III](#) do the same.

Declarations. The sections `const`, `type` (none in this example) and `var` declare names before use, as in the whole Pascal family. `N = 50` declares a compile-time constant; the array length `array N of bool` must be such a constant. Note the declared procedure `clear` nested among the declarations: procedures may be declared wherever declarations are allowed, including inside other procedures (but a nested procedure cannot access the local variables of its enclosing procedure; what looks like a closure in other languages is not one in Micron, see [Chapter 5](#)).

Statements. The control statements are those of Oberon: `if`, `while`, `repeat`, `for`, `case`, `loop`, each closed with an explicit `end` rather than braces or `begin/end` pairs. `~` is boolean

negation (`not` is also accepted), `&` is conjunction (`and` is also accepted). `inc(count)` is a predeclared procedure equivalent to `count := count + 1`. Assignment is `:=` and equality is `=`; the C programmer's `==` trap does not exist.

Input and output. `print` and `println` are predeclared procedures that accept a value of any basic type or a string; they exist so that programs like this one need no library. Real programs use library modules, such as the `Out` module of the Oakwood guidelines [R10] that ships with the compiler, or the `Files` and `Texts` modules of [Part III](#).

2.2. Two modules

The interesting properties of a module system appear only with a second module. The following module exports a procedure:

```
module Sorter

procedure sort*(a: ^[]integer; n: uint32)
var i, j, t: integer
begin
  for i := 1 to n-1 do
    t := a[i]
    j := i-1
    while (j >= 0) & (a[j] > t) do
      a[j+1] := a[j]
      dec(j)
    end
    a[j+1] := t
  end
end sort

end Sorter
```

and a main module imports and uses it:

```
module TourMain

import Sorter

var
  a: array 8 of integer
  i: integer

begin
  a[0] := 5; a[1] := 3; a[2] := 8; a[3] := 1
  a[4] := 9; a[5] := 2; a[6] := 7; a[7] := 4
```

```

Sorter.sort(@a, len(a))
for i := 0 to len(a)-1 do print(a[i]); print(" ") end
println("")
end TourMain

```

```

> micc --run TourMain.mic
1 2 3 4 5 7 8 9

```

The command names only the main module; the compiler finds `Sorter.mic` by its module name, compiles it first, and reports both modules. There are no header files and no makefile: the import list is the dependency graph.

The asterisk in procedure `sort*` is the export mark. Only marked identifiers are visible to importing modules; everything else is private. A second mark, `_`, exports read-only, which is used extensively in the Micron System to publish state that only its owning module may change. The importing module refers to the procedure as `Sorter.sort`; imported names are always qualified, so the reader of `TourMain` sees at every call which module serves it.

The parameter list of `sort` shows three things worth noting on first contact. First, `^[]integer` is a pointer to an open array of integers; the caller passes `@a`, the address of its array. Micron has no VAR parameters, so passing by reference is always visible as a pointer at the call site. Second, the open array `[]integer` (also written `array of integer`) has no intrinsic length; the length travels as a separate parameter, as it would in C. The predeclared function `len` yields the length of a fixed-size array. Third, `len` returns an unsigned value, and the parameter `n` is declared `uint32` to match: Micron does not mix signed and unsigned operands implicitly, a discipline that [Chapter 3](#) treats in detail.

2.3. The toolchain in one page

The compiler `micc` covers the whole workflow of this book with four modes:

```

micc --check Main.mic      # parse and typecheck only
micc --run  Main.mic      # compile to MIL and interpret
micc --run  project.micpro # the same, for a project file
micc --target rv32 -o Main ... # cross-compile and link an executable

```

`--run` uses no external tools and is how all examples of Parts I and II are executed. A

`.micpro` project file lists source files and options for larger programs; the "Are We Fast Yet?" suite is run this way. Cross-compilation produces a statically linked ELF executable for a real instruction set (RISC-V 32-bit, ARMv7, or x86); [Part III](#) uses it to build an entire operating system with one command:

```
micc --target rv32 -g --base 0 --runtime MIC+.mil -I . -o Main Main.mic
rv32vm --elf --ram 4 --disk disk.img Main
```

The first command compiles and links the 27 modules of the Micron System; the second boots the result on an emulated RISC-V machine. That this fits in two lines is not a demonstration trick but a design property of the language: no separate assembler, no linker script, no C runtime. The reader will find the complete instructions in [Appendix A](#) and the system itself in [Part III](#).

2.4. Why there is no extra optimization

In its current incarnation, `micc` performs no optimization passes beyond straightforward code generation. It is a design decision, in the tradition of Wirth's compilers, none of which ever contained an optimizer, and it deserves the same explanation as any other decision in this book.

For programs in the style this book teaches, the measured difference between unoptimized and fully optimized translation is modest. Comparing e.g. `-O0` with `-O2` for C compilers on realistic benchmark suites yields a typical factor between two and three on average ^[1]; published measurements on standard suites report averages in the same range (see [\[TBD\]](#)). Over the same decades in which optimizers grew to this factor, processors became faster by a factor of sixty to a hundred for the identical unoptimized code: a program whose performance was judged acceptable on a workstation of the early nineties runs nearly two orders of magnitude faster today before any optimizer has touched it.

A state-of-the-art optimizing infrastructure (such as [\[TBD\]](#)) comprises millions of lines of code, maintained by corporations. The costs are not only lines and work: optimizing compilers are big and slow, and their output no longer corresponds to the source under a debugger. A compiler small enough to be read, fast enough to be rerun without thought, and literal enough to be trusted is worth more to a systems

programmer than the last factor of two. Where that factor genuinely matters, the remedy of first resort is a better algorithm, and that is something no optimizer supplies.

The Micron System of [Part III](#) is the standing test of this position: an entire operating system with editor, compiler and graphics, compiled without an optimizer, responsive on an emulated machine. Wirth made the same wager for thirty years and never lost it.

2.5. Language levels at first sight

One line in [Part III](#)'s sources has no counterpart in the examples above. The Kernel module of the Micron System begins:

```
module Kernel [level=1]
```

The attribute declares the module's *language level*. Levels stratify the language by the runtime support a feature needs: level 0 code needs neither stack nor heap and is suitable for boot code and interrupt handlers; level 1 adds ordinary procedures and the stack; level 2 adds interface types; level 3 adds the heap with explicit `new` and `dispose`; level 4 adds object types with dynamic dispatch; level 5 adds garbage collection. A module may only import modules of the same or a lower level, so the property "this code does not allocate" is not a comment but a checked invariant. A module without the attribute, like the examples of this chapter, defaults to level 3; disciplined projects declare their levels, and the Micron System does so throughout. The levels organize [Part II](#) of this book: each chapter that introduces a feature states the level at which the feature becomes available.

[1] Averaged so that no single benchmark dominates; single benchmarks, and microbenchmarks in particular, scatter widely in both directions and prove nothing on their own; the suite average is the number that predicts real performance.

Chapter 3. Notation, Types and Values

This chapter describes the lexical notation of Micron and its basic types, with emphasis on the part of the design that most visibly separates Micron from its ancestors: explicit signed and unsigned integer types, and the discipline that governs their combination and conversion. Structured types follow in [Chapter 6](#).

3.1. Notation

Micron source text consists of identifiers, reserved words, literals, operators and delimiters, and comments. Identifiers are sequences of letters, digits and underscores beginning with a letter or underscore, and they are case sensitive. Reserved words and predeclared identifiers are the exception: each exists in a fully lower-case and a fully upper-case form with identical meaning, so `begin` and `BEGIN`, `integer` and `INTEGER` are interchangeable. Mixed case is not recognized. This book writes lower case throughout.

Comments come in two forms: `//` to the end of the line, and `(* ... *)`, which may be nested. Statement separators are optional; white space separates statements, and a semicolon may be used to put several statements on one line.

3.2. Number literals

An integer literal is decimal by default; the suffixes `H`, `O` and `B` select hexadecimal, octal and binary representation. A hexadecimal literal must begin with a digit, so the byte value 255 is written `0ffh`. Digits may be interspersed with underscores for readability:

```
1234
0ffh      // 255
1010b     // 10
1_000_000
```

An unsuffixed integer literal has no fixed type. Its type is *universal integer*, and the concrete type is determined from the context in which the literal is used: in `u + 1` with `u: uint32`, the literal becomes a `uint32`; in `s + 1` with `s: int32`, an `int32`. Where the context is not sufficient, a type suffix fixes the type explicitly: `100u1` is a `uint8`, `100i8` an `int64`, and

the width-free suffixes `u` and `i` select the smallest unsigned or signed type that holds the value.

A real literal contains a decimal point with at least one digit before it, and optionally a scale factor: *E* for *times ten to the power of*, with `F` forcing type `flt32` and `D` forcing `flt64` (the traditional names `real` and `longreal` are synonyms, see below):

```
4.567e8
0.5770-6
```

A character literal is written as the ordinal number of the character in hexadecimal notation followed by `x`, for example `0dx` for carriage return; characters are 8-bit Latin-1. A string literal is enclosed in single or double quotes and must not span lines; a string of length 1 is interchangeable with a character literal. String literals can be concatenated with `+`. Finally, a *hex string* encodes raw bytes between dollar signs, two hex digits per byte, with white space ignored; the Micron System uses hex strings for icons, cursor shapes and font data:

```
const arrow = $0F0F 0060 0070 0038 001C 000E 0007 8003$
```

3.3. Basic types

The basic types and their sizes are fixed by the language, not by the platform:

Type (synonyms)	Size	Values
<code>bool</code> (<code>boolean</code>)	1 byte	<code>true</code> , <code>false</code>
<code>char</code>	1 byte	Latin-1 characters
<code>uint8</code> (<code>byte</code> , <code>u1</code>)	1 byte	0 .. 255
<code>int8</code> (<code>i1</code>)	1 byte	-128 .. 127
<code>uint16</code> (<code>u2</code>)	2 bytes	0 .. 65535
<code>int16</code> (<code>shortint</code> , <code>i2</code>)	2 bytes	-32768 .. 32767
<code>uint32</code> (<code>u4</code>)	4 bytes	0 .. 4294967295
<code>int32</code> (<code>integer</code> , <code>i4</code>)	4 bytes	$-2^{31} .. 2^{31}-1$
<code>uint64</code> (<code>u8</code>)	8 bytes	$0 .. 2^{64}-1$

Type (synonyms)	Size	Values
int64 (longint, i8)	8 bytes	$-2^{63} \dots 2^{63}-1$
flt32 (real, r4)	4 bytes	IEEE 754 single precision
flt64 (longreal, r8)	8 bytes	IEEE 754 double precision
set	4 bytes	sets of the integers 0 .. 31

The traditional names (`integer`, `byte`, `real`, ...) make Oberon and Pascal sources read naturally; the systematic names (`int32`, `uint8`, `flt32`, ...) say exactly what they are, in the manner of C's `stdint.h`. Both name the same types; the choice is style. Signed integers use two's complement. Within each family the larger type *includes* the smaller: `int64` $\geq$ `int32` $\geq$ `int16` $\geq$ `int8`, likewise for the unsigned types and for the two floating point types.

Two consequences of this table deserve emphasis for the C programmer. There is no platform-dependent `int`: `integer` is `int32` everywhere. And there is no implicit conversion between integers and floating point in either direction; `flt(i)` converts an integer to floating point, and `floor(x)` truncates a floating point value to an integer.

3.4. Signed and unsigned

Micron requires that the two operands of an arithmetic operator are both signed integers, both unsigned integers, or both floating point. A universal integer literal adapts itself to the other operand, so `u + 1` needs no annotation; but `u + s` with `u`: `uint32` and `s`: `int32` is an error, not a silent conversion. The same discipline applies to assignment: a byte value is not assignable to an `integer` variable, although every byte value is representable, because the types belong to different families. Modula-2 had this rule for `CARDINAL` and `INTEGER`; C abandoned it in favor of the "usual arithmetic conversions", whose interactions of rank, promotion and signedness are complicated enough that entire coding standards exist to avoid them.

Since programs legitimately need to move values between the families, Micron provides explicit operations, distinguished by intent:

- `val(T, x)` *converts the value*: the result is the value of `x` represented in type `T`, with overflow or precision loss accepted where the value does not fit.

- `cast(T, x)` *reinterprets the representation*: the bits of `x` read as a `T` of the same size. `cast(int32, 0xffffffff)` is -1.
- `sigl(x)` converts an unsigned value to the next larger signed type, which always preserves the value; the unary `+` or `-` applied to an unsigned operand is a shortcut for it. `sigc(x)` and `usig(x)` reinterpret between signed and unsigned of the same width, as shortcuts for the corresponding `cast`.

The following excerpt from this chapter's example module shows the discipline in practice:

```

u := 0xffffffff      // largest uint32 value
s := cast(int32, u)   // reinterpret the bits: -1
println(s)
s := -5
u := cast(uint32, s)  // 4294967291
println(u)
b := 200
s := sigl(b)          // value conversion to the next larger signed type
println(s)

```

The distinction between `val` and `cast` makes visible in the source what C hides in one `cast` syntax: whether the program converts a value to another type or reinterprets an unchanged bit representation as that type.

3.5. Division

`div` and `mod` truncate toward zero, like the division instructions of common processors and the `/` and `%` operators of C and Go: `(-7) div 2` is -3 and `(-7) mod 2` is -1. Oberon and Pascal define floor division instead, where `(-7) DIV 2` is -4 and the remainder is never negative. Where the Oberon semantics is genuinely wanted, the predeclared functions `obdiv` and `obmod` provide it:

```

x := -7; y := 2
println(x div y)      // -3
println(x mod y)      // -1
println(obdiv(x, y))  // -4 Oberon semantics on request
println(obmod(x, y))  // 1

```

Arithmetic is not overflow-checked; the result of an overflowing operation is truncated

to the width of its type.

3.6. Bit operations

Bit manipulation is provided by predeclared functions rather than operators: `band`, `bor`, `bxor`, `bnot` for the bitwise connectives, `shl` and `shr` for logical shifts, `asr` for the arithmetic shift right. Their names keep bit twiddling recognizable at a glance, which matters in systems code where logical and and bitwise `band` may appear in the same expression.

TBD

3.7. Sets

The type `set` holds a set of the integers 0 to 31, represented as a machine word. Set constructors list elements and ranges in braces; `+`, `-`, `*` and `/` denote union, difference, intersection and symmetric difference; `in` tests membership. `ord(s)` yields the word as an unsigned number, and `bset(x)` the set corresponding to a number:

```
caps := {0, 2, 8..11}
println(2 in caps)      // true
caps := caps + {3}
caps := caps - {2}
println(ord(caps))      // the bit pattern as a number: 3849
```

Sets are Wirth's alternative to C's flag words, with the compiler checking element ranges; the Micron System uses them for everything from sector maps to key states.

3.8. Enumerations

An enumeration declares a type by listing its values:

```
type Color = (red, green, blue)
```

The values are ordered constants of a distinct type, not integers: a `Color` cannot be added to anything, and `ord` must be used to obtain the ordinal where one is needed. Oberon dropped enumerations; Micron restores them, since a systems language without enumerations pushes programmers to integer constants and loses the type

checking exactly where state machines and device modes would profit from it.

3.9. Constants and variables

A constant declaration binds an identifier to a compile-time expression. Constant expressions are evaluated by the compiler, using the same arithmetic rules as at run time; if all operands are unsuffixed literals, the constant is itself of universal integer type and adapts to its use sites like a literal.

Variables are declared in `var` sections with name and type. The initial value of a variable is *undefined*, at module level as well as locally; Micron does not zero memory behind the program's back. Where a defined initial value is wanted, the program assigns it, or uses `default(T)`, which yields the zero-like value of any type. This is a deliberate difference from Go and from managed languages: implicit zeroing costs time exactly where systems code can least afford it, and it hides missing initializations instead of exposing them. Level 5 is the exception: where a garbage collector scans memory, pointers are automatically NIL, as [Chapter 8](#) explains.

Chapter 4. Micron for C, Pascal and Go Programmers

The reader of this book already has programming experience. The next chapters present Micron systematically, feature by feature, and they are written to be read in order; this one is different in kind. It talks to three arrivals separately, in their own terms: what will feel familiar, what is genuinely different, and which habits are worth carrying across or leaving behind. [Appendix H](#) gives the same mapping as lookup tables; this chapter gives the reasoning. Readers may fairly skip the sections about languages they do not come from.

4.1. If you come from C

C and Micron agree on more than the surface suggests. Both are small languages that compile to predictable machine code; both let the programmer say exactly where data lives; both trust the programmer with pointers. A C programmer's mental model of the machine, frames, alignment, the cost of an indirection, transfers without change, and so does most of the working vocabulary: `int32/uint8` are the fixed-width types C99 spells `int32_t/uint8_t`, records lay out like structs, `div` and `mod` truncate exactly like `/` and `%`, and unsigned arithmetic wraps. Micron even has `goto`, kept deliberately so that C is easier to migrate.

The first difference that bites is that *implicit conversions are gone*. C converts freely between integer types and lets the "usual arithmetic conversions" decide what a mixed expression means; four decades of subtle bugs, sign-extension surprises and compiler warnings testify to the price. In Micron, signed and unsigned are two separate families, mixing them in one expression is a compile-time error, and every crossing is written out: `sig` and `sigl` to move between families, `val` for value conversion, `cast` for bit reinterpretation. Expect your first Micron programs to be red with such errors; expect, after a week, to find the C rules harder to defend than you did. [Chapter 3](#) presented the system in full.

The second difference is where the discipline comes from. In C, hygiene is a matter of convention and tooling: `-Wall`, static analyzers, code review. Micron moves a useful part of that into the language. `const` on a parameter is transitive and cannot be cast away ([Chapter 5](#)); a module exports exactly what carries a mark and nothing else

(ibid.); pointer arithmetic exists but only by name, `ptrdiff_t`, never by accident; and the language levels (Chapter 2) turn "this file must not allocate" from a review comment into a compiler check. None of this costs run-time performance, which is the point: the claim Micron makes against C is not that it is safer at a price, but that most of C's danger never paid for anything.

What C programmers must actually give up is small: the preprocessor (imports are semantic, constants are `const`, conditional compilation is a source directive, Chapter 11); the textual header file (the compiler extracts interfaces itself, Chapter 5); variadic functions (an array argument takes their place, Chapter 5); and `malloc` is untyped bytes (`new` allocates a typed value, Chapter 8). In exchange, the linker model becomes trivial, a build is `micc` on the root module (Chapter 2), and the FFI chapter (Chapter 13) shows that the C world stays one `extern C` declaration away, so nothing already built is lost.

4.2. If you come from Pascal, Modula-2 or Oberon

The Wirthian reader is at home immediately: modules with explicit import and export, strong typing, `begin/end` structure, nested procedures, and a compiler that is fast because the language was designed single-pass. Micron is a descendant of this line through Oberon+ and Luon, and Chapter 1 told that story. What this section owes the Pascal programmer is a candid list of the places where Micron chose differently, and why.

Micron is *lower-level on purpose*. Pascal and Oberon describe an abstract machine and hide the real one; Micron's reason to exist is systems and embedded programming, so the real machine shows through where that is useful. Integer types have explicit widths instead of one `INTEGER`; records lay out C-compatibly; a pointer may address a stack variable, a module variable or the middle of an array, not only a heap record; and the lowest language level has neither heap nor stack, which is what startup code and tiny microcontrollers need. Where Oberon quarantined such things in the pseudo-module `SYSTEM`, Micron admits them openly, typed and named, and uses the level system rather than a module name to mark the code that needs review.

Three concrete changes deserve their warning here. There are *no VAR parameters*: a procedure that modifies its argument takes a pointer, and the caller writes `count(@text, 'b')`, so every call site shows which arguments can change, information that in Pascal

only the declaration has. `div` and `mod` truncate toward zero, following C and standard Pascal, not Oberon's floor semantics; `obdiv` and `obmod` are there when the Oberon behavior is wanted, and [Chapter 3](#) explained why the default flipped. And garbage collection is not assumed: `new` pairs with `dispose`, and collected allocation is a separate, higher language level (`newgc`), chosen per module rather than imposed on the program. An Oberon programmer should read [Chapter 8](#) before porting anything.

What Micron gives back is the modern half of the family tree that classical Oberon refused: enumerations and variant records return (Pascal programmers never stopped missing them), constructors write structured constants in one line, generic modules ([Chapter 10](#)) replace the source-level template tricks once used for generic programming in Modula-2, interfaces and objects ([Chapter 7](#), [Chapter 9](#)) give dynamic dispatch without requiring a class-centred programming model, and finally with `pcall` ([Chapter 17](#)) covers the error-handling ground that FreePascal's `try` does, in a form a systems kernel can afford. The FreePascal programmer in particular will find the distance smaller than expected: most of what FPC added to Turbo Pascal exists here, usually in a leaner form.

4.3. If you come from Go

Go and Micron do not descend from one another, but they share ancestors and an attitude. The module/package with explicit exports, the composition-over-inheritance instinct, structural embedding of records, the refusal of implicit conversions between integer types, fast whole-program compilation: a Go programmer recognizes the Oberon inheritance in Go the moment Micron makes it visible again. Even the export mark is the same idea, spelled differently: Go capitalizes the identifier, Micron appends `*`, and Micron's `▯` adds read-only export, which Go lacks.

The real difference is the runtime, and it changes daily habits more than any syntax. Go's model is services: goroutines, channels, and a garbage collector that is always on; the language can be high-level because a scheduler and a collector are underneath everything. Micron's model is systems: there is no scheduler, no collector unless a module asks for one (level 5, `newgc`), no green stacks, and a program's memory behavior is decided by the program. Where a Go program says `go serve(c)`, a Micron program has an event loop it wrote itself ([Chapter 15](#) builds one; the Micron System of [Part III](#) is

exactly such a loop grown into an operating system). Concurrency as a language feature is simply not part of Micron; what exists is what systems code always had, interrupts and explicit state machines.

Smaller adjustments follow from the same divide. Slices become pointers to arrays whose length travels separately, so the convenient `len(s)` on a dynamic value is gone ([Chapter 6](#), [Chapter 8](#)); strings are 0X-terminated char arrays, not immutable values; maps are a library data structure you choose or write, not syntax; `defer` becomes `finally`, once per procedure rather than per statement ([Chapter 17](#)); and errors are return values by convention here as there, with `pcall` in the exceptional corner where Go would panic. Interfaces exist and are satisfied structurally exactly as in Go ([Chapter 7](#)), but `dispatch` stays a visible mechanism a programmer can point at, not runtime magic. A Go programmer who has ever asked what the runtime does under a `pgraph` will find in Micron the answer-by-construction: nothing the program did not write.

4.4. Habits worth keeping, habits worth dropping

The three sections meet in a short list. Keep from C the cost model and the respect for layout; drop the preprocessor artistry, the casts, and the assumption that every integer converts to every other. Keep from Pascal and Oberon the discipline of interfaces and the habit of letting the compiler check what can be checked; drop the expectation that the machine is hidden and that somebody else manages memory. Keep from Go the small-interface style and composition instinct; drop the assumption of an omnipresent runtime. What all three readers can rely on: the language is small enough to hold in one head, the compiler is fast enough to keep up with thought, and nothing happens at run time that the program text does not show. The systematic treatment starts on the next page.

Part II: The Language in Depth

This part covers the language feature by feature. TBD

Chapter 5. Program Structure: Modules, Procedures, Statements

[Chapter 2](#) showed the shape of a Micron program in passing. This chapter treats it systematically: the module as compilation unit, the declaration of procedures, and the statements of the language.

5.1. Modules

Every language of the Algol line structures computation the same way: statements compose into procedures, procedures into programs. What Modula-2 added, and every Wirthian language since has kept, is one more level above the procedure: the module. Its purpose is different in kind from the procedure's. A procedure abstracts an *action*; a module draws a *boundary*,

TBD

A module is a collection of declarations of constants, types, variables and procedures, together with an optional statement sequence that assigns initial values to the variables. It is the unit of compilation, and it is the unit of information hiding: nothing declared inside a module is visible outside unless its declaration carries an export mark.

```
module M [level = 1]
import Files, F := Fonts, lib.Util

const ...
type ...
var ...
procedure ...

begin
    // executed once, when the module is loaded
end M
```

The import list names the modules whose exported identifiers M uses; an exported `x` of module A is written `A.x`, and the form `F := Fonts` introduces a shorter alias. An import may carry a path prefix such as `lib.Util`; how paths map to files is the compiler's business, not the language's. Cyclic imports are illegal. The statement sequence after

`begin` is the module body; it runs when the module is loaded, after the bodies of all imported modules, so a module can rely on its imports being initialized. In a statically linked program such as the Micron System, loading simply means execution in import dependency order before the main program starts.

The bracketed attribute after the module name declares properties of the whole module. The most important attribute is the language level, introduced in [Chapter 2](#): `[level = 1]` (or `[standard = 1]`) restricts the module to the features of level 1 and permits imports at levels 0 and 1 only. A second attribute, `visibility`, changes the default visibility of module-level declarations, which is useful for modules that export nearly everything.

5.1.1. Export marks

An identifier declared at module level may carry one of three marks: `*` exports it, `~` exports it read-only, and `~` (or no mark) keeps it private. Read-only export is a Micron inheritance from Oberon that C has no counterpart for: a module can publish a variable or a record field that clients may inspect but not assign, without writing accessor procedures. The fields of an exported record type are private by default and must be marked individually, so a module can export a type while keeping parts of its representation to itself.

Pascal programmers will recognize the module as the unit that replaces both the unit interface/implementation split and the header file discipline of C. There is no separate interface text to maintain: the compiler extracts the exported interface from the marked declarations, and separate compilation rests on that extracted interface TBD.

5.1.2. Clients and providers

Decomposing a program into modules divides its programmers into two roles: the *provider*, who writes a module and carefully chooses what to export, and the *client*, who imports it and deals with the module through the exported names only. The set of exported declarations, seen from the client's side, is called the module's *interface*. A small example makes the two views concrete. The provider is a character queue of fixed capacity:

```

const Cap = 64

var
  count: uint32           // read-only for clients
  buf: array Cap of char
  in_, out: uint32

procedure Put*(ch: char): boolean
begin
  if count = Cap then return false end
  buf[in_] := ch in_ := (in_ + 1) mod Cap
  inc(count)
  return true
end Put

procedure Get*(ch: ^char): boolean
begin
  if count = 0 then return false end
  ch^ := buf[out] out := (out + 1) mod Cap
  dec(count)
  return true
end Get

begin
  count := 0 in_ := 0 out := 0
end Buffers

```

The interface consists of `Put`, `Get` and the counter `count`; the array `buf` and the two indices remain invisible. The client sees exactly this and nothing more:

```

import Buffers

var
  ch: char
  ok: boolean

begin
  ok := Buffers.Put('h') ok := Buffers.Put('i')
  println(Buffers.count)           // 2: reading is allowed
  // Buffers.count := 0           // error: read-only export
  // Buffers.buf[0] := 'x'       // error: buf is not exported
  while Buffers.Get(@ch) do print(ch) end
  println("")                     // hi
end UseBuffers

```

The choice of exports is the design decision. Exporting `buf`, `in_` and `out` would let a client insert an element directly, and the first client that did so incorrectly would corrupt the queue for every other client, with the failure surfacing far from its cause. By exporting

procedures only, the module *guarantees the integrity* of its data: no execution of any client, correct or buggy, can violate the queue invariant, because the only code that touches the representation is the code in this file. C offers this protection per file with `static`, but without a checked import mechanism on the other side; in Micron the compiler verifies both sides of the contract.

The read-only mark refines the contract. `count` is information a client legitimately wants, and a `Count()` function procedure would be the classic answer; the `⊖` mark provides it for free while still preventing assignment, as the commented lines in the client show. A thin interface pays a second dividend besides safety: it is the module's documentation. If names are well chosen, the half dozen marked declarations state what the module does more reliably than any comment, and they are checked against the implementation on every compilation.

The third dividend is *freedom of implementation*. Nothing in the interface of `Buffers` reveals that the representation is a circular array; a linked list, or a heap block of growing size, would present the identical interface. The provider may exchange the representation tomorrow, and no client needs to be edited, because no client can have depended on what it could never see. TBD

5.1.3. Separate compilation and initialization order

Modules are compiled separately, but not independently. The distinction, due to Modula-2, is worth spelling out for readers coming from C, where the compiler processes one translation unit at a time and takes declarations on faith: a mismatch between a header and its library surfaces, at best, at link time, and at worst at run time. When `micc` compiles a module, it reads the *actual interfaces* of the imported modules, extracted from their sources by the compiler itself, and type-checks every use of an imported name against the real declaration. The faith is gone; a client compiled against yesterday's interface of a changed provider is rejected, not linked.

Separate compilation dictates an order: a provider before its clients. The import graph therefore must be free of cycles, and the compiler rejects cyclic imports outright rather than attempting a resolution. The restriction bites occasionally, when two modules seem to need each other, but the cure is always the same and always an improvement: the declarations both sides depend on move into a third module below both, or one

direction of the dependency is inverted with procedure variables TBD. A cycle-free import graph is also what makes the initialization rule of this chapter's first section well defined: module bodies run in import dependency order, providers first, so every module may rely on its imports being fully initialized, and the whole program starts in a defined state without any of the static-initialization-order accidents C++ programmers know.

5.2. Procedures

A procedure declaration consists of a heading and a body; the procedure name is repeated after `end`, a small redundancy that pays off in long files. Procedures whose heading specifies a result type are function procedures and are called inside expressions; every control path of a function procedure must return a value. All declarations inside a procedure are local, and locals have undefined values on entry, as module variables do. Procedures may be nested, and in contrast to traditional Oberon a local procedure may be assigned to a procedure variable.

5.2.1. Parameters

Micron has no VAR parameters. Where Oberon or Pascal would pass a variable by reference, a Micron procedure takes a pointer, and the caller applies the address operator:

```
procedure count(const s: ^array of char; c: char): integer
var i, n: integer
begin
  i := 0; n := 0
  while s[i] # 0x do
    if s[i] = c then inc(n) end
    inc(i)
  end
  return n
end count
```

The call site then reads `count(@text, 'b')`, and the reader sees at a glance that `text` might be affected. What Micron adds instead of VAR is `const`: a parameter marked `const` is read-only in the body, and the property is transitive, so the elements and fields reached through a `const` pointer are read-only too. This restores what C expresses with

`const T*`, but checked without casts and holes.

There are no variadic parameters. A variable number of arguments of one type is passed as a pointer to an array plus a count; heterogeneous argument lists, as in a formatted print, are passed as an array of pointers to `any` together with a format string. Both cost one line at the call site and remove an entire category of ABI complexity and run-time error from the language.

The homogeneous case in full:

```
procedure sum(const a: ^array of int32; n: uint32): int32
var i: uint32; s: int32
begin
  s := 0; i := 0
  while i < n do s := s + a[i]; inc(i) end
  return s
end sum
```

The parameter is an open array (no length in the type), so the count travels beside it, and the same procedure serves arrays of any length. The caller fills an array of whatever size the call needs and passes its address together with its length:

```
println(sum(@{1,7,3,9}, 4))    // 20
```

An array constructor (see [Section 6.5](#)) is used as an anonymous local variable (with the `@` operator), so we don't need an extra declaration.

What C's `sum(4, 1, 7, 3, 9)` hides in `varargs` machinery, the ellipsis, the promotion rules, `va_list` and the unchecked agreement between count and arguments, Micron states as an ordinary array whose construction the reader can see and whose element type the compiler checks at every call.

The heterogeneous case, formatted printing, works the same way with `pointer to any`, the type to which a pointer of any type is assignable. The format string names the type of each argument, and the procedure casts accordingly (a subset of the usual specifiers suffices to show the pattern; the rest follow the same scheme):

```
type str = array of char

procedure format(const fmt: ^array of char; const args: ^array of ^any)
var i, a: uint32
```

```

begin
  i := 0; a := 0
  while fmt[i] # 0x do
    if fmt[i] = '%' then
      inc(i)
      case fmt[i] of
      | 'd': print(cast(@int32, args[a])^); inc(a)
      | 's': print(cast(@str, args[a])); inc(a)
      | 'c': print(cast(@char, args[a])^); inc(a)
      | '%': print('%')
      end
    else
      print(fmt[i])
    end
    inc(i)
  end
end format

```

The call site again builds the argument array with a constructor:

```

var n: int32; c: char; s: array 8 of char

n := 42; c := 'x'; s := "world"
format("hello %s: n = %d, c = %c (100%)", @s, @n, @c); println("")

```

which prints `hello world: n = 42, c = x (100%)`. Where C's `printf` trusts the caller, the specifiers here are checked against an argument list whose construction is visible; a mismatch between specifier and argument remains possible, but it is confined to one small procedure whose cases can be read, rather than distributed over every variadic call in the program.

A function may return records and arrays by value, which Oberon and Oberon-2 did not allow. Open arrays cannot be returned, since their length is not part of the type.

5.2.2. **Inline, invar, extern and forward**

Four markers can follow a procedure heading, and together they replace a surprising amount of C preprocessor and linker lore:

inline requests that the body be expanded at the call site; no call happens, and no stack frame is needed. Inline procedures cannot be recursive. At language level 0, where no stack exists yet, all procedures are implicitly inline; TBD.

invar (invariable) declares a procedure whose result depends only on its arguments: it may call only other *invar* procedures, and it cannot touch module variables. The reward is that the compiler can evaluate it at compile time, so an *invar* procedure may be used in constant declarations:

```
procedure cube(x: integer): integer invar
begin
    return x*x*x
end cube

const c8 = cube(8)    // evaluated by the compiler
```

Where C programs compute tables with external scripts or preprocessor contortions, a Micron program writes an ordinary procedure and marks it *invar*. TBD

extern declares a procedure implemented elsewhere: in another Micron object file, or, with the form *extern C*, in a C library reached through the foreign function interface, optionally with the external name given as a string. This is the door to the outside world TBD.

A *forward declaration*, marked with *^*, announces a procedure before its implementation, for the case of mutual recursion in a single-pass compilation:

```
procedure ^ odd(n: integer): bool

procedure even(n: integer): bool
begin
    if n = 0 then return true end
    return odd(n-1)
end even

procedure odd(n: integer): bool
begin
    if n = 0 then return false end
    return even(n-1)
end odd
```

5.3. Statements

The statement forms are those of the Oberon family, with a uniform bracketing: every structured statement is closed by `end`, and the dangling-else problem of C and Pascal does not exist. Statement separators are line ends; semicolons are optional.

Assignment is `:=`, equality is `=`, inequality is `#`. The boolean operators `&` and `or` evaluate their second operand only if the first does not decide the result, exactly like `&&` and `||` in C.

`if` takes any number of `elsif` arms and an optional `else`. `while` and `repeat ... until` are the guarded loops. `for` iterates an integer or enumeration control variable from a first to a last value, with an optional constant by step; a downward loop needs an explicit negative step. `loop` with `exit` expresses repetitions with exits in the middle or at several points:

```
procedure classify(const s: ^array of char)
var i: integer
begin
  i := 0
  loop
    case s[i] of
      | 'a'..'z', 'A'..'Z': print("L")
      | '0'..'9': print("D")
      | 0x: exit
      else print("?")
    end
    inc(i)
  end
  println("")
end classify
```

The `case` statement selects on an integer, enumeration or character value; labels are constants or ranges, may not overlap, and if no label matches and there is no `else`, the program aborts, which is the correct default for state machines that are supposed to be exhaustive. A second form of `case` selects on the dynamic type of an object pointer TBD.

Finally, Micron has `goto`, with labels declared in the statement sequence and jumps forbidden into structured statements. It was not added out of conviction. It was added because translated C code is full of it, and mechanical migration of C programs is a stated goal of the language. New Micron code should not need it, and the Micron

System does not contain a single one.

The `return` statement terminates a procedure, with an expression when the procedure is a function. In proper procedures the end of the body implies the return, so an explicit `return` marks an exceptional early exit. Return statements are illegal in module bodies.

Chapter 6. Structured Types

The basic types of [Chapter 3](#) are combined into structured types in four ways: as arrays, records, pointers and procedure types. TBD This chapter also introduces constructors, the literal notation for writing structured values.

6.1. Arrays

An array is a sequence of elements of one type, indexed from 0. A fixed array declares its length as a constant expression:

```
var line: array 128 of char
    m: array 2 of array 3 of integer    // an array of arrays
```

Arrays are one-dimensional; a matrix is an array of arrays, indexed `m[i][j]`. The length of a fixed array is a compile-time quantity, obtained with `len`. Whole arrays of the same type may be assigned with one assignment.

An *open* array, written `array of T` or `[]T`, leaves the length out of the type. Open arrays occur in two places: as the base type of a pointer, and as the type of dynamically allocated storage ([Chapter 8](#)). TBD. No length is stored, no descriptor exists, and the program passes the length separately, as it would in C. What Micron adds over C is that fixed arrays do know their length at compile time, that `len` yields it, and that index checks can be generated where lengths are known.

A string is an array of char terminated by the character `0x`, as in C. A string literal may be assigned to any char array it fits into, and `strlen` returns the length up to the terminator.

6.1.1. Working with arrays

A short worked example shows the idioms together. The task: count how often each letter occurs in a line of text. The natural data structure is an array of 26 counters, indexed by the letter's position in the alphabet, and the index is computed from the character with `ord`, the technique every table-driven program uses:

```
procedure tally(const s: ^[]char; freq: ^[26]uint32)
```

```

var
  i: uint32
  ch: char
begin
  i := 0
  while s[i] # 0x do
    ch := s[i]
    if (ch >= 'a') and (ch <= 'z') then
      inc(freq[ord(ch) - ord('a')])
    elseif (ch >= 'A') and (ch <= 'Z') then
      inc(freq[ord(ch) - ord('A')])
    end
    inc(i)
  end
end tally

```

The parameter types deserve a look. The line arrives as `const ^[]char`, a pointer to an open array: the procedure works for a line of any length and promises not to modify it, and no length parameter is needed because the string carries its own terminator. The counters arrive as `^[26]uint32`, a pointer to a *fixed* array: here the length is part of the contract, and an index computed from a letter can be checked against it. Choosing between the two forms, open when the length is the caller's business, fixed when it is part of the type, is a small decision that recurs in every Micron program. The calling module owns the storage and passes both addresses:

```

var
  freq: array 26 of uint32
  k, max: uint32

begin
  line := "the quick brown fox jumps over the lazy dog"
  for k := 0 to len(freq) - 1 do freq[k] := 0 end
  tally(@line, @freq)
  max := 0
  for k := 0 to len(freq) - 1 do
    if freq[k] > max then max := freq[k] end
  end
  println(max)    // 4: 'o' occurs four times
end Histogram

```

Note `len(freq)` in the loops: because the length of a fixed array is a compile-time quantity, changing 26 in the declaration adjusts every loop, and no separate size constant needs maintaining.

6.2. Records

A record combines a fixed number of fields with possibly different types:

```
type
  Point = record x, y: integer end
  Rect  = record origin: Point; w, h: integer end
```

Fields are accessed by name, `r.origin.x`, and records of the same type are assignable as a whole. When a record type is exported, its fields stay private unless individually marked; a module can thus publish a type as an opaque handle, or expose some fields read-only with `var`. The layout of a record follows the same rules as a C struct on the target platform, which matters for interoperability and for memory-mapped hardware.

6.2.1. Embedded records

A field of record type may be prefixed with `inline`, which makes the field name transparent: the members of the embedded record appear as members of the enclosing one, while the explicit path remains valid.

```
type
  Named = record
    inline p: Point
    name: array 16 of char
  end

n.x := 3           // the fields of Point appear in Named
n.p.y := 4         // the explicit path is also valid
n.name := "origin"
println(n.x + n.p.y) // 7
```

Go programmers know this as struct embedding; here as there, it gives composition much of the convenience that inheritance is usually bought for, without introducing any subtype relation. If names collide, the directly declared member wins, and the embedded one stays reachable through its explicit path.

6.2.2. Arrays of records

The most common composite of the two constructors so far is the *table*: an array whose elements are records. A table of mountains, to be sorted by height:

```
const N = 4

type
  Entry = record
    name: array 12 of char
    height: integer    // meters
  end

var peaks: array N of Entry
```

Sorting moves whole entries, and whole-record assignment makes the insertion sort read like its textbook description:

```
procedure sort(a: ^[]Entry; n: uint32)
var
  i, j: uint32
  t: Entry
begin
  for i := 1 to n - 1 do
    t := a[i]           // whole-record assignment
    j := i
    while (j > 0) and (a[j-1].height < t.height) do
      a[j] := a[j-1]
      dec(j)
    end
    a[j] := t
  end
end sort
```

`t := a[i]` copies the complete entry, name and height together, where a C program would call `memcpy` or copy field by field; the compiler knows the size from the type. Note also that the procedure takes `^[]Entry` with a separate count: the sort works for a table of any length, so the open-array form with an explicit `n` is the right signature, the same decision as in the histogram example. Filling and printing the table completes the program:

```
var k: uint32

begin
  peaks[0].name := "Everest"    peaks[0].height := 8848
```

```

    peaks[1].name := "Kilimanjaro" peaks[1].height := 5895
    peaks[2].name := "Mont Blanc"  peaks[2].height := 4808
    peaks[3].name := "K2"          peaks[3].height := 8611
    sort(@peaks, N)
    for k := 0 to N - 1 do
        print(peaks[k].height) print(" ") println(@peaks[k].name)
    end
end Table

```

The name field is assigned with a string literal, using the rule stated in the array section; the assignment checks at compile time that the literal fits. Running the module prints the four peaks in descending order. Tables of records, searched, sorted or indexed, are the workhorse data structure of systems programs TBD.

6.2.3. Variant records

A record may end in a variant part: a set of alternative field groups that share the same storage, discriminated by a tag field that the program maintains:

```

type
    Kind = (rect, circle)
    Shape = record
        kind: Kind
        case
        | r: Rect
        | radius: integer
    end
end

```

The size of the variant part is the size of its largest alternative. This is the classical variant record of Pascal, or the tagged union that C programmers assemble by hand from a `union` inside a `struct`; Micron gives the pattern a syntax but, at this language level, no enforcement, exactly like C. Systems code needs the overlay; file formats, protocol frames and instruction encodings are variant records by nature. Where checked alternatives are wanted, object types with type tests provide them. TBD

6.2.4. Bit fields

Fields of boolean or integer type in the fixed part may be packed at the bit level by appending `bits n` to the field declaration:

```
type Status = record
  ready:  bool bits 1
  error:  bool bits 1
  ..      bits 2      // explicit padding
  count:  uint8 bits 4
end
```

Consecutive `bits` fields are grouped into packing units, each sized to the smallest integer type that holds them, and `..` inserts explicit padding. In contrast to C, where the layout of bit-fields is implementation-defined and their use in drivers a portability hazard, the packing rules are part of the language. TBD

6.3. Pointers

A pointer holds the address of a value of its base type, which may be any type: a record, an array, a basic type, a procedure. The declaration is `pointer to T` or, shorter, `^T`; dereferencing is `p^`, and the two usual abbreviations of the Oberon family apply, `p.f` for `p^.f` and `p[i]` for `p^[i]`.

A pointer value comes from three sources: `nil`; the address operator `@` applied to a variable, parameter, field or array element; or dynamic allocation with `new`, `newinit` or `newgc` (Chapter 8). The second source distinguishes Micron sharply from Oberon, whose pointers could refer to heap records only. Micron pointers may point into the stack, into module variables, into the middle of arrays and records. That is what systems programming requires, and it is what Oberon had to hide behind `SYSTEM.ADR` and untyped loopholes. Micron admits the capability openly and keeps it typed; `ptroff(p, n)` performs typed address arithmetic where a driver or an allocator genuinely needs it.

6.4. Procedure types

A procedure type describes a parameter list and result type; its values are procedures and `nil`:

```
type
  Area = procedure(const s: ^Shape): integer
```

A procedure variable is assigned any procedure whose signature matches, and called like a procedure. The example module stores `rectArea` and `circleArea` in the same variable and computes areas through it; TBD. C has the same feature with a less readable notation; Pascal and Oberon programmers will feel at home.

6.5. Constructors

A constructor writes a structured value as a literal: an optional type name followed by a braced component list.

```
s1 := Shape{ kind: rect, r: Rect{ Point{1,2}, 10, 20 } }
s2 := Shape{ kind: circle, radius: 5 }
```

Components may be positional, named (`field: value`), or indexed for arrays (`[3]: x`); unnamed fields receive their `default` value, so a partial constructor is well-defined. When the target type is known, the type name may be dropped. If all components are constant, the constructor is a constant and may appear in `const` declarations, giving initialized tables at no run-time cost. A char array takes a string literal as constructor, a byte array a hex string. For a record with a variant part, the components must be named and only one variant may be set. Prefixed with `@`, a constructor denotes an anonymous local variable, convenient for passing a one-off argument by pointer.

Constructors came to Micron from Luon, and they remove a whole class of initialization boilerplate that Oberon programs paid for the absence of: the assignments in the example above would otherwise be eight statements.

Chapter 7. Abstraction without Object-Orientation

Object-oriented programming acquired, over three decades, a reputation it did not earn on its merits alone, and lately a rejection it does not entirely deserve either. The interesting observation behind both is that most of what programs need from "objects" is not inheritance and not virtual dispatch. It is three humbler things: bundling operations with the data they operate on; passing behavior around as a value; and programming against a contract rather than a concrete type.

Micron provides all three below the object-oriented level. Procedures can be bound to record types, statically and without any dispatch machinery; a bound procedure together with its receiver can be captured in a *delegate* (i.e. bound procedure type); and *interface types* specify contracts that any suitable type satisfies structurally, in the manner of Go. None of this requires objects, inheritance, or virtual method tables; all of it is available from language level 1 or 2 and resolved, wherever possible, at compile time. Object types and dynamic dispatch, and the cases where they are genuinely the right tool, are the subject of [Chapter 9](#).

This chapter develops the three means in order, then compares them on a real program: the "Are We Fast Yet?" benchmark suite (see [\[R5\]](#)), which exist in several complete Micron implementations differing exactly in the means of abstraction they use.

7.1. Procedures bound to record types

A procedure can be declared with a *receiver*, which binds it to a record type declared in the same scope:

```
type
  Ball = record
    x, y, xVel, yVel: int32
  end

  procedure Ball.init()
  begin
    self.x := 0; self.y := 0
    self.xVel := 0; self.yVel := 0
  end Ball.init
```

`Ball.init` is local to the type `Ball`: its name does not compete with other declarations of the module, and other types may have their own `init`. Within the body, the receiver is available as `self`, a pointer to the record the procedure was called on. The receiver is always passed as a pointer, regardless of the syntax at the call site; for a variable `b: Ball`, the call `b.init()` is valid and passes `@b` automatically. (An explicit receiver name may be declared instead of the implicit `self`; both forms occur in this book.)

The value of this feature is best measured against its absence. The same code in plain procedural style, the style of the C-derived *Procedural* version of the benchmark suite, reads:

```
procedure Ball_init(me: ^Ball)
begin
    me.x := 0; me.y := 0
    me.xVel := 0; me.yVel := 0
end Ball_init

...
Ball_init(@balls[i])
```

Nothing is wrong with this; it is exactly what a C programmer writes, with the type prefix `Ball_` maintained by hand and the address taken explicitly at every call. Bound procedures let the compiler maintain the prefix and take the address: the declaration becomes `procedure Ball.init()`, the call becomes `balls[i].init()`. No dispatch is involved; the procedure is resolved at compile time from the static type of the receiver, exactly like the underscored version. Records do not support inheritance, so there is nothing to dispatch on. The feature costs nothing at run time; what it removes is boiler-plate, and with it a class of clerical errors.

Coming from Go: this is Go's receiver syntax without Go's method-set rules. Because the receiver is always a pointer, the distinction between value receivers and pointer receivers, with its puzzles about addressability and automatic dereferencing, does not exist.

7.2. Delegates: behavior as a value

Procedure types have been part of the Wirth line since Modula-2, and Micron has them ([Chapter 6](#)): a variable of type `procedure(x: integer)` holds any procedure with a matching signature, and the Oberon System of [Part III](#) is built almost entirely on this mechanism. But a plain procedure variable holds only the procedure. If the procedure is to operate on a particular record, the record must travel separately, by convention as an explicit first parameter:

```
type
  Sink1 = record
    put: procedure(me: ^Sink1; x: integer)
    sum: integer
  end

  procedure accumulate(me: ^Sink1; x: integer)
  begin
    me.sum := me.sum + x
  end accumulate

  procedure feed1(s: ^Sink1)
  var i: integer
  begin
    for i := 1 to 10 do s.put(s, i) end
  end feed1
```

This is the "Wirth style": a record of procedure fields, installed at initialization time, invoked with the record itself as first argument (`s.put(s, i)`). It works, it is completely explicit, and [Part III](#) shows it carrying an entire operating system. Its weakness is also visible: the pairing of object and procedure is reestablished by hand at every call, and nothing prevents calling `s.put(t, i)` with the wrong record of the same type.

A *delegate*, in the language a variable of *type-bound procedure type*, captures the pair as one value. The type is written `proc(^)(...)`, read "a procedure bound to some receiver":

```
type
  Acc = record sum: integer end

  procedure Acc.put(x: integer)
  begin
    self.sum := self.sum + x
  end Acc.put
```

```

procedure feed2(put: proc(^)(x: integer))
var i: integer
begin
  for i := 1 to 10 do put(i) end
end feed2

...
var a: Acc
feed2(a.put)    // captures both a and Acc.put

```

The designator `a.put`, not called but assigned, produces the delegate: a compound value containing the receiver pointer and the procedure. `feed2` needs to know nothing about `Acc`; it receives behavior with its state already attached. This is what closures and iterator blocks provide in higher-level languages, reduced to its systems-programming core: no environment capture, no allocation, two words passed by value.

The *Hybrid* version of the benchmark suite is built this way; the delegate-based emulation of the original Smalltalk closures and iterators is markedly leaner than the object-oriented version based on inheritance and virtual methods.

7.3. Interfaces: contracts satisfied structurally

The third mechanism abstracts one step further: not one procedure, but a set of them, specified without reference to any implementing type.

```

type
  Sink3 = interface
    proc put(x: integer)
  end

  procedure feed3(s: Sink3)
  var i: integer
  begin
    for i := 1 to 10 do s.put(i) end
  end feed3

  ...
  var a: Acc
  feed3(@a)    // Acc satisfies Sink3: it has put(x: integer)

```

An interface declares procedure signatures and nothing else. A pointer to any record (or object) type whose bound procedures cover those signatures may be assigned to an interface variable: no declaration of intent, no `implements` clause, no common base type.

Satisfaction is structural, as in Go. The contract is the shape.

Two properties distinguish Micron's interfaces from the object-oriented machinery of [Chapter 9](#). First, they are available at language level 2, below dynamic memory and below objects; a driver may define interfaces over statically allocated records. Second, for record types there is no dynamic dispatch involved in satisfying them: which put is meant is known at compile time; the interface value, like the delegate, is a receiver pointer paired with procedures. The compiler generates the necessary tables per interface and type pair, which is the reason interfaces sit at level 2 rather than level 1.

Compared with a record of procedure variables, the interface moves the wiring from run time to compile time: nothing is installed, nothing can be installed wrongly or forgotten, and the calls need no explicit receiver argument. Compared with a delegate, the interface groups several procedures under one binding.

The complete three-style example of this chapter is a single runnable module, `examples/ch07/Styles.mic`; all three variants compute the same sum through the same loop, differing only in the mechanism by which behavior reaches it.

7.4. One program, three ways: the benchmark suite

Doctrine is cheap; the "Are we fast yet" ([\[R5\]](#)) repository offers evidence. The suite, nine to fourteen benchmarks originally designed to compare languages, including object-heavy programs such as Richards, DeltaBlue and a JSON parser, exists in four complete Micron implementations, the following three at language levels 2 and 3, none using object types:

Procedural is a direct migration of the C implementation: records, free-standing procedures with the type prefix in the name, explicit `@` at call sites. It is the baseline; everything the other two versions do can be done this way, at the price of boiler-plate.

Hybrid rewrites the same programs with bound procedures and delegates. The framework's central type shows the Wirth style retained where it fits, a `Benchmark` record with four procedure fields installed by each benchmark's `create`, while the benchmarks themselves use bound procedures throughout, and delegates replace the closures of the Smalltalk original.

Interfaces replaces the record of procedure fields by a four-procedure interface; each benchmark provides a record whose bound procedures satisfy it structurally:

```
type
  Benchmark* = interface
    proc benchmark(): integer
    proc verifyResult(result: integer): boolean
    proc innerBenchmarkLoop(innerIterations: integer): boolean
    proc release()
  end
```

The framework calls `me.verifyResult(me.benchmark())` on the interface; the `create` procedures shrink to an allocation and a `return`, with no field wiring and no casts. The suite's Readme draws the conclusion this chapter is built on: the interface version is the most elegant of the three, with the least boiler-plate, and all of it below the object-oriented level.

The reader is encouraged to run all three (`micc --run awfy.micpro` in each directory) and to diff one benchmark across the versions; `Bounce.mic` is the smallest. Few exercises teach the cost and payoff of abstraction mechanisms more directly.

7.5. Choosing

A rule of thumb, refined through [Part III](#):

- Data with associated operations, one implementation, no substitution: **bound procedures**. Zero cost, less boiler-plate; this should be the default over free procedures with type-prefixed names.
- A single behavior handed to another module, such as a callback, a comparison, an iterator body or a completion action: a **delegate** (or a plain procedure type where no state is needed).
- Several implementations behind one contract, chosen by the provider rather than the caller: an **interface**.
- Substitutable implementations that must also **extend** one another, with inherited state, overridden behavior, or heterogeneous collections of a common base: only then object types, in [Chapter 9](#).

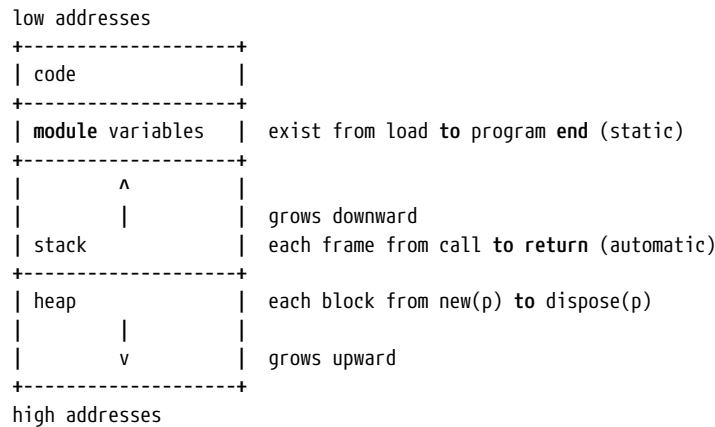
Chapter 8. Dynamic Memory

Everything up to this point ran without a heap. Level 3 introduces it, and with it the pointers whose targets outlive the procedure that created them. This chapter covers explicit allocation and deallocation; garbage collection is a separate decision and a separate chapter TBD.

8.1. Three kinds of storage

Before the heap arrives it is worth drawing the whole picture, because a Micron variable lives in one of three places, and the place determines its lifetime:

The three storage classes and their lifetimes (the Micron System’s arrangement)



Module variables are allocated once, when the program is loaded, and exist until it terminates; their addresses are fixed. Local variables and parameters live in the *stack frame* of their procedure: the frame is created by the call and vanishes with the return, which is why a pointer to a local variable must never survive the procedure, a rule the language states but, at these levels, cannot enforce. Both classes share one property: their size and number are determined by the program text. A program restricted to them, which is any program below level 3, can be proven to run in bounded memory, which is why embedded firmware often stops there deliberately.

What the two static classes cannot express is a data structure whose size depends on the *input*: a list of unknown length, a tree grown from a parsed file, a buffer sized at

run time. For these the third class exists. The *heap* is storage that the program requests and releases explicitly, block by block, in any order, and a heap block's lifetime is bounded by no procedure and no module: it begins with `new` and ends with `dispose`, wherever they occur. That freedom is the point, and it is also the cost, because the compiler can no longer pair allocation and release for you. The exact arrangement of the classes in the address space is the target platform's business (the picture above shows the map of the Micron System, inherited from Project Oberon [R2], where the stack grows down toward the module data and the heap grows up above it); the lifetime rules are the language's.

8.2. new and dispose

If `p` is a pointer to a type `T`, the predeclared procedure `new(p)` allocates a variable of type `T` on the heap and assigns its address to `p`. If `T` is an open array (and only then), the length is passed as a second argument: `new(p, n)`. Memory obtained with `new` is uninitialized, like every other variable in Micron; `newinit(p)` and `newinit(p, n)` allocate and set the storage to its default value where that is worth its cost. When the storage is no longer needed, `dispose(p)` frees it. All three are available from level 3.

The classical example is the linked list:

```

type
  Item      = ^ItemDesc
  ItemDesc = record value: integer; next: Item end

procedure prepend(head: ^Item; v: integer)
var p: Item
begin
  new(p)
  p.value := v
  p.next := head^
  head^ := p
end prepend

procedure sum(p: Item): integer
var s: integer
begin
  s := 0
  while p # nil do s := s + p.value; p := p.next end
  return s
end sum

```

```

procedure free(head: ^Item)
var p, q: Item
begin
  p := head^
  while p # nil do q := p.next; dispose(p); p := q end
  head^ := nil
end free

```

The `ItemDesc` record refers to itself through the pointer type `Item`, declared before it in the same type section; the deferred resolution of pointer base types makes the forward reference legal. The procedures take `^Item` where they modify the head of the list, and `Item` where they only walk it, so the signature states the effect.

A picture of the resulting store shows what the declarations mean. After three calls of `prepend`, one module variable of pointer type refers to the first of three records, each allocated by its own `new` and each holding the address of the next:

A linked list: one static pointer, three heap records

module variable	heap		
+-----+	+-----+	+-----+	+-----+
head ---->	value 3 +-->	value 2 +-->	value 1
+-----+	next ---+--+	next ---+--+	next nil
	+-----+	+-----+	+-----+

Nothing in the picture is contiguous: the three records may lie anywhere in the heap, in any order, and only the `next` fields hold the structure together. This is the fundamental trade against the array: an array buys direct access by index with one block of fixed size, the list buys insertion and removal in constant time with one pointer per element and no index at all. Removing the middle record is one pointer assignment plus a `dispose`; removing the middle element of an array is a shift of everything behind it.

A dynamically allocated open array is, once more, a typed memory area and nothing else:

```

procedure fill(a: ^[]integer; n: uint32)
var i: integer
begin
  for i := 0 to n-1 do a[i] := i*i end
end fill

```

```

new(a, 5)           // allocate 5 integers, uninitialized
fill(a, 5)
println(a[4])      // 16
dispose(a)

```

The allocation knows the length, but the array does not store it; `fill` receives the length as a parameter, exactly as its C counterpart would. Programs that need self-describing buffers declare a record with a length field and the array behind it, and pay for the field visibly.

8.3. Ownership without a collector

C programmers know the discipline that explicit deallocation demands, and Micron does not pretend to remove it: whoever owns a heap value must dispose of it exactly once, and no one may use it afterwards. What the language contributes is a smaller attack surface. There is no pointer arithmetic by accident (only by explicit `ptroff` and pointer `inc/dec`), no silent conversion between pointers and integers, no cast needed to allocate, and the level system confines allocation to the modules that declare at least level 3. A driver at level 1 cannot leak memory, because it cannot allocate any; the reviewer knows this from the module heading, not from reading every line.

The `finally` section of a procedure supports the discipline further: its statements run whenever the procedure body terminates, on the normal path and on early returns alike, so a procedure that allocates temporary storage can place the `dispose` once, next to the `exit`, instead of before every `return`. TBD

Fragmentation and allocation strategy belong to the runtime rather than the language. The interpreter uses the host allocator; TBD. The language only defines the contract: `new`, `newinit` and `dispose` can be routed to a custom implementation, so a freestanding program can choose its own allocator by implementing three procedures.

8.4. `newgc`, a preview

Level 5 adds `newgc`, which allocates under a garbage collector; such storage is never disposed explicitly. The two allocation worlds do not mix on a single pointer: what was allocated with `new` is freed with `dispose`, what was allocated with `newgc` is reclaimed by

the collector. The Micron System runs its user-facing data structures, texts and viewers under the collector while its kernel stays at the explicit levels; TBD.

Chapter 9. Object-Oriented Programming

[Chapter 7](#) built abstraction from records, delegates and interfaces, all without inheritance and mostly without dynamic dispatch. Level 4 adds the remaining pair: object types with inheritance, and type-bound procedures that dispatch on the dynamic type. This chapter shows the machinery, and then returns to the question of [Chapter 7](#): when is this the right tool, and in which of two quite different styles.

9.1. Object types

An object type is declared like a record, with an optional base type in parentheses:

```
type
  Figure = ^FigureDesc
  FigureDesc = object
    x, y: integer
  end

  Circle = ^CircleDesc
  CircleDesc = object (FigureDesc)
    radius: integer
  end

  Box = ^BoxDesc
  BoxDesc = object (FigureDesc)
    w, h: integer
  end

procedure (f: Figure) area(): integer
begin
  return 0
end area

procedure (c: Circle) area(): integer // redefinition
begin
  return 3 * c.radius * c.radius // coarse
end area

procedure (b: Box) area(): integer // redefinition
begin
  return b.w * b.h
end area
```

CircleDesc and BoxDesc are extensions of FigureDesc: each consists of the inherited fields followed by its own. A pointer to an extension is assignable to a pointer of the base

type, so a `Circle` may stand wherever a `Figure` is expected. This is the subtype relation that records deliberately lack; whether a type participates in it is decided by writing `object` or `record`, and is visible at the declaration.

Procedures are bound to an object type in either of two equivalent notations. The example uses the traditional Oberon-2 receiver syntax [R7], where the receiver is a named parameter in front of the procedure name, `procedure (f: Figure) area(): integer`, and the body refers to it by that name. The bound syntax of Chapter 7, `procedure Figure.area(): integer` with the implicit receiver `self`, is equally valid for object types. Which to use is a matter of taste; Micron prefers the latter; the traditional Oberon-2 syntax is meant for backward compatibility.

The difference to the record-bound procedures of Chapter 7 is dispatch: a procedure bound to an object type is implicitly bound to every extension, an extension may override it with a redefinition whose parameters match, and a call `f.area()` executes the procedure bound to the *dynamic* type of `f`, which the compiler resolves through a method table at run time. In the example, the same statement `f.area()` computes a circle area or a box area, depending on what `f` points to at that moment. Within a redefinition, `self.area^()` calls the overridden procedure of the base type, the *super call* of other languages.

9.2. Type tests, guards and type case

Assignments narrow static knowledge: after `f := c`, the variable `f` is statically a `Figure`, although it points to a circle. Three constructs recover the information, all checked:

- The test `f is Box` yields true if the dynamic type of `f` is `Box` or an extension of it.
- The guard `f(Box)` asserts the dynamic type and gives the designator the static type `Box`; a failing guard aborts the program.
- The type `case` branches on the dynamic type, and within each arm the case variable is known with the tested type:

```
procedure describe(f: Figure)
begin
  case f of
  | Circle: print("circle r="); print(f.radius)
  | Box:    print("box ");      print(f.w); print("x"); print(f.h)
```

```

| NIL:    print("nothing")
else      print("figure")
end
if f # nil then print(" area="); print(f.area()) end
println("")
end describe

if f is Box then println(f(Box).w) end    // 4

```

The `nil` label handles the empty pointer in the same statement. This is the mechanism Oberon used instead of pattern matching, and for systems code it has aged well: the tests are explicit, cheap and local, and the compiler checks that each label is an extension of the case variable's type.

9.3. Two styles of object orientation

The literature tends to present object orientation as one thing. The Oberon lineage suggests distinguishing two styles, and Micron supports both.

The first, which this book calls *Simula OO*, is the familiar one: business objects, class hierarchies, virtual dispatch. An extension inherits fields and procedures, overrides some, and clients program against the base type. The *Figures* example is Simula OO in miniature, and for data that is genuinely hierarchical, a scene graph, an instruction set, a widget tree, it remains the most direct expression available.

The second, *Wirth OO*, appears throughout Project Oberon [R2] and its Micron port: a record carries a procedure variable, the *handler*, and components communicate by sending messages through it. Inheritance is involved, but on the message side: the message types form a hierarchy rooted in a most general message type, and a handler takes that base type as its parameter,

```
Handler* = procedure(F: Frame; M: pointer to FrameMsg)
```

and recovers the concrete message with the constructs of the previous section, a type case, an `is` test or a type guard on the dynamic message type. The call through the handler is an ordinary procedure-variable call; extension happens by adding message types and by wrapping or replacing handlers at run time. [Chapter 21](#) shows it at full scale in the viewer system, where frames of unknown, separately compiled types

cooperate through one message protocol.

The engineering trade-off is real. Simula OO gives static checking of the full procedure set and the fastest dispatch, but fixes the hierarchy at compile time. Wirth OO gives open protocols and late binding of behavior; the object's record type stays outside any hierarchy, only the messages inherit, but the compiler cannot enumerate what a handler understands, and unknown messages must be ignored by convention. Interface types ([Chapter 7](#)) sit between the two: structural contracts, checked statically, no hierarchy. Micron provides all three, each at the lowest level that can support it, and [Part III](#) demonstrates them.

9.4. Objects and the heap

Object types can live on the heap: `new(c)` allocates them like records, and level 4 presupposes level 3. Stack allocation is possible, and a bound procedure called on a stack object receives its address, as noted in [Chapter 7](#). The dynamic type information that `is`, `guards` and the type `case` consult is stored with the object; it is the one hidden field Micron adds, the price of dispatch, and it is paid only by object types. Records stay exactly as C would lay them out.

Chapter 10. Generic Modules

A stack of integers and a stack of characters differ in one type name and nothing else. Languages without generics offer two bad answers: copy the code, or erase the type (the `void*` container of C, with the type checks abandoned at the door). Micron's answer is the generic module: the unit of genericity is the module itself, not the individual type or procedure. This is the approach of Oberon+ and, before it, of Modula-3's generic interfaces, and Ada's generic packages, and CLU's clusters, and it keeps the machinery remarkably small.

10.1. Meta parameters

A module becomes generic by a list of meta parameters after its name. Meta parameters denote types or, with the `const` prefix, constants. A parameter may carry a constraint like an ordinary parameter declaration, as `CAP: uint32` below.

```
module Stack(T; const CAP: uint32)
var
    a: array CAP of T
    n*: integer

procedure push*(x: T)
begin
    a[n] := x; inc(n)
end push

procedure pop*(): T
begin
    dec(n)
    return a[n]
end pop

begin
    n := 0
end Stack
```

Inside the module, `T` and `CAP` are used like an ordinary type and an ordinary constant: `CAP` fixes the length of a static array, `T` its element type. Nothing in the module says integers or characters, and the compiler accepts every operation that is valid for any type, such as assignment and parameter passing. An unconstrained meta parameter is itself generic: without the `uint32` constraint, `CAP` would have no integer type and could

not serve as an array length. Constraints are the subject of a later section; the Stack needs this one from the start.

10.2. Instantiation at import

A generic module is instantiated by importing it with actual meta parameters, which must be constant expressions or type names:

```
module UseStack
import si := Stack(integer, 10)
      cs := Stack(char, 4)
begin
  si.push(3); si.push(4)
  cs.push('x')
  println(si.pop() + si.pop()) // 7
  println(cs.pop())           // x
end UseStack
```

Every distinct set of meta actuals creates a distinct module: `si` and `cs` have separate storage, separate `n`, and separate module bodies, executed like any other imported module body. The identity is determined by the actuals, not by the import: two imports of `Stack(integer, 10)`, in the same module or in different ones, denote the same instance, so generic instances are shared across a program the way ordinary modules are. The same generic module may be instantiated several times in the same importer, distinguished by the alias. There is no instantiation at the call site and no inference: what is generic and where it becomes concrete is visible in the import list, and the cost model is that of ordinary modules.

For a systems language the constant meta parameter matters as much as the type parameter. A driver can import `Stack(Frame, 32)` and get a statically allocated, fixed-capacity structure with no heap involvement; the same generic module serves at level 1, where `new` does not exist. This is the intended pattern for driver buffers, DMA descriptor rings and similar fixed-size structures, typed without preprocessor arithmetic.

10.3. Constraints

An unconstrained meta parameter permits only the operations that every type supports. The `sum` procedure below needs more, namely `+` and the constant `0`, so it constrains `T`. A constraint on a basic type is written inline, as here and in the Stack; for types the heading cannot see yet, imported interfaces among them, the `where` declaration after the import list constrains the parameter by name. Each meta parameter takes at most one constraint:

```
module Sum(T: integer)
  procedure sum*(const a: ^[]T; n: integer): T
  var i: integer; s: T
  begin
    s := 0
    for i := 0 to n-1 do s := s + a[i] end
    return s
  end sum
end Sum
```

```
module UseSum
  import s1 := Sum(int32)
  var a: array 3 of int32
  begin
    a[0] := 1; a[1] := 2; a[2] := 39
    println(s1.sum(@a, 3))    // 42
  end UseSum
```

The constraint names a type with which the actual parameter must be assignment compatible; `integer` admits the signed integer family. The constraint is checked when the module is instantiated, and within the generic module it determines the permitted operations, so a misuse of `T` is reported in the generic module, at the place that causes it, not as an avalanche in the client's build.

A constraint can also be an interface type, a use of interfaces at compile time that I have not seen in other languages. The actual type must satisfy the interface structurally; it need not be declared with any reference to it:

```
module Writable [level = 2]
  type Writer* = interface proc write() end
end Writable
```

```
module Printer(T) [level = 2]
```

```

import Writable
where T: Writable.Writer
procedure writeTwice*(x: ^T)
begin
    x.write(); x.write()
end writeTwice
end Printer

module UsePrinter [level = 2]
type Star = record end

procedure Star.write*()
begin print("**")
end write

import p := Printer(Star)
var s: Star
begin
    p.writeTwice(@s)    // **
    println("")
end UsePrinter

```

Within `Printer`, the constraint tells the compiler that `T` has a `write` procedure, so `x.write()` is legal and statically checked. `Star` satisfies `Writer` by shape alone. Readers coming from Go will recognize the idea and notice what is absent: there are no type sets, no unions of constraints, no interplay between methods and operators. A constraint is a type, satisfaction is the ordinary compatibility test, and the semantics of the generic module can be read without a second language in the head. It is also what sets these generics apart from their Modula-3 ancestors, where a generic interface stated no constraints at all and errors surfaced only when an instantiation was compiled.

10.4. What generics replace

The C programmer covers this ground with macros and `void*`; the specification's own summary is accurate: generic modules offer most of the capabilities of preprocessor macros, but parsed, scoped and type checked. The price of the module-level granularity is that a single generic procedure needs a module around it. In practice the module boundary was already there: a container, its operations and its invariants belong together, and the AWFY collection classes `TBD`, `Vector(T)` among them, show the pattern in a real application.

Chapter 11. Compile-Time Programming

TBD

Chapter 12. Low-Level Programming

The chapters so far stayed at the higher levels of the language. This chapter descends to the features that exist because hardware is made of bits, addresses and registers: the bit manipulation functions, bit reinterpretation, memory-mapped device access, register access, and the two lowest language levels where these features live. Oberon collected such operations in the pseudo-module `SYSTEM`, a fig leaf that every real program had to import. Micron makes them ordinary predeclared functions and confines them by level instead: a module's heading states how deep it reaches, and an importer sees that statement.

12.1. Bit manipulation

The functions `band`, `bor`, `bxor` and `bnot` operate bitwise on integers; `shl` and `shr` shift logically, `asr` shifts arithmetically, preserving the sign:

```
println(band(0F0FH, 00FFH)) // 15
println(bor(1, 4))           // 5
println(bxor(5, 1))          // 4
println(shl(1, 5))           // 32
println(shr(0FFH, 4))        // 15, logical
println(asr(-16, 2))         // -4, arithmetic
```

These are functions, not operators. C overloads `&`, `|` and `>>` on its integer types and leaves the semantics of shifting signed values partly undefined; Micron names each operation, fixes logical and arithmetic shift apart, and defines the result types: the shifts yield the unsigned type of the operand's width for `shl/shr` and the operand's signed type for `asr`. Sets remain the better choice where the bits are individually meaningful flags ([Chapter 3](#)); the bit functions serve where the value is genuinely a machine word, as in checksums, hashes and device protocols.

12.2. Reinterpretation: cast and usig

[Chapter 3](#) separated the signed and unsigned families and introduced `val` for value conversion. Low-level code often needs the other conversion, the one that keeps the bits and changes the label:

```

u1 := 0FFFFFFFFH
println(cast(int32, u1))      // -1, same bits
u := usig(-1)
println(u)                   // 4294967295, same bits

```

`cast(T, x)` reinterprets the bits of `x` as type `T`; both types must have the same byte width, so a cast never truncates or extends silently. `usig(x)` is the shorthand for casting to the unsigned type of the same width. The pair replaces the union trick and the pointer-cast trick of C with one explicit, width-checked operation.

12.3. Addresses and memory-mapped devices

A pointer constructor with an address expression yields a pointer to a chosen location. This is how the Micron System talks to its devices; the module `SYS` writes a character to the RS232 debug console like this:

```

procedure TRACE*(ch: char)
  type Int32Ptr = pointer to int32
begin
  Int32Ptr{MIC.rs232data}^ := +ord(ch)
end TRACE

```

`Int32Ptr{MIC.rs232data}` makes a pointer from the device address, a named constant, and the dereference `^` becomes a store to the device register. The SPI driver in the system's Kernel module reads and writes the disk the same way:

```

Int32Ptr{spiCtrl}^ := CARD0
Int32Ptr{spiData}^ := n
repeat
  ctrl := Int32Ptr{spiCtrl}^
until ((ctrl mod 2) # 0)
data := Int32Ptr{spiData}^

```

The idiom is exactly C's `(volatile int)0xFF04 = n`, minus the cast chain and with the address kept as a typed constant. There is no `volatile` keyword; how repeated device accesses interact with optimization is a property of the compiler rather than of the language, and `micc` emits them as written. Records with bits fields ([Chapter 6](#)) complete the picture for registers with packed layouts.

Pointer arithmetic exists, but it is spelled out. `ptroff(p, n)` advances a pointer by `n`

elements of its base type, and `inc/dec` on a pointer move by one element:

```
a[0] := 10; a[1] := 20
p := @a[0]
p := ptrdiff(p, 1)      // advance by one int32
println(p^)             // 20
```

Nothing here is hidden, and nothing happens through operator overloading: the reader can find every address computation by searching for three names.

12.4. Registers, interrupts and level 0

Below memory lies the processor itself. `getreg(r, v)` and `putreg(r, v)` copy between a register, named by a constant number, and a variable; `cli()` and `sti()` disable and enable interrupts; `nop()` emits a no-operation for timing loops and pipeline flushes. All of them are restricted to levels 0 and 1. The system's garbage collector uses `getreg` to capture the callee-saved registers as roots:

```
procedure spillRegs*(p: ^array of uint32)
begin
  getreg(1, p[0]) // ra
  getreg(2, p[1]) // sp
  getreg(8, p[2]) // s0
  // ... s1 through s11
end spillRegs
```

Level 0 is the floor of the language: no stack, no heap, no recursion, every procedure implicitly `inline`. It exists for the moments before a machine is a machine, when the stack pointer is not yet set and calling a regular procedure is impossible. A boot sequence written at level 0 is a sequence of expanded procedure bodies that initialize the processor, the memory controller and the stack, and then jump into level 1 code. What was traditionally a few pages of assembler becomes Micron with the same portability properties as assembler, which is to say none, but with the same syntax, types and compiler as the rest of the system.



at the time of this writing, the available compiler backends still assume the presence of a stack; so for some systems, a bit of assembler is still required to set the CPU in the right mode and for

some other features (SMP, some interrupt handlers, etc.).

12.5. The discipline of levels

The point of these features is not that they are novel; C has had their equivalents, in rawer form, for fifty years. The point is containment. In C, any function in any file may cast an integer to a pointer and write through it; the reader who wants to rule it out must read everything. In Micron, a module using device addresses or registers is at level 0 or 1, and a module at level 3 cannot import it upward, so the unsafe operations sit in a layer that is closed from below and visible in every module heading. [Part III](#) shows the payoff at system scale: 27 modules, of which a handful touch hardware, and the headings alone document which ones.

Chapter 13. Interfacing with C

No systems language starts in an empty world. The operating systems, libraries and device vendors of this world usually speak C, and a language that cannot call into that world confines itself to its own islands. Micron interoperates with C at the level that matters, the calling convention, without adopting C's preprocessor, headers or type laxity. This chapter shows the mechanism and states its scope precisely.

13.1. Extern procedures

A procedure declared `extern` has no body in Micron; its implementation lives elsewhere. Plain `extern` means the implementation is another Micron object; `extern C` means a foreign function with C calling convention, and an optional string names the symbol when it differs from the Micron name:

```
procedure abs(x: int32): int32; extern C
procedure toupper(c: int32): int32; extern C
procedure write(fd: int32; const buf: ^[]char; n: uint32): int32
    extern C "write"
```

The declaration carries full Micron types, so every call is checked like a call to a native procedure; the trust boundary is the declaration itself, exactly as with a C prototype, but written once instead of included textually everywhere. `extern` procedures require level 1, so a module's heading again reveals that it depends on foreign code.

13.2. What maps, and what does not

Micron's basic types map directly onto the C types of the target: the fixed-width integers onto `int8_t` through `uint64_t`, `char` onto `char`, the floating point types onto `float` and `double`, and pointers onto pointers. A `^[]char` parameter matches a C `char*` pointing to a terminated string; a record type declared in Micron matches a C `struct` with the same members in the same order. The example passes a character array to `write` this way.

Micron does not support the full C ABI, and says so rather than approximating silently. The principal restriction is structured returns: a Micron procedure cannot receive a

`struct` returned by value from C, and C code cannot receive one from Micron. The workaround is the one C compilers use internally anyway: pass a pointer to the result as the first argument.

```
// C: struct div_t div(int num, int den); -- not callable directly
// C wrapper: void div2(div_t* res, int num, int den) { *res = div(num,den); }
type DivT = record quot, rem: int32 end
procedure div2(res: ^DivT; num, den: int32); extern C
```

Variadic functions are the second deviation: Micron has no variable argument lists (Chapter 5), so `printf` and friends are not callable; a fixed-arity wrapper on the C side, or the pointer-plus-count conventions of Chapter 5, take their place. A general wrapper cannot forward a run-time argument array to `printf`, because C varargs are fixed at each call site; what it can do is parse the format string itself, as Chapter 5's Micron formatter does, and dispatch each specifier to a fixed-arity call:

```
void format(const char* fmt, void** args)
{
    int a = 0;
    while (*fmt) {
        if (*fmt == '%') {
            switch (*++fmt) {
                case 'd': printf("%d", *(int32_t*)args[a++]); break;
                case 's': fputs((const char*)args[a++], stdout); break;
                case 'c': putchar(*(char*)args[a++]); break;
                case '%': putchar('%'); break;
                /* ... further specifiers follow the same pattern ... */
            }
        } else
            putchar(*fmt);
        fmt++;
    }
}
```

One such adapter serves every format string composed of the supported specifiers, and Micron passes it the same (format, argument array) pair as the pure Micron version, so the choice between the two is a build decision, not a design one. Finally, the compiler option `--abi c` selects a convention *close to* the platform's C ABI on each target, not a guarantee of bit-for-bit conformance in every corner; the cases above are the documented boundary.

13.3. Building against C code

`micc` links foreign code directly: object files and archives are given as positional arguments, and `-L/-l` name library directories and archives in the usual way.

```
micc --target x86 -L /usr/lib32 -l c crt.o Main.mic
```

There is no header translation step because there are no headers to translate; the Micron declarations of the foreign procedures are the interface. For a large C library, writing those declarations by hand is tedious, and the companion tool `c2mil` goes further than a binding generator: it translates C source, headers and implementations alike, into MIL, the Micron intermediate language (see [Appendix D](#) on MIL), as `.mil` files or relocatable objects with integrated MIL interface, which the Micron compiler uses like any other module. The reverse direction also works: a Micron module compiled with `-c` yields an ordinary object file, and its exported procedures are callable from C when their signatures stay within the mapped subset.

13.4. Separate compilation

Interoperability includes one's own past: a library should not have to be recompiled by every client. Micron modules compile separately by design. `micc -c` produces a `.mob` object file per module, which carries the module's compiled code together with its interface; a later compilation that imports the module finds the `.mob` through the import path and links it without seeing the source. Source is preferred over prebuilt artifacts when both are present, so a checkout with sources behaves as expected, and a distribution without them works too. The mechanism is Wirth's symbol-file idea in one file instead of two, and it is what makes the one-command build of the Micron System possible: `micc` is handed `Main.mic` and finds, compiles or reuses the other 26 modules by itself.

13.5. The moral

C interoperability in Micron is a border crossing, not a merger. Foreign procedures are declared explicitly, typed fully, confined to level 1 and higher modules, and checked at every call; what cannot be expressed, structured returns and variadic calls, is excluded

openly with stated workarounds (for variadic interfaces, an array plus a count on the Micron side, as shown in [Chapter 5](#)). Programs get the entire C ecosystem at the cost of a declaration per function, and the rest of the program remains as checkable as it was.

Chapter 14. The Oakwood Modules

Every program in this book so far has produced its output with `print` and `println`, the two predeclared procedures that exist precisely so that the early chapters could show programs without first introducing a library. Real programs need more: files, command line arguments, formatted output, elementary functions, a clock. Micron's compiler ships with a small set of library modules for these needs, kept in a directory called `oakwood`, and this chapter introduces them.

One caution before the tour. These modules are a working set, not a finished standard library; the standard library of Micron is still under design and will grow in a directory of its own. What the oakwood modules offer is the minimum a practical program needs today, with interfaces stable enough to build on, and everything this chapter shows about using them, the declaration conventions, the result conventions, the interplay with `extern`, carries over to whatever library comes later.

Why "oakwood"?

The name comes from the *Oakwood Guidelines for Oberon-2 Compiler Developers* of 1995 [R10], named after the hotel where the compiler writers met. The guidelines standardized the library modules that live *outside* an Oberon System, so that ordinary programs could be ported between compilers: `In`, `Out`, `Files`, `Strings`, `Math` and a few more. Oberon+ adopted these interfaces nearly verbatim; Luon already deviated; and Micron, with its focus on embedded targets and its C affiliation, kept the names but redesigned much of the substance. About a third of what follows is still Oakwood: `Out` and `Math` are close to the guidelines, `Files` has become a C-like handle interface, and `Args`, `Time` and `Screen` (the subject of [Chapter 15](#)) never were Oakwood at all. The directory keeps its name as a record of that lineage.

14.1. Interface modules and their implementations

The oakwood modules demonstrate the extern mechanism of [Chapter 13](#) in production: each module is a Micron source file containing only declarations, every procedure marked `extern`, and the implementation lives in C (for instance `Files+.c` beside `Files.mic`) or, when running under `micc`'s interpreter, in built-in procedures of the interpreter itself. The same interface file therefore serves a native build and the interpreted mode. Programs import the modules as usual and `micc` finds them through the include path:

```
micc --run -I <oakwood> Program.mic
```

A word on appearance: the oakwood sources are written with keywords in upper case, `MODULE`, `PROCEDURE`, `EXTERN`, as Oberon tradition demands and as their Oakwood descent suggests. Micron accepts both spellings, as introduced in [Chapter 3](#); this book's examples stay in lower case, and both sides compile unchanged.

14.2. Out: formatted output

`Out` writes human-readable text to standard output. Its interface is the Oakwood one, minus the historical `Open`:

```
PROCEDURE Char (ch: CHAR) EXTERN;  
PROCEDURE String (const str: POINTER TO ARRAY OF CHAR) EXTERN;  
PROCEDURE Int (i: INT64; n: INT32) EXTERN;  
PROCEDURE Real (x: REAL; n: INT32) EXTERN;  
PROCEDURE LongReal (x: LONGREAL; n: INT32) EXTERN;  
PROCEDURE Ln EXTERN;
```

The second parameter `n` is the minimum field width, with `0` meaning "as wide as needed"; `Ln` ends the line. The difference to `println` is control: `println` takes one value and always ends the line, `Out` composes a line from pieces. The quadratic solver below prints both roots on one line this way. Note also what `Out` does not have: no `Open`, no error result. Output to a standard stream either works or the program has bigger problems; a library should not make the common case verbose.

14.3. Math and MathL

`Math` declares the elementary functions over `REAL`: `sqrt`, `exp`, `ln`, `power` and `log` (both with the base as second parameter), `round`, the trigonometric family `sin` through `arctan2`, and the hyperbolic family, plus the constants `pi` and `e`. `MathL` is the same interface over `LONGREAL`. The split by precision instead of overloading is Oakwood heritage and fits Micron anyway: the language has no overloading, so a procedure's argument and result types are fixed in its declaration.

A worked example, the numerically unexciting but familiar quadratic equation:

```
procedure solve(a, b, c: real)
  var d, x1, x2: real
begin
  d := b*b - 4.0*a*c
  if d < 0.0 then
    Out.String("no real roots") Out.Ln
    return
  end
  d := Math.sqrt(d)
  x1 := (-b + d) / (2.0*a)
  x2 := (-b - d) / (2.0*a)
  Out.Real(x1, 0) Out.Char(' ') Out.Real(x2, 0) Out.Ln
end solve
```

Called with `solve(1.0, -3.0, 2.0)` this prints the roots 2 and 1; a negative discriminant takes the early exit. The complete program is in the book sources as `Roots.mic`.

14.4. Args: the command line

`Args` is as small as its job:

```
PROCEDURE Count (): UINT32 EXTERN;
PROCEDURE Arg (i: UINT32): String EXTERN;
```

`Count()` includes the program name, which is argument 0, as in C's `argv`; `Arg` returns a pointer to a runtime-managed character array, or `nil` for an index out of range. Where a C programmer writes `argc/argv` and a Pascal programmer `ParamCount/ParamStr`, the Micron programmer imports a module; the information is the same.

14.5. Files: a C-like file interface

Files is where Micron departs furthest from Oakwood. Wirth's original Files module implements the rider concept: a file is a sequence of bytes, and any number of riders, positioned read/write heads, move over it. Micron's Files instead wraps the C file model that every host system provides: a File is an opaque handle with one implicit position, obtained from Open and returned to the system with Close.

TYPE

```
File = POINTER TO Handle;  
Mode = (ReadOnly, Create, Modify)
```

```
PROCEDURE Open (name: POINTER TO ARRAY OF CHAR; mode: Mode): File EXTERN;  
PROCEDURE Close (f: File) EXTERN;  
PROCEDURE System (what: Std): File EXTERN;
```

The three modes map directly onto C's "rb", "wb+" and "rb+": ReadOnly opens an existing file for reading, Create truncates or creates for reading and writing, Modify opens an existing file for reading and writing without truncation. Open returns nil on failure, and System yields handles for the three standard streams (Stdin, Stdout, Stderr), so the same read and write procedures serve files and console alike.

The read and write procedures follow one rule worth stating explicitly: *operations transferring a value of known size return BOOLEAN; operations transferring a variable amount return the transferred count as UINT32, with 0 indicating failure or end of file.*

```
PROCEDURE Read (f: File; x: POINTER TO BYTE): BOOLEAN EXTERN;  
PROCEDURE ReadInt (f: File; x: POINTER TO INT32): BOOLEAN EXTERN;  
PROCEDURE ReadBytes (f: File; x: POINTER TO ARRAY OF BYTE; n: UINT32): UINT32 EXTERN;  
PROCEDURE Write (f: File; x: BYTE): BOOLEAN EXTERN;  
PROCEDURE WriteInt (f: File; x: INT32): BOOLEAN EXTERN;  
PROCEDURE WriteBytes (f: File; x: POINTER TO ARRAY OF BYTE; n: UINT32): UINT32 EXTERN;
```

A single byte or an INT32 either transfers completely or not at all, hence BOOLEAN; a block of n bytes may transfer partially, hence the count. The same families exist for INT64, REAL, LONGREAL, SET and BOOLEAN; the typed variants read and write the value's memory representation, so they are for binary files whose producer and consumer agree on the format. Around them, Length, Pos and Set handle positioning with UINT32 byte positions, Eof tests for end of file, Delete and Rename manipulate the directory, and OpenDir/ReadDir/CloseDir enumerate one.

Two worked utilities show the style. The first is the classic copy program, argument handling included:

```
begin
  if Args.Count() < 3 then
    Out.String("usage: Copy from to") Out.Ln
    return
  end
  src := Files.Open(Args.Arg(1), Files.ReadOnly)
  if src = nil then Out.String("cannot open input") Out.Ln return end
  dst := Files.Open(Args.Arg(2), Files.Create)
  if dst = nil then Out.String("cannot create output") Out.Ln return end
  while Files.Read(src, @b) do
    if not Files.Write(dst, b) then
      Out.String("write error") Out.Ln
      return
    end
  end
  Files.Close(dst)
  Files.Close(src)
end Copy
```

The copy loop is the byte-at-a-time idiom from Wirth's books, and the C library underneath buffers it, so it is less naive than it looks; `ReadBytes` and `WriteBytes` transfer blockwise when it matters. The second utility counts bytes and lines, a fragment of `wc`:

```
procedure count(name: ^[]char)
  var
    f: Files.File
    b: byte
    bytes, lines: int32
begin
  f := Files.Open(name, Files.ReadOnly)
  if f = nil then Out.String("cannot open file") Out.Ln return end
  bytes := 0 lines := 0
  while Files.Read(f, @b) do
    inc(bytes)
    if b = 10 then inc(lines) end // 10 = line feed
  end
  Files.Close(f)
  Out.Int(bytes, 0) Out.String(" bytes, ")
  Out.Int(lines, 0) Out.String(" lines") Out.Ln
end count
```

One detail in `count` deserves attention because it catches programmers arriving from managed languages: `bytes` and `lines` are explicitly zeroed before the loop. Local

variables in Micron are *not* initialized automatically (only level 5 initializes pointer values, to nil); a forgotten initialization compiles fine and delivers whatever the stack held. The test program around `count` writes its own input file first, using `Files.Write` per character, and deletes it afterwards with `Files.Delete`.

14.6. Time: measuring durations

The `Time` module replaces Oakwood's `Input.Time` and does exactly one thing:

```
CONST TimeUnit = 1000000;  
PROCEDURE Ticks (): UINT64 EXTERN;
```

`Ticks` returns the number of microseconds (`TimeUnit` counts per second) elapsed since the application started; the counter is monotonic and starts at zero, so it measures durations and is not a clock. The result type `UINT64` is no luxury: a microsecond counter in 32 bits wraps after 71 minutes, silently, in the middle of exactly the kind of long-running measurement it exists for. The pattern is the obvious one:

```
begin  
  start := Time.Ticks()  
  s := 0  
  for i := 1 to 1000000 do s := s + i mod 7 end  
  stop := Time.Ticks()  
  Out.Int(s, 0) Out.String(" in ")  
  Out.Int(val(int64, stop - start), 0) Out.String("us") Out.Ln  
end Timing
```

This is the same pattern the `Are-we-fast-yet` harness uses around every benchmark: two `Ticks()` calls, a subtraction in the unsigned domain, and the difference converted once for printing.

14.7. What is deliberately missing

Readers coming from Oberon will miss `In` and `Strings`; readers coming from C will miss `printf` and `malloc`. None of these is an oversight. Oakwood's `In` reads from "the current input", a concept from the Oberon System where input is a marked text; on a host system `Files.System(Stdin)` and the `Files` read procedures do the job without a second convention and without `In`'s global 'Done' flag. Oakwood's `Strings` assumes arrays that

know their length, which Micron arrays do not ([Chapter 6](#)); a string library for Micron must be designed with explicit capacities and is part of the standard library work. `printf` needs variadic procedures, which Micron does not have ([Chapter 5](#)); composing output from typed `Out` calls is the replacement. And memory management needs no library at all: it is in the language, at the levels described in [Chapter 8](#).

The sum is characteristic of the language it serves: a handful of small modules, each an ordinary interface file the reader can open and understand in a minute, no hidden runtime behind them, and a stated rule where uniformity matters, as in the `Files` result convention. The next chapter uses the largest of the modules, `Screen`, to put interactive graphics on the same foundation.

Chapter 15. Graphics with Screen

The Screen module is the largest of the oakwood set; it is a replacement and extension of the Oakwood XYplane and Input modules. It gives a program one window, a monochrome framebuffer, raw input events, mouse polling, a cursor and a millisecond timer; on the host it is implemented against SDL, XCB or Qt, behind the same Micron interface. That short list is enough for an interactive application, and this chapter builds two: a sketch pad and Barnsley's fractal fern, the closing example of Wirth's *Programming in Oberon*, here in Micron.

The design is worth a moment before the procedures. Screen does not offer drawing operations, no lines, no fills, no text. It offers *memory*: the program owns a byte array, tells Screen to treat it as the picture, changes bytes, and reports which region changed. Everything a reader learned about bit manipulation in [Chapter 12](#) becomes directly useful, and everything a graphics library would hide, pixel format, row layout, dirty regions, is in plain sight. This is the same discipline as the memory-mapped display of the Micron System and of historical machines from the Alto onward, which is precisely why the module exists: the Smalltalk-80 VM of [\[TBD\]](#) renders through Screen, and so does the Oberon System example when it runs directly on the host.

15.1. The framebuffer

One bit per pixel: a byte holds eight horizontal neighbours, a row of width W occupies $W \div 8$ bytes (the *stride*), and the buffer holds $\text{stride} * H$ bytes for H rows. The program allocates this buffer itself, as an ordinary array:

```
const
  W = 512
  H = 512
  Stride = W div 8    // bytes per framebuffer row

var fb: array Stride * H of byte
```

Two flags in Open set the interpretation, because history left two conventions of each kind: `LsbFirst` makes bit 0 of each byte the leftmost pixel (as in the Oberon System), its absence makes bit 7 leftmost (as in Smalltalk-80); `BottomUp` puts row 0 at the bottom edge (Oberon) instead of the top (Smalltalk). With MSB-first, top-down layout, the

empty flag set, setting a single pixel is the bit arithmetic of [Chapter 12](#):

```
procedure plot(x, y: int32)
  var i: int32
begin
  if (x < 0) or (x >= W) or (y < 0) or (y >= H) then return end
  i := y * val(int32, Stride) + x div 8
  fb[i] := val(byte, bor(val(uint32, fb[i]), shl(val(uint32, 1), usig(7 - x mod 8))))
end plot
```

The byte index combines row offset and $x \text{ div } 8$; the mask puts a one at bit $7 - x \text{ mod } 8$, counting from the left; `bor` merges it into the byte, and the `val` conversions move between the `byte` element type and the `uint32` domain of the shift. Bounds checking in `plot` rather than at every call site keeps callers simple.

15.2. Window, dirty rectangles, and the loop

Three procedures manage the window:

```
PROCEDURE Open (buf: POINTER TO ARRAY OF BYTE; bufLen, w, h: INT32;
  flags: SET): BOOLEAN EXTERN;
PROCEDURE Close EXTERN;
PROCEDURE UpdateArea (x, y, w, h: INT32) EXTERN;
```

`Open` hands the buffer to the module and creates the window; it may be called again to resize or change flags, and returns `FALSE` if the platform refuses. `UpdateArea` does *not* draw; it marks a rectangle dirty. Rendering happens inside `ProcessEvents`, which a running program calls anyway:

```
PROCEDURE ProcessEvents (sleep: INT32): INT32 EXTERN;
```

`ProcessEvents` polls the host's event system, waits at most `sleep` milliseconds, copies dirty framebuffer regions to the window, and queues raw input events. Its result encodes three cases: `-1` means the user asked to quit (closed the window), `0` means "nothing new", and a positive count reports queued events. The separation of "mark dirty" from "render" means a program can set thousands of pixels and pay for one texture upload per loop iteration, the standard economy of framebuffer graphics.

15.3. Events as packed integers

Input arrives as INT32 values from a queue:

```
PROCEDURE NextEvent (): INT32 EXTERN;
```

`NextEvent` returns 0 when the queue is empty; otherwise bits 27..24 hold the event type and bits 23..0 the data, mouse coordinates, a button code, or a key code, depending on the type. The type codes are constants of the module, `EvtMouseMove` through `EvtKeyChar`, as are the special key codes (`KeyEscape`, `KeyReturn`, arrow keys and so on; printable keys report their ASCII value). Unpacking is two masks:

```
kind := val(int32, band(shr(usig(evt), 24), 0FH))
data := band(evt, 0FFFFFFH)
```

A packed event in one machine word instead of a record may look austere, but it is deliberate and has precedent: the event queue costs no allocation, fits any transport, and the Oberon System's `Input` module and the Smalltalk VM decode the same kind of word. For programs that prefer state polling over events, `GetMouseState` returns the current position through out-pointers and the button state as a bitmask (`MouseLeft`, `MouseMid`, `MouseRight`), matching the style of the Oberon System's mouse handling.

The remaining procedures round off the window: `SetCursor` installs a custom 1bpp cursor image with hotspot, `SetCursorPos` warps the pointer, and `GetTicks` returns milliseconds since the module started, a coarser sibling of `Time.Ticks` for animation pacing.

15.4. A sketch pad

The smallest interactive program uses both input styles at once: events for the keyboard, polling for the mouse.

```
procedure run()
  var
    res, evt, kind, x, y, buttons: int32
    quit: boolean
  begin
    if not Screen.Open(@fb, val(int32, Stride * H), val(int32, W), val(int32, H), {}) then
      Out.String("cannot open window") Out.Ln
      return
```

```

end
quit := false
repeat
  res := Screen.ProcessEvents(10)
  evt := Screen.NextEvent()
  while evt # 0 do
    kind := val(int32, band(shr(usig(evt), 24), 0FH))
    if (kind = Screen.EvtKeyDown) and
      (band(evt, 0FFFFFFH) = Screen.KeyEscape) then
      quit := true
    end
    evt := Screen.NextEvent()
  end
  buttons := Screen.GetMouseState(@x, @y)
  if band(buttons, Screen.MouseLeft) # 0 then
    plot(x, y)
    Screen.UpdateArea(x, y, 1, 1)
  end
until (res < 0) or quit
Screen.Close
end run

```

The loop shape is the one every `Screen` program has: `ProcessEvents` with a small sleep bounds the loop frequency; the inner `while` drains the event queue; the poll acts on current state. Painting sets one pixel and marks a one-pixel rectangle dirty, which is honest bookkeeping rather than pessimism, since `Screen` accumulates dirty rectangles until the next render. Escape sets `quit`, the window's close button makes `ProcessEvents` return `-1`, and both paths leave through the same `until`.

15.5. The fractal fern

Wirth ends *Programming in Oberon* with Barnsley's fern, drawn by an iterated function system: a point hops through the plane, and each hop applies one of four affine maps, chosen at random with fixed probabilities. The attractor of this random walk is the fern. The maps and probabilities are Barnsley's:

```

procedure fern(points: int32)
  var
    x, y, xn, yn, r: longreal
    i: int32
begin
  x := 0.0 y := 0.0
  for i := 1 to points do
    r := random()

```

```

    if r < 0.01 then
        xn := 0.0
        yn := 0.16 * y
    elseif r < 0.86 then
        xn := 0.85 * x + 0.04 * y
        yn := -0.04 * x + 0.85 * y + 1.6
    elseif r < 0.93 then
        xn := 0.2 * x - 0.26 * y
        yn := 0.23 * x + 0.22 * y + 1.6
    else
        xn := -0.15 * x + 0.28 * y
        yn := 0.26 * x + 0.24 * y + 0.44
    end
    x := xn y := yn
    plot(val(int32, floor(x * 48.0)) + val(int32, W) div 2,
        val(int32, H) - 1 - val(int32, floor(y * 48.0)))
end
end fern

```

The scale factor 48 and the offsets place the fern’s coordinate range, roughly -2.2..2.7 in x and 0..10 in y, on the 512-pixel canvas, with y inverted because our layout is top-down. Two hundred thousand points suffice for a dense image.

The one ingredient Micron does not provide is randomness, so the program brings its own, the classic linear congruential generator in ten lines:

```

procedure random(): longreal    // uniform in [0, 1)
begin
    seed := seed * 1664525 + 1013904223
    return flt(val(int32, shr(seed, 8))) / 16777216.0
end random

```

The multiplication overflows `uint32` routinely, which is exactly the intended arithmetic: unsigned types compute modulo 2^{32} by definition ([Chapter 3](#)), so the generator needs no masking. Discarding the low bits (`shr(seed, 8)`) matters with an LCG, whose low bits cycle short; the top 24 bits divided by 2^{24} give a uniform value in $[0, 1)$.

The main program computes all points into the framebuffer first and only then opens the window, marks everything dirty once, and enters the event loop waiting for Escape or the close button.



The complete program is 90 lines. That is the measure of the module: an interactive graphical application, event handling included, fits in a chapter listing, with no framework, no callbacks to register, no state machine imposed from outside. The program owns the loop, the pixels and the timing, which is how the systems in [Part III](#) want to be written, and the reader who has followed since [Chapter 12](#) already knew every operation used to get the fern on the screen.

Chapter 16. Stepwise Refinement and Data Abstraction

TBD

Chapter 17. Error Handling, FINALLY and Exceptions

TBD

Part III: Systems Programming — The Micron System

Present the Micron System, a port of Niklaus Wirth and Jürg Gutknecht's Project Oberon (edition 2013) to Micron.

Chapter 18. The Machine and the Build

[Part III](#) presents the Micron System, the port of Project Oberon (Wirth and Gutknecht, edition 2013) that this book uses as its running systems example. The chapters follow the system bottom-up, from the machine to the applications, and each stays close to the corresponding chapters of Wirth’s book, which remains the reference for the design; here the subject is what Micron makes different. The port’s own `Design.md`, kept with the sources, records every deviation; this part of the book draws on it and credits it once for all.

18.1. The machine

The target is the RV32 machine of the Project Oberon [\[R2\]](#) emulator: a 32-bit RISC-V core in place of Wirth’s RISC-5 processor (his own design, not to be confused with RISC-V; [Appendix C](#) relates the two), with the original machine’s devices at their original addresses: the millisecond timer, the SPI interface to the SD card, LEDs, RS232, mouse and keyboard, all memory-mapped in the top page of the address space. [Chapter 12](#) showed how such devices are reached with pointer constructors; the Kernel’s constants name them:

```
const
  timer = 0FFFFFFC0H
  spiData = 0FFFFFFD0H
  spiCtrl = 0FFFFFFD4H
  LEDadr = 0FFFFFFC4H
```

One thing moved: Wirth’s memory map is fixed, with the display frame buffer at `0E7F00H` and 164 KB left for the heap, too little for the conservative collector of the next chapter. The emulator therefore takes a `--ram <MB>` option, places the frame buffer at the top of RAM, and passes the layout to the software in a small boot header at the bottom of memory: the heap origin at address `08H`, the origin of the writable data at `0CH`, and the end of usable RAM at `010H`. `Kernel.Init` reads its world from that header rather than from constants:

```
procedure Init*()
begin
  MemLim := Int32Ptr{010H}^
  dataOrg := AnyPtrPtr{dataOrgAdr}^
```

```

heapOrg := AnyPtrPtr{08H}^
...
end Init

```

With `--ram 1` the layout coincides bit for bit with Wirth's and an original Project Oberon System binary and disk image still boot; the system itself runs with 4 MB, which yields about 3.2 MB of heap. The disk is a plain image file served by the emulator over the SPI device, so the Kernel's `Fsoffset` is 0 instead of the `80000H` with which the original skips the FAT partition of a real SD card.

18.2. One command, one file

Wirth's system is built in two stages. `ORP` compiles each module to an `.rsc` object file; a boot linker prelinks the inner core into a boot image loaded from the disk's boot sectors, and `Modules` loads the outer core dynamically, module by module, on first import.

The Micron System is built in one stage. `micc` compiles and links all 27 modules into a single statically linked, freestanding ELF file, with no C library involved:

```

micc --target rv32 -g --base 0 --runtime MIC+.mil -I . -o Main Main.mic
rv32vm --elf --ram 4 --disk disk.img Main

```

The compiler follows the imports from `Main` down, compiles all modules, links, and the emulator loads the result directly. What the two-stage build achieves with a boot image and a loader, the one-stage build gets from a language rule ([Chapter 5](#)): module bodies run in import dependency order. The bodies of `Kernel`, `Files`, `Display` and the rest initialize the system exactly as the boot linker's generated initialization sequence did, and no separate boot format is needed.

18.3. Main as the boot linker

`Main` imports every module of the outer core and is therefore the last body to run; it plays the role of Wirth's boot linker. Its body does two things: it registers the commands, and it enters the central loop.

```

module Main [level=5]

```

```

import Modules, Oberon, System, Edit, Curves, Rectangles, Draw,
    GraphTool, Hilbert, Sierpinski, Stars, Hennessy

begin
    Modules.Register("System", "Date", System.Date)
    Modules.Register("System", "Watch", System.Watch)
    // ... some ninety registrations ...
    Modules.Register("Edit", "Open", Edit.Open)

    Oberon.Loop()
end Main.

```

In Wirth's system a command is a name resolved by the module loader: clicking `Edit.Open` makes `Modules` find the module's object file, load it if necessary, and jump to an address in its entry table. Here there is no loader and no entry table, and the language provides the natural replacement: a command is a procedure type,

```
Cmd* = procedure() // a command procedure
```

and `Modules.Register` stores name and value in one table, `cmdTab`:

```

procedure Register*(const modname, cmdname: pointer to array of char; adr: Command)
begin
    if nofCmds < MaxCmds then
        cmdTab[nofCmds].mod_ := ThisModule(modname)
        copy(@cmdTab[nofCmds].name, cmdname, 32)
        cmdTab[nofCmds].adr := adr
        inc(nofCmds)
    end
end Register

```

`Modules.ThisCommand` becomes a table lookup returning a procedure, and `System.ShowCommands` lists what was registered. The user-visible behavior is Wirth's: middle-click a name of the form `M.P` and the procedure runs. What changed is below: the binding happens at link time and startup, in ordinary Micron, instead of at run time as in the Oberon System.

`Modules` keeps its interface nonetheless, and its descriptor records, so that the correspondence to Wirth's book stays visible; `load` merely finds the descriptor of an already linked module, and `Free` removes descriptors without unloading anything.

18.4. Levels in the system

The module headings of the system show the level attribute of [Chapter 2](#) doing real work. `SYS`, `MIC`, `Kernel` and `FileDir` are declared `[level=1]`, the systems level; `Display` and `Input`, whose handlers use the object machinery, are `[level=4]`; and the rest of the system, from `Files` up to `Main`, runs at `[level=5]`, where `newgc` and the collector are available. The import rule, a module imports only modules at its own or a lower level, turns the layering of Wirth's book from a documentation convention into a checked property of the source text.

The next chapter descends into the largest deviation from the original: the Kernel's heap and the conservative collector behind `newgc`.

Chapter 19. The Kernel: Memory, Heap, Collector

The Kernel owns the machine's memory, as it does in Wirth's system, and this chapter is where the port deviates furthest from the original: `micc` generates neither module pointer tables nor type descriptors that Wirth's precise collector expects, so the Micron Kernel collects conservatively, in about 120 lines. On the way there, the chapter shows how the compiler's `new` reaches the Kernel at all, without any compiler support beyond an ordinary procedure variable.

19.1. How `new` reaches the Kernel

[Chapter 8](#) introduced the allocation procedures and [Chapter 13](#) the runtime interface: for `new`, `newinit`, `newgc` and `dispose` the generated code calls runtime procedures declared by the intrinsic `MIC$` module, instead of expanding code inline. On a hosted target the runtime sits in a C library. This machine has none, so the system implements the runtime itself, in Micron: the MIL module `MIC+.mil` declares the `MIC$` names the generated code refers to, and forwards each to the module `MIC`, written in ordinary Micron. The file is called `MIC+.mil` because a Micron module name cannot contain a dollar sign, deliberately, so no module can collide with a runtime symbol.

`MIC` is small: console output over RS232 for `println`, string comparison, set operations, `halt` and `assert`, and the allocation entry points. But `MIC` does not own the heap; the Kernel does, and the connection between the two is nothing more than two procedure variables:

```
type
  Allocator* = procedure(size: int32): AnyPtr

var
  newSys: Allocator // the heap owner installs these, see Install
  newGc: Allocator

procedure Install*(new: Allocator; gc: Allocator)
begin
  newSys := new
  newGc := gc
end Install
```

At the end of `Kernel.Init` stands `MIC.Install(newSys, newGc)`, and from then on every

allocation the compiler generates anywhere in the system runs through the Kernel. In Wirth's system the same connection needs compiler fixups resolved by the boot linker; here it is an import and a call, visible in the source. One rule keeps the arrangement sound: MIC must not itself depend on anything the generated code needs from MIC\$, or the runtime would call itself.

Two kinds of blocks result from calls to the allocation procedures. `newSys` serves `new` and `newinit`: blocks that live as long as the system, used for the Kernel's own structures, the sector buffers of `Files`, the module descriptors. `newGc` serves `newgc`: blocks the collector may reclaim; when the heap is full it collects once and retries:

```

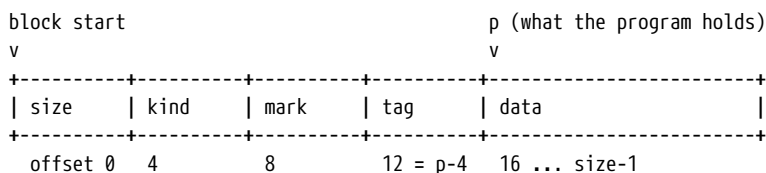
procedure newGc(size_: int32): AnyPtr
  var p: AnyPtr
begin
  New(@p, nil, GcBlk, size_)
  if p = nil then
    Collect()
    New(@p, nil, GcBlk, size_)
  end
  return p
end newGc

```

19.2. The heap: Wirth's free lists

`Kernel.New` keeps the structure of Wirth's allocator unchanged: four free lists for blocks of 32, 64, 128 and `n-256` bytes, where a request is rounded up to the next size, larger free blocks are split, and a missing small block is made by splitting the next larger one (`GetBlock32` asks `GetBlock64` asks `GetBlock128` asks `GetBlock`). Every block carries a 16 byte header: its size, its kind (`SysBlk`, `GcBlk` or `FreeBlk`), its mark, and a tag word that keeps Wirth's layout expectation, a type tag at `ptr-4`, describable for a future precise collector. The header is the same for all kinds of block, so Wirth's three trap entry points `NewRec`, `NewArr` and `NewSys` collapse into the one procedure `New`, reached through the callbacks.

A heap block; the pointer the program holds refers past the header



SysBlk,	0	unmarked	kept for
GcBlk or	1	reached	Wirth's
FreeBlk	2	done	layout

The header fields are read and written through the pointer arithmetic of [Chapter 12](#), wrapped in inline procedures so the offsets appear once:

```

procedure blkKind(p: AnyPtr): int32 inline
begin
  return cast(Int32Ptr, ptroff(p, 4))^
end blkKind

```

19.3. The conservative collector

Wirth's collector is precise: `Kernel.GC(roots)` marks from a table of global pointer offsets that the boot linker generates for each module, and follows the pointer offsets recorded in each object's type descriptor. `micc` generates neither table, and extending the compiler for one example system was not worth it. The Micron Kernel therefore collects conservatively, in the style of the Boehm collector for C: anything that looks like a pointer is treated as one.

The roots are three memory areas, marked word by word: the spilled registers (below), the static data of all modules, and the whole stack, dead frames included, so no stack pointer is needed:

```

procedure Collect*()
  var regs: array NofRegs of uint32
  ...
begin
  inc(Collections)
  SYS.spillRegs(@regs) // a pointer may be in a register; must be the first statement
  mapBlocks()
  markTop := 0
  markOverflow := false
  markArea(cast(AnyPtr, @regs[0]), cast(AnyPtr, ptroff(@regs[0], NofRegs)))
  markArea(dataOrg, ptroff(stackOrg, -stackSize)) // the global variables of the modules
  markArea(ptroff(stackOrg, -stackSize), stackOrg) // the whole stack
  markReachable()
  ...
end Collect

```

A candidate word keeps a block alive when it addresses the block's interior, not only

its start; this is not generosity but necessity, since the glyph pointers in `Fonts` point into the middle of a block. The candidate test must therefore answer, for an arbitrary 32 bit word, which heap block contains this address, and answer it fast. `mapBlocks` rebuilds a block map at the start of every collection, recording for each 1 KB chunk of the heap the block covering its first byte; `blockOf` then indexes the map and walks forward at most 32 blocks. A block whose size is not a positive multiple of 8 ends the walk, so a word addressing a region the heap walk does not describe cannot fabricate a pseudo block, and an address inside a header does not count as a reference.

Marking uses an explicit stack of 1024 entries rather than recursion. On overflow a block simply waits: it keeps mark 1 (reached, not yet expanded) instead of mark 2 (done), and after the stack drains, the heap is rescanned for waiting blocks. Marking is therefore complete but may take several passes over the heap in the rare case of overflow, a classic space-for-certainty trade.

The sweep is Wirth's: a linear walk over the heap coalesces each run of free and unmarked `6cBlk` blocks into one area, and `freeArea` cuts each area into blocks of 32, 64, 128 and `n*256` bytes for the four free lists, exactly as the original's `Scan` does. `SysBlk` blocks are never reclaimed, whatever points at them.

The consequences are deliberate and worth stating plainly. A word that only looks like a heap address retains a block; the collector is conservative, and no claim beyond that is made. In exchange, no type information, no pointer maps and no compiler support are needed, and the whole collector, map, mark and sweep, fits in about 120 lines of the same language it collects.

19.4. Registers as roots

A conservative collector must also see pointers the compiler keeps in registers. `SYS.spillRegs` stores `ra`, `sp` and `s0` to `s11`, the registers of the RV32 calling convention that a callee must preserve, which are the only ones that can hold a pointer across a call, into fourteen words the collector passes in; `Collect` marks that area first. The procedure is written with `getreg` ([Chapter 12](#)), so `SYS` needs no assembler, no MIL and no `extern`. One subtlety shows the reasoning that low-level Micron requires: since `spillRegs` is a real call, `getreg` may read the callee's saved `s0` rather than the caller's; the caller's value is then in `spillRegs` own stack frame, which is part of the stack the collector scans

anyway.

19.5. When collection runs

As in the original, the collector runs as a task of the central loop ([TBD]), between commands, when the heap is nearly full or after a number of input actions; `Oberon.GC` calls `Kernel.Collect()`, and there is no root set to pass. An object reachable only from a local variable of a running command can therefore never be collected, which is what makes the whole-stack scan affordable. `System.Watch` prints the heap figures, with `Kernel.Collections` added to Wirth's numbers.

The next chapter climbs one layer: the disk, the file system, and what remains of `Modules` when there is nothing left to load.

Chapter 20. Disk, Files, Modules

Above the Kernel sit the file system and the module machinery, `FileDir`, `Files` and `Modules`. Their design is Wirth's, chapter 7 of his book describes it, and the port changes nothing in it; what this chapter describes is the Micron in it: the sector arithmetic, the serialization rewritten in the unsigned domain, and what remains of `Modules` when there is nothing left to load.

20.1. Sectors

The Kernel exports the disk as numbered sectors of 1 KB. Wirth multiplies every sector number by 29, a prime, so that a corrupted or fabricated sector number is overwhelmingly unlikely to be divisible by 29 and is caught at the next use. In the original this check is `ASSERT(SYSTEM.H(0) = 0)`, reading the remainder register of the RISC-5 divider; Micron has `mod`, so the check says what it means:

```
procedure GetSector*(src: int32; dst: pointer to array of byte)
begin
  assert(src mod 29 = 0)
  src := src div 29
  src := src * 2 + FSoffset
  ReadSD(src, cast(AnyPtr, dst))
  ReadSD(src + 1, @dst[512])
end GetSector
```

The free sectors are kept in a bitmap, an array of set, and marking a sector is a set operation, + {sec mod 32}, exactly as in the original; sets over machine words is one Oberon idea that transfers to Micron without any translation at all.

20.2. The file directory and the unsigned domain

`FileDir` implements Wirth's B-tree directory with the root page at sector 29; `Files` implements files as a head sector with a sector table, extension sectors and buffered riders. Structurally the modules correspond to the originals procedure for procedure. The interesting differences are all of one kind, and [Chapter 3](#)'s signedness discipline predicted them.

Oberon has one integer type, so Wirth's serialization moves bytes through `CHR` and `ORD`

and reinterprets freely. Micron distinguishes the signed number from the bit pattern, and the port works in the unsigned domain wherever bits are meant:

```
procedure ReadInt*(R: ^Rider; x: ^int32)
  var x0, x1, x2, x3: byte
begin
  ReadByte(R, @x0)
  ReadByte(R, @x1)
  ReadByte(R, @x2)
  ReadByte(R, @x3)
  x^ := sigc(shl(x3, 24) + shl(x2, 16) + shl(x1, 8) + x0)
end ReadInt
```

The bytes are assembled with shifts in the unsigned domain and reinterpreted once, at the end, with `sigc`. The magic numbers of the directory show the other direction:

```
DirMark*   = sigc(9B1EA38DH)
HeaderMark* = sigc(9BA71D86H)
```

9B1EA38DH does not fit in an `int32` as a number, but the stored word is a bit pattern that happens to have the sign bit set, so `sigc` states the reinterpretation where it happens, once, in the constant declaration. Which conversion to use, `sigc` to reinterpret, `sigl` or unary `+` to widen, `val` to change width, is the recurring decision when porting Oberon code, and [Chapter 21](#) shows what happens when it is made wrongly.

One allocation detail is worth noticing: `Files` allocates its buffers and file descriptors with `newgc`, so an abandoned, unregistered file is eventually reclaimed by the collector of the previous chapter, as in the original, where losing a file object is harmless by design.

20.3. Modules without a loader

In Wirth's system `Modules` is the dynamic loader: `ThisMod` reads a module's `.rsc` file, allocates its blocks, relocates its code and runs its body. In the Micron System all 27 modules are linked statically and their bodies have run before `Main` ([Chapter 18](#)), so `Modules` keeps its interface and loses its purpose:

```
procedure Load*(const name: pointer to array of char; newmod: ^Module)
  // all modules are linked statically and their bodies have run
  var mod_: Module
```

```

begin
  mod_ := root
  res := 0
  while (mod_ # nil) & (name^ # mod_.name) do
    mod_ := mod_.next
  end
  if mod_ = nil then
    error(2, name)
  end
  newmod^ := mod_
end Load

```

`Load` finds the descriptor of an already linked module and creates it on demand, since no boot linker generated one; Free removes descriptors without unloading anything; and commands live in the table `cmdTab` that `Main` filled at startup, each a procedure value rather than an entry-table address. The descriptor record `ModuleDesc` keeps all of Wirth's fields, including those only a loader would use, so the correspondence to his book remains visible. `Modules.res` is 0 on success and a small documented number otherwise, with `importing` naming the culprit.

A small Micron lesson hides in `ThisCommand`. Wirth bounds his loop with `LEN(mod.cmds^)`, the intrinsic length of an Oberon dynamic array; a Micron dynamic array is typed memory with no length ([Chapter 6](#)), so the port ends the table with a sentinel entry instead:

```

// Oberon: WHILE (res # 0) & (i < LEN(mod.cmds^)) DO; micc has no LEN for a
// dynamic array, so the command list ends with an empty name
while (res # 0) & (mod_.cmds[i].name[0] # 0X) do

```

The sentinel also gives the table a well-defined end for a future precise collector, which would need to know where the pointers stop.

20.4. What dynamic loading would take

Nothing in the language forbids a real loader; it's all in the object format. A dynamically loadable module needs its code and data in a relocatable form, a list of the addresses to fix up against the already loaded modules, and a table of its own entry points, which is what Wirth's `.rsc` format, his loader in `Modules` and the fingerprints in the symbol files provide. `micc` compiles to MIL and links statically; a port of Wirth's scheme would define an analogous format for MIL-compiled code and reinstate

`ThisFile`, whose skeleton the module still carries. The port leaves this as an outlook, deliberately: the system it serves fits in one linked image, and the book's subject is the language, not a loader.

The next chapter turns to the screen: the display driver, the input devices, and the viewer machinery above them.

Chapter 21. Display, Input, Viewers

This chapter reaches the screen. `Display` draws into the frame buffer, `Input` reads the mouse and the keyboard, and `Viewers` tiles the screen into the frames that make Oberon's user interface. It is also where the Wirth OO from [Chapter 9](#) transitions from a concept to a functional system.

21.1. The frame buffer

The display is one bit per pixel, and `Display` sees it as memory: a `Words` value, `pointer` to `set`, addresses 32 pixels of a display line. The base address comes from the boot header of [Chapter 18](#) rather than from Wirth's fixed `0E7F00H`, and everything else is address arithmetic:

```
procedure adr(x, y: int32): Words inline
  // the display word which holds the pixel x, y
begin
  return ptrdiff(Base, y * L + word(x))
end adr
```

Declaring the display word as a `set` is Wirth's insight, and it survives the port unchanged: setting a pixel is `set union`, inverting is `symmetric difference`, clearing is `set difference`, and the three drawing modes of the whole system, `paint`, `invert` and `replace`, are exactly the three operations:

```
procedure Dot*(col, x, y, mode: int32)
  var a: Words; s: set
begin
  a := adr(x, y)
  s := {obmod(x, W)}
  if mode = paint then
    a^ := a^ + s
  elseif mode = invert then
    a^ := a^ / s
  ...
end Dot
```

Two details show the arithmetic discipline of [Chapter 3](#) at work. The coordinates use `obdiv` and `obmod`, the flooring division of Oberon, because a clipped frame can hand in a negative `x`, and `-1 div 32` must be `-1`, not `0`, to land in the display word to the left. And

where a pattern line is shifted across a word boundary, the code works on `uint32` bit patterns and converts once, with `bset` to reenter the set domain:

```
a^ := a^ + bset(shl(pwd, usig(k)))
if k + w > W then // spill over
  ptroff(a, 1)^ := ptroff(a, 1)^ + bset(shr(pwd, usig(W - k)))
end
```

A `Pattern` is a `pointer to byte: width, height`, then the raster bytes. `CopyPattern` reads the header with the widening conversion, `w := patadr^`, unary`` because the first byte represents the width. This is the conversion whose confusion with reinterpretation, `sigc`, broke four modules during the port: a glyph metric, a pattern size or a key code of 128 or more silently turns negative when reinterpreted, and the symptom appears far from the cause, as garbled characters or a dead key.

21.2. Input

`Input` polls the mouse and keyboard registers of the machine:

```
procedure Mouse*(keys: ^set; x, y: ^int32)
  var w: int32
begin
  w := Int32Ptr{msAdr}^
  keys^ := bset(usig(w div 16777216 mod 8))
  x^ := w mod 1024
  y^ := w div 4096 mod 1024
  ...
end Mouse
```

The mouse state is one machine word, fields extracted with division and modulus by powers of two, as in the original. The keyboard delivers PS/2 scan codes which a table maps to characters; the lookup carries the war story of the previous paragraph in one line:

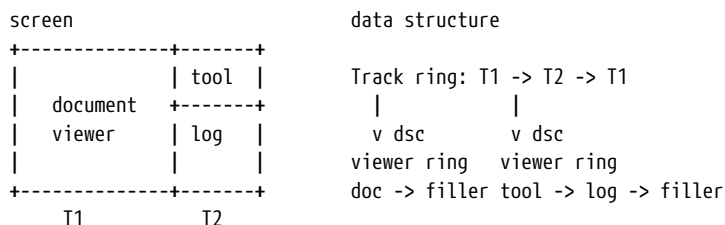
```
ch^ := cast(CharPtr, ptroff(KTabAdr, +ord(kbdCode)))^
```

The `+` widens the code to a non-negative index. With the reinterpretation instead, every code at and above 128, which `Read` produces for every shifted key, would index backwards from the table.

21.3. Viewers and the message hierarchy

Viewers tiles the screen into tracks and viewers, Wirth's design unchanged: the screen is divided into vertical *tracks*, each track into a stack of *viewers*, and no window ever overlaps another. The data structure behind the tiling is a two-level ring of frames, anchored in one filler viewer per track:

The tiled display and the frame rings that describe it



Every element of both rings is a `Frame`; a track's `dsc` field refers to its viewer ring, and heights adjust so that a track's viewers always cover it exactly, which is what makes redraws local and cheap. For this book the module's main interest is that it is the root of the system's message hierarchy, the Wirth OO of [Chapter 9](#) in production. `Display` declares the base object and the handler type:

```

FrameMsg* = object
end
Handler* = procedure(F: Frame; M: pointer to FrameMsg)
FrameDesc* = object
  next*, dsc*: Frame
  X*, Y*, W*, H*: int32
  handle*: Handler
end

```

and every layer above extends `FrameMsg` with the messages it defines: `Viewers.ViewerMsg` for restore, modify and suspend; `Oberon.InputMsg` for keys and characters. A frame knows one procedure, its handler; a handler narrows the received message with the type case of [Chapter 9](#) and ignores what it does not know. That last property is the extensibility argument: a new message type extends `FrameMsg`, and it is sent to whoever listens, and every existing handler remains correct without recompilation.

21.4. The central loop

`Oberon.Loop` is the whole scheduler of the system:

```
repeat
  Input.Mouse(@keys, @X, @Y)
  if Input.Available() > 0 then
    Input.Read(@ch)
    ...
    M.id := consume
    M.ch := ch
    FocusViewer.handle(FocusViewer, @M)
  elseif keys # {} then
    M.id := track
    ...
    V := Viewers.This(M.X, M.Y)
    V.handle(V, @M)
  else
    ...
    CurTask := CurTask.next
    t := Kernel.Time()
    if t >= CurTask.nextTime then
      CurTask.handle()
    end
  end
end
until false
```

A keystroke becomes an `InputMsg` with id `consume` sent to the focus viewer; mouse activity becomes `track` messages sent to the viewer under the cursor; and an idle loop iteration runs one background task, among them the garbage collector of [Chapter 19](#). There is no preemption and no process: a command runs to completion in the loop, which is why the collector may scan the whole stack between commands, and why a looping command hangs the system, a trade Wirth documents and accepts. The loop dispatches on procedure values from bottom to top: `CurTask.handle`, `V.handle`, and inside the handlers the type case on the message, three mechanisms from [Part II](#) and not one of them a virtual method table.

The next chapter climbs into the text system, where the same messages carry an editor.

Chapter 22. Texts, Fonts and the Editor

The text system is the largest subsystem of the Micron System, Texts, Fonts, TextFrames, MenuViewers and Edit, almost 3000 lines, and the part every user touches: in Oberon, text is the user interface. For this book it is a showcase for different language features: garbage-collected data structures, objects with inheritance used three different ways, and the model-view separation that Wirth built from one procedure variable.

22.1. Texts: the piece list

A Text never stores characters. It stores a doubly linked ring of pieces, each pointing into a file:

```
PieceDesc = record
  f: Files.File
  off: int32
  len_: int32
  fnt: Fonts.Font
  col: int32
  voff: int32
  prev: Piece
  next: Piece
end
```

Editing never moves text: an insertion splits a piece and links in new ones, a deletion unlinks a run of pieces, and the characters themselves stay in the files they came from, the original one and the unnamed scratch files that writers append typed input to. The following example makes the invariant visible: the text the user sees is the concatenation of the pieces in ring order, while the characters sit unmoved in two files. After inserting the word `not` in the middle of a sentence read from `doc`, the ring holds three pieces where one was before:

A text after one insertion: three pieces, two files, no character moved

```
shown text: "this is not character soup"

text ring: [doc, off 0, len 8] -> [scratch, off 0, len 4] -> [doc, off 8, len 14]

file doc:      "this is character soup"      (untouched)
              ^          ^
              off 0    off 8
file scratch:  "not "
```

Deleting undoes nothing in the files either; it only unlinks pieces. This is the piece list, the data structure behind many editors since; `SplitPiece` and `FindPiece`, with its one-piece cache in the `Text`, together fit on a page.

Like the file buffers of [Chapter 20](#), pieces are allocated with `newgc` ([Chapter 8](#)) and never freed, but here at a different rate: an editing session produces garbage at typing speed, and the collector of [Chapter 19](#), run between commands by the loop of [Chapter 21](#), is what makes the design affordable. Writing the same module with manual disposal would mean reference counting the pieces shared between a text and the buffers moving text around.

22.2. Readers, Scanners, Writers

Access to a text goes through objects:

```
Reader* = object
  eot*: boolean
  fnt*: Fonts.Font
  col*: int32
  voff*: int32
  ref: Piece
  org: int32
  off: int32
  rider: Files.Rider
end

Scanner* = object (Reader)
  nextCh*: char
  line*: int32
  class*: int32
  i*: int32
  x*: real
  ...
end
```

`Scanner` extends `Reader` with the fields of a lexical scanner: `Texts.Scan` reads names, numbers, strings and special characters, and it is the same scanner for everyone, the reason every Oberon command parses its parameters the same way. Note what this inheritance is for: reuse of data and procedures, not dynamic dispatch; nothing here is overridden. `Writer` accumulates output in a buffer, and its `WriteInt`, `WriteHex` and `WriteReal` are where the formatting lives that `println` provided in the hosted chapters.

The serialization of a text block reads its bytes with the conversions of [Chapter 20](#), and one line shows all three intents side by side. A font number is a small count, widened with `+`; a vertical offset is a signed byte:

```
q.col := +bt
q.voff := cast(int8, bt) // the byte is a signed offset
```

22.3. Fonts

A font file is parsed once into a `FontDesc`: metrics, a table of 128 glyph pointers, and a raster area of 2360 bytes. Some fonts need more, and the module covers them with a subtype:

```
LargeFontDesc = object (FontDesc)
  ext: array 2560 of byte // enlarges the raster of the base type
end
```

Inheritance is used here for its memory layout alone: a `LargeFontDesc` is a `FontDesc` whose raster runs over into the extension, no method, no dispatch, just the guarantee that the extension lies behind the base fields. `Fonts.GetPat` hands a glyph to `Display.CopyPattern` by address, `patadr^ := @g.w`, because a glyph stores width, height and raster lines in exactly the `Pattern` layout of [Chapter 21](#); this is also why `Fonts` imports `Display`, for the `Pattern` type.

22.4. TextFrames: the view

`TextFrames` renders a text into a frame and edits it with the caret and the selection. Its connection to `Texts` is one procedure variable, the notifier: a text knows nothing about frames, but calls `T.notify` after every change, and the display side installs:

```
procedure NotifyDisplay*(T: Texts.Text; op: Texts.Op; beg, end_: int32)
  var M: UpdateMsg
begin
  M.id := op
  M.text := T
  M.beg := beg
  M.end_ := end_
  Viewers.Broadcast(@M)
end NotifyDisplay
```

The change becomes an `UpdateMsg` broadcast to every frame on the screen, and each frame showing that text repairs itself. Two frames on the same text stay consistent with no observer list and no registration: model to notifier to broadcast to type case. The receiving end is `TextFrames.Handle`, the largest handler of the system and the full picture of the dispatch style of [Chapter 9](#):

```

procedure Handle*(F: Display.Frame; M: ^Display.FrameMsg)
  var F1: Frame
begin
  case F of
  | Frame: // Oberon 90's WITH F: Frame DO
    case M of
    | Oberon.PInputMsg:
      if M.id = Oberon.track then
        Edit(F, M.X, M.Y, M.keys)
      elseif M.id = Oberon.consume then
        ...
    | Oberon.PControlMsg:
      ...
    | Oberon.PSelectionMsg:
      GetSelection(F, @M.text, @M.beg, @M.end_, @M.time)
    | Oberon.PCopyMsg:
      ...
    | MenuViewers.PModifyMsg:
      ...
    | PUpdateMsg:
      if F.text = M.text then
        Update(F, M)
      end
    end
  end
end
end Handle

```

One procedure receives everything the frame can experience: input, defocus, the selection query, copying, resizing, text updates. Messages it does not name fall through the type case and are ignored, which is the extension guarantee of [Chapter 21](#) seen from the receiving side. `TextFrames.Call` closes the circle to [Chapter 18](#): a middle click scans the name under the cursor and hands it to `Oberon.Call`, which finds the registered command; every text on the screen is also a menu.

22.5. MenuViewers and Edit

A `MenuViewers.Viewer` stacks two frames, a menu bar and a main frame, and routes messages between them; its `Handle` translates a viewer-level `modify` into `ModifyMsg` messages for the two children. On top of all this, the editor is small. `Edit.Open` is the whole procedure that opens a named text in a new viewer:

```
procedure Open*()
  var T: Texts.Text
      S: Texts.Scanner
      V: Viewers.Viewer
      X, Y, beg, end_, time: int32
begin
  Texts.OpenScanner(@S, Oberon.Par.text, Oberon.Par.pos)
  Texts.Scan(@S)
  ...
  if S.class = Texts.Name then
    Oberon.AllocateUserViewer(Oberon.Par.vwr.X, @X, @Y)
    V := MenuViewers.New(TextFrames.NewMenu(@S.s, StandardMenu),
      TextFrames.NewText(TextFrames.Text(@S.s), 0), TextFrames.menuH, X, Y)
  end
end Open
```

It scans its parameter, allocates a screen position, and composes a viewer from a menu frame and a text frame. There is no editor object and no application class; the "editor" is the sum of `Texts` (the model), `TextFrames` (the view and the interaction) and a handful of commands like this one, each callable from any text on the screen. The `StandardMenu` string, `"System.Close System.Copy System.Grow Edit.Search Edit.Store"`, is itself just a text whose words happen to be commands.

What the text system demonstrates, beyond its own machinery, is how far the middle of the language carries: pieces and buffers need level 5, the handlers need the objects of level 4, and everything else, notifiers, riders, scanners, menu composition, is procedure variables, records and modules, available from level 1. The next chapter turns to the graphics editor, where the same frame becomes a drawing surface.

Chapter 23. Graphics and Applications

The last layer of the Micron System is what a user would call the applications: the line drawing editor Draw, built on `Graphics` and `GraphicFrames`, and the small demonstration programs, Hilbert, Sierpinski, Stars and the Hennessy benchmark suite. Draw is the second object-oriented framework of the system, and the more radical one: where the text system dispatches on message types, Draw also dispatches on object classes that the framework has never heard of.

23.1. Graphics: objects with a method suite

A drawing is a list of objects:

```
ObjectDesc* = object
  x*, y*, w*, h*: int32
  col*: byte
  selected*, marked*: boolean
  do_*: Method
  next: Object
end
```

Every object carries `do_`, a pointer to its method suite:

```
MethodDesc* = record
  module_*: Name
  allocator*: Name
  new_*: Modules.Command
  copy_*: procedure(from: Object; to_: Object)
  draw*: Handler
  change*: Handler
  selectable*: procedure(obj: Object; x: int32; y: int32): boolean
  read*: procedure(obj: Object; R: pointer to Files.Rider; C: pointer to Context)
  write*: procedure(obj: Object; cno: int32; R: pointer to Files.Rider; C: pointer to Context)
  print_*: procedure(obj: Object; x: int32; y: int32)
end
```

This is a virtual method table written by hand: one record of procedure variables per class, one pointer per object. It is worth pausing on what that buys over the `object` dispatch of [Chapter 9](#). Bound procedures dispatch on operations fixed in the type declaration; a method suite is a value, built at run time, shared by all instances of a class, and extensible by modules the framework does not import. The framework

operates on it blindly:

```
procedure Draw*(G: Graph; M: ^Msg)
  var obj: Object
begin
  obj := G.first
  while obj # nil do
    obj.do_.draw(obj, M)
    obj := obj.next
  end
end Draw
```

Graphics knows lines, captions and macros; everything else comes from outside.

23.2. Rectangles: a new class in one module

Rectangles adds a class to Draw without any module of Draw importing it. The new object type extends ObjectDesc, and the module body builds the method suite once:

```
RectDesc* = object (Graphics.ObjectDesc)
  lw*: int32
  vers*: int32
end
...
begin
  newgc(method)
  copy(@method.module_, "Rectangles", Graphics.NameLen)
  copy(@method allocator, "New", Graphics.NameLen)
  ...
```

Its Draw handler narrows the received message with the type case, and its Copy narrows the object with type guards:

```
dst(Rectangle).lw := src(Rectangle).lw
dst(Rectangle).vers := src(Rectangle).vers
```

The guard dst(Rectangle) asserts and checks that this Graphics.Object is a Rectangle; inside its own class module the assertion holds by construction. Both dispatch directions of Wirth OO meet here: the method suite selects the class procedure, the type case inside the procedure selects the message.

The method suite carries two names, module_ and allocator, and they are what makes a

drawing file portable across programs: `Graphics.Store` writes "Rectangles" and "New" next to the objects of the class, and `Graphics.Load` resolves them at load time:

```
Modules.Load(module_, @mod_)
com^ := Modules.ThisCommand(mod_, allocator)
```

In the original system this loads the module from disk; in the statically linked port, `Modules.Load` of [Chapter 20](#) finds the already linked module, and the file format keeps working unchanged. A class is named by a command, the same M.P convention that runs the system, now used as a serialization format.

23.3. Draw: commands over a shared model

`Draw` itself is, like `Edit` in [Chapter 22](#), thin: a set of commands, `Draw.Open`, `Draw.SetWidth`, `Draw.ChangeColor`, `Draw.Macro`, operating on the focused `GraphicFrames.Frame`. `GraphicFrames` mirrors `TextFrames`: it extends `Display.FrameMsg` with its own update messages, broadcasts changes so that two viewers on one drawing stay consistent, and translates a `Graphics.Msg` into screen coordinates by adding the frame origin before handing it to the objects. Nothing in this layering is new; that is the observation worth making. The mechanisms of chapters [Chapter 21](#) and [Chapter 22](#) carry a second, structurally different application without modification.

23.4. The small programs

The remaining applications fit in a page each and each shows one trait. Hilbert and Sierpinski draw recursive curves directly onto the display; Wirth expresses the four mutually recursive curve parts with procedure variables:

```
var
  A, B, C, D: procedure(i: int32)

procedure HA(i: int32)
begin
  if i > 0 then
    D(i - 1); W(); A(i - 1); S(); A(i - 1); E(); B(i - 1)
  end
end HA
...
A := HA; B := HB; C := HC; D := HD
```

In Micron the forward declaration of [Chapter 5](#) would do, but the port keeps Wirth's formulation, which needs no language support at all: mutual recursion as data. Stars animates through an Oberon background task, Stars.Run installs it and Stars.Stop removes it, the task mechanism of [Chapter 21](#) used from application code. Hennessy [\[TBD\]](#), the classic benchmark collection, times its computations with Oberon.Time and prints through Texts; it is also the module to copy from when measuring anything in the system.

With it the tour of the Micron System is complete: from the boot header to a benchmark suite, every line in Micron, every mechanism one of those introduced in [Part II](#). The next chapter leaves the system and looks at what else has been written in the language.

Chapter 24. Case Studies Beyond the System

The Micron System shows what the language does on a bare machine. This closing chapter looks at the other large programs written in Micron so far, all of them ports of well-documented originals, chosen so that a reader can hold the Micron text against a book that explains it line by line. They are hosted programs, compiled with `--run` or linked against a C library, and they exercise a different profile than an operating system: long-lived interpreters, compilers, and a benchmark suite. All of them ship with the Micron or related repositories.

24.1. S9: a Scheme system

S9fES, Scheme 9 from Empty Space [\[R6\]](#), is Nils M Holm's Scheme interpreter, written in C and explained in his book of the same name. The Micron port keeps the original function and variable names, and the logic inside the procedures nearly line for line, so the book explains the Micron text as well as the C text. Nine modules, about 7500 lines, all at level 3.

Level 3 deserves a remark: a Scheme system lives on garbage collection, yet the port does not use `newgc`. It follows the original design instead, where a cell is an index:

```
cell* = int
```

The heap is a set of parallel arrays, `Car`, `Cdr` and `Tag`, and the collector is S9's own mark and sweep over them, with an explicit root array and an explicit shadow stack (`M.save`, `M.unsave`) protecting intermediate results, exactly the discipline that [Chapter 8](#) described for manual memory and that a C implementation of an interpreter has to follow anyway. A managed language is implemented here in an unmanaged one, which is the same relation as Kernel to the system in [Chapter 19](#), one level down.

S9's reader is also where `goto` earns the place that [Chapter 5](#) reserved for it: the scanner leaves its nested loops with `goto num_done`, `goto real_frac_done` and their kin, as the C original does, and restructuring them into flags would move the Micron text away from the book that documents it.

24.2. P4: the portable Pascal compiler

P4 is the 1976 portable Pascal compiler of Nori, Ammann et al. [R4], in the source form documented by Pemberton and Daniels; it compiles Pascal to P-code and interprets it. The Micron port stays faithful to the original, and the original line numbers are referenced in comments, so the Pemberton and Daniels book works as its commentary.

Its interest for this book is the Pascal heritage coming full circle, and one feature in production use: the variant records of [Chapter 6](#). P4's central data structures are Pascal variant records, and they translate directly:

```
StructRec* = record
...
  case
    | inline scalar_v*: record scalkind*: DeclKind; fconst*: Ctp end
    | inline subrange_v*: record rangetype*: Stp; min*: Valu; max*: Valu end
...
end
```

Each variant is an inline record overlaying the others, which is what Pascal's `case` inside a record meant, without Pascal's undefined behavior when the tag and the access disagree.

24.3. A Smalltalk-80 virtual machine

The third study implements the Smalltalk-80 virtual machine after the "Blue Book", Goldberg and Robson's Smalltalk-80: The Language and its Implementation [R3], whose part four specifies the VM precisely enough to code from. Three modules carry the design: `ObjectMemory`, which loads a Smalltalk-80 image into the object table and reclaims with mark and sweep where the Blue Book prescribes reference counting; `Interpreter`, the bytecode interpreter, the largest module at about 2900 lines; and `BitBlit`, the raster operation of chapter 18 of the Blue Book, a deliberate textbook transcription. Where S9 works with array indices, the Smalltalk VM works with object pointers as 16-bit words, fields extracted with the shift and mask idioms of [Chapter 12](#), and the display is a bitmap driven by `BitBlit` alone. Smalltalk is the third object system in this book, after the two of the Micron System, and the most instructive one to compare: classes, metaclasses and dynamic dispatch exist here purely as data in the object memory, interpreted by a program that itself uses no object orientation at all.

24.4. Are We Fast Yet: one suite, four styles

The awfy benchmark suite, derived from the SOM "Are We Fast Yet" benchmarks, exists in Micron in four parallel implementations of the same programs:

- Procedural, derived from the C version, plain records and procedures;
- Hybrid, with statically bound procedures and delegates ([Chapter 7](#));
- Interfaces, where the benchmark harness sees each program through an interface;
- Objects, the classical class-based form of [Chapter 9](#), at level 4.

The object-oriented variant is the most direct descendant of the originals, which come from the Smalltalk worlds: every benchmark extends an abstract `Benchmark` object and overrides its hooks,

```
Benchmark* = object end
```

```
procedure Benchmark.benchmark*(): integer
```

```
procedure Benchmark.verifyResult*(result: integer): boolean
```

```
procedure Benchmark.innerBenchmarkLoop*(innerIterations: integer): boolean
```

and the harness runs whatever `getSuiteFromName` returns as a pointer to `Benchmark`, dispatching dynamically. Two of its choices are worth reading closely. First, it stays at level 4: instead of `newgc` there is a root object with a reference count and `addRef/release` bound procedures, and `release` calls a `deinit` hook before `dispose`, so every class frees what it owns. Object orientation does not imply a collector, and the ownership discipline of [Chapter 8](#) carries over unchanged. Second, the collection library under `som/` crosses objects with the generic modules of [Chapter 10](#): `Vector(T)` is a generic module whose element container is an object type, so `DeltaBlue` can extend `Vector(ConstraintPtr)` into its `Plan` class, and the iteration interfaces (`ForEach`, `Test`, `Comparator`) are empty objects from a generic `Interfaces(T)` module that a caller extends locally, often as a type declared inside the using procedure.

The suite is the ground for claims about cost: the same algorithms, the same structure, only the abstraction mechanism varies, so a measured difference is attributable to the mechanism. It is also the largest body of idiomatic application Micron outside the system, and a good place to read how the styles of [Chapter 7](#) and [Chapter 9](#) look at scale.

24.5. What the studies say

Four programs, four profiles: an operating system on bare metal, two language implementations, a VM specified in a book, a benchmark suite. None of them needed a language extension, a compiler switch beyond the target, or an escape into C. That is the claim of [Chapter 1](#), tested: the language levels span the space from boot code to garbage-collected applications, and the programs that prove it can be read next to the books that explain them.

Part IV: Appendices

TBD

Appendix A: The Tools

This appendix collects what is needed to reproduce every example of the book. The main text uses the tools without explaining them; this is the one place that does.

A.1. micc

`micc` is the Micron compiler. One invocation parses the main module, follows its imports, compiles everything, and links. There is no separate build system for a Micron program; the import graph is the build description.

The modes, selected by one option each:

(none)	compile and link an executable for the selected target
<code>--run</code>	run the program in the built-in MIL interpreter, no code generated
<code>--check</code>	parse and check only, no code generated
<code>-c</code>	compile all modules to object files (<code>.mob</code>), do not link

The options used throughout this book:

<code>-I <path></code>	add a directory searched for imported modules (<code>.mic</code> , <code>.mil</code> , <code>.mob</code>)
<code>-o <file></code>	name of the executable (link mode)
<code>--target <t></code>	code generator: <code>rv32</code> , <code>rv32-esp32</code> , <code>armv7</code> or <code>x86</code>
<code>--base <addr></code>	load address of the executable, hexadecimal
<code>--runtime <file></code>	a MIL file implementing the <code>MIC\$</code> runtime, for freestanding targets
<code>--abi <default c></code>	calling convention; <code>c</code> for interoperation with C
<code>--dump <mil ml mrl></code>	write the intermediate representation next to the output
<code>-g</code>	generate debug information
<code>-q</code>	report only warnings and errors
<code>-l, -L</code>	link C archives, as usual

The main argument is a module file or a project file (`.micpro`), followed optionally by foreign object files and archives for the linker.

Most examples of this book are hosted programs and run with the interpreter:

```
micc --run Example.mic -I <oakwood>
```

where `<oakwood>` is the directory of the oakwood library (`Out`, `In`, `Files`, `Args`, `Math`...) that ships with the Micron repository TBD

A.2. The RV32 emulator

The Micron System of [Part III](#) runs on `rv32vm`, an emulator combining the `rv32emu` RISC-V interpreter core, Berkeley SoftFloat, and the Project Oberon devices (display, PS/2 keyboard and mouse, SPI disk, timer, LEDs, RS232) rendered through SDL2. It lives in the `vm` directory of the OberonSystem repository and builds with one script:

```
cd vm && ./build.sh          # requires a C compiler and SDL2 (sdl2-config)
```

The result is `vm/build/rv32vm`. Its two relevant options are `--elf`, to load a statically linked ELF file such as the one `micc` produces, and `--ram <MB>`, the RAM size that ends up in the boot header.

A.3. Building and running the Micron System

With `micc` and `rv32vm` in place, the system builds and boots with two commands from the system's source directory:

```
micc --target rv32 -g --base 0 --runtime MIC+.mil -I . -o Main Main.mic
rv32vm --elf --ram 4 --disk disk.img Main
```

`disk.img` is a plain file serving as the SPI disk; it contains the Oberon file system with the fonts and sample texts.

TBD

Appendix B: The Micron System: Build, Boot and Design Notes

[Part III](#) explains the Micron System where it teaches the language. This appendix collects the operational facts in one place: how the system is built and started, and the design decisions of the port in summary form. It condenses the design description that accompanies the sources ("The Micron Project Oberon System, design description", Design.md), which remains the authoritative document and covers each point in more depth.

B.1. Build and startup

Wirth's system is built in two stages: the compiler produces `.rsc` object files, and a boot linker prelinks the inner core into a boot image that the machine loads from the disk's boot sectors; the outer core is loaded dynamically by `Modules`. The Micron System has one stage. `micc` compiles and links all 27 modules into a single ELF file (the command line is in [Appendix A](#)), and the emulator loads it directly.

The module bodies run in import dependency order and replace the initialization the boot linker generates. `Main` is the last body to run and plays the role of Wirth's boot linker for the outer core: it registers the commands of `System`, `Edit`, `Draw` and the example modules with `Modules.Register`, and it enters `Oberon.Loop`, which in Wirth's system is entered at the end of `Oberon's own body.

B.2. The design decisions in summary

Modules

There is no loader and no boot linker. `Modules.Load` finds the descriptor of an already linked module and creates it on demand; `Kernel.modules` is the list of these descriptors.

Memory

The Kernel owns the heap; `new` and `newgc` reach it through the `MIC$` runtime callbacks installed at startup, without compiler magic. The free lists and block headers are Wirth's design, unified to a 16-byte header. See [Chapter 19](#).

Collector

The collector is conservative: it scans the stack, the static data area and the spilled registers, and keeps everything that looks like a pointer into the heap. Registers are made visible with `SYS.spillRegs`. See [Chapter 19](#).

Memory layout

Wirth's fixed map places the display frame at `0E7F00H`. The emulator instead takes `--ram <MB>`, places the framebuffer at the top of RAM and passes the layout in the boot header: heap origin at `08H`, origin of the writable data at `0CH`, and end of usable RAM (`displayBase - 16`) at `010H`. `Kernel` and `Display` read the header instead of using constants. With `--ram 1` the layout is bit-for-bit Wirth's and an original Oberon disk image still boots; the system itself runs with 4 MB. See [Chapter 18](#).

Machine access

There is no `SYSTEM` module. Pointer literals, `ptroff`, `cast`, `sigl/usig` and `val` express in the language what Oberon delegates to `SYSTEM`; the remaining machine-specific pieces (`ROT`, `spillRegs`, `TRACE`) are the small module `SYS`. All absolute addresses of the port are concentrated in `SYS`, `Kernel`, `MIC` and `Display`, so grepping for a pointer literal audits every place where the system touches the machine. See [Chapter 12](#) and [Chapter 19](#).

Arithmetic

Address and size computations moved to the unsigned domain; Oberon's truncating division survives as `obdiv/obmod` where the original algorithms depend on it. See TBD.

Not implemented

The port has no traps and no dynamic module loading; the compiler is not part of the system (cross development from the host). The design description discusses each omission and what restoring it would take.

Appendix C: Wirth's RISC5 and Our RV32

Two processor names appear in this book that are easily confused and must not be: RISC-5 and RISC-V.

RISC-5 is Wirth's own processor, designed for Project Oberon and described in his book about the computer [TBD]. It is a 32-bit machine of deliberate minimality: 16 registers, a small single-cycle instruction set, and a few special registers, among them `H` for the high word of a multiplication and the remainder of a division. Wirth implemented it on an FPGA; the "5" is a version number of his RISC series.

RISC-V is the open instruction set architecture from Berkeley. The Micron System targets its 32-bit variant: the emulator's core is `rv32emu`, configured with the `M` (multiply/divide), `A` (atomics) and `F` (single float) extensions and without the compressed instructions, and `micc --target rv32` generates code for exactly that.

Both Berkeley's and Wirth's share common design goals and features, but are independent achievements.

The port keeps everything around the processor which it replaces:

- The devices are Wirth's, at Wirth's addresses: display frame buffer, millisecond timer, SPI to the disk, PS/2 keyboard and mouse, LEDs, RS232. The software of [Part III](#) cannot tell it is not talking to the FPGA board.
- The memory map is Wirth's when run with `--ram 1`, and generalized through the boot header otherwise ([Chapter 18](#)).
- The code is RISC-V, produced by `micc`, in a statically linked ELF file instead of `.rsc` object files and a boot image.

What the change of processor cost in the sources is instructive because it is so little, and all of it is visible in the language:

SYSTEM.H

On RISC-5 a division leaves the remainder in the `H` register and `SYSTEM.H(0)` reads it. RISC-V has no such register; Micron expresses the remainder as `mod` (or `obmod` where Oberon's truncating semantics are required, [Chapter 3](#)), and Wirth's `ASSERT(SYSTEM.H(0) = 0)` checks became real assertions on `mod` TBD.

Rotation

RISC-5 has a rotate instruction, reached as `SYSTEM.ROT`. The base RV32 instruction set has none, so `SYS.ROT` composes it from two shifts and a modulo on the count ([Chapter 12](#)).

Traps

Wirth's compiler emits trap instructions for run-time checks and the Kernel catches them. The port has no trap machinery; `assert` and `halt` report through the RS232 trace instead. Restoring traps would be an exercise in RISC-V exception handling, not in Micron.

Everything else, including the raster operations, the disk driver and the collector, is processor-independent Micron and moved without change to its logic. That is the practical meaning of the claim in [Chapter 1](#) that the language, not the processor, carries the system.

Appendix D: Reading MIL

`micc` compiles Micron to MIL, the Micron Intermediate Language, before generating native code or interpreting. MIL is inspired by the ECMA-335 [TBD] (.NET) common intermediate language with the baggage removed: a typed stack machine with modules, procedures and the Micron type system, no classes, no metadata streams, no exceptions machinery beyond what the language needs. MIL has a textual form, and being able to read it pays off in three places of this book: checking what `inline` expanded to (Chapter 5), verifying a calling convention at the C boundary (Chapter 13), and understanding what the runtime interface `MIC$` receives from generated code ([TBD], the system's `MIC+.mil` is a handwritten MIL module).

D.1. A module and its MIL

The option `--dump mil` writes `<Module>.mil` next to the output. For the small module

```
module Demo
var x: int32
procedure Twice(n: int32): int32
begin
  return n * 2
end Twice
begin
  x := Twice(21)
end Demo
```

`micc` produces:

```
module Demo
var x*: int32
procedure Twice* (n: int32):int32
begin
  ldarg_0
  ldc_i4_2
  mul
  ret
end Twice
procedure begin$() init
begin
  ldc_i4_s 21
  call Twice
  stvar x
end begin$
```

end Demo

The shape mirrors the source: a module with declarations and procedures, the module body compiled into a procedure marked `init`. Only the statement bodies changed representation: each is a sequence of stack instructions.

D.2. The instruction vocabulary

Reading MIL needs a small vocabulary, best learned from the dump above:

- Loads push values: `ldarg_0` the first parameter, `ldc_i4_2` the constant 2 as a 32-bit integer (`_s` marks a short immediate operand), `ldvar` a module variable, `ldloc` a local, `ldind` the value a pointer points to.
- Stores pop values into places: `stvar x`, `stloc`, `stind`.
- Operators pop their operands and push the result: `mul`, `add`, `div`, the comparisons, the conversions.
- Control is structured: `call` and `ret` as expected, and MIL keeps `if/while/switch` blocks rather than raw jumps, which is why a dump remains readable.

Suffixes carry the type (`i4`, `i8`, `u4`, `r4`...), so the dump also answers signedness questions: a Micron `div` on `uint32` becomes an unsigned division instruction, visibly distinct from the signed one, the guarantee behind the discipline of [Chapter 3](#).

D.3. The two lower forms

Two further dumps show the later stages of the pipeline. `--dump mll` writes the lowered linear form, still a stack machine but with blocks flattened to numbered instructions and every operation carrying its type:

```
proc Demo!Twice
  0: ldarg_i4 0
  1: ldc_i4_2
  2: mul_i4
  3: ret 4
```

`--dump mrl` writes the register form the native code generators work from, with the stack

traffic converted to virtual registers:

```
proc Demo!Twice (vregs: 3, ops: 4)
  0: v0 = ldarg_i4 0
  1: v1 = ldc_i4 2
  2: v2 = mul_i4 v0, v1
  3: ret v2
```

From here to RISC-V, ARM or x86 is register allocation and instruction selection. For the purposes of this book the `.mil` form suffices; the lower forms are useful when a question is about generated code quality rather than semantics.

D.4. Handwritten MIL

MIL files are also compiler input: an import is satisfied by a `.mic`, `.mil` or `.mob` file alike. The system uses this once, for `MIC+.mil` ([TBD]): the runtime interface `MIC$` is not expressible as a Micron module name, so a small handwritten MIL module forwards each `MIC$` procedure to its implementation in the ordinary Micron module `MIC`. Reading it is a good exercise: it contains nothing but the vocabulary above.

Appendix E: Syntax Summary

The complete syntax of Micron in EBNF, reproduced from the language specification [R9], which remains the authoritative source. Alternatives are separated by |, brackets [] mean optionality and braces { } repetition.

```
ident  = ( letter | '_' ) { letter | digit | '_' }
letter = 'A' ... 'Z' | 'a' ... 'z'
digit  = '0' ... '9'
number = integer | real
integer = (digit {digit|'_'} ['0' | 'B'] | digit {hexDigit|'_'} 'H')
         ['U' | 'U1' | 'U2' | 'U4' | 'U8' | 'I' | 'I1' | 'I2' | 'I4' | 'I8']
real    = digit {digit|'_'} '.' {digit|'_'} [Exponent]
Exponent = ('E' | 'D' | 'F' ) ['+' | '-' ] digit {digit}
hexDigit = digit | 'A' ... 'F'
digit    = '0' ... '9'
character = digit {hexDigit} ('X' | 'x')
string = ''' {character} ''' | "" {character} ""
qualident = [ident '.'] ident
identdef  = ident ['*' | '-']
ConstDeclaration = identdef '=' ConstExpression
ConstExpression  = expression
TypeDeclaration = identdef '=' type
type             = NamedType | ArrayType | RecordType | PointerType | ProcedureType
                  | enumeration | ObjectType | InterfaceType
NamedType        = qualident
ArrayType        = ARRAY [ length ] OF type | '[' [ length ] ']' type
length           = ConstExpression
varlength        = expression
RecordType       = RECORD [FixedPart][VariantPart] END
FixedPart        = FieldList { [';'] FieldList }
FieldList        = IdentList ':' type [ BITS integer ]
                  | '..' BITS integer
                  | INLINE identdef ':' type
VariantPart      = CASE ['|'] Variant ( { '|' Variant } | { Variant } )
Variant          = [INLINE] identdef ':' type [';']
IdentList        = identdef { [';'] identdef }
ObjectType       = OBJECT ['(' BaseType ')'] { IdentList ':' type [ ';' ] } END
BaseType         = NamedType
PointerType      = ( POINTER TO | '^' ) type
ProcedureType    = PROCEDURE [FormalParameters]
ProcedureType    = ( PROCEDURE | PROC ) '(' ( POINTER | '^' ) ')' [FormalParameters]
InterfaceType    = INTERFACE {InterfaceProc} END
InterfaceProc    = [PROCEDURE | PROC] identdef [FormalParameters] [';']
enumeration ::= '(' constEnum ')'
constEnum ::= ident [ '=' ConstExpression ] { [';'] ident }
VariableDeclaration = IdentList ':' type
designator = (qualident|constructor) {selector}
selector  = '.' ident | '[' expression ']' | '^' | '(' qualident ')'
```

```

ExpList    = expression {[','] expression}
expression = SimpleExpression [ relation SimpleExpression ]
relation   = '=' | '#' | '<' | '\<=' | '>' | '>=' | IN
SimpleExpression = ['+' | '-' ] term { AddOperator term }
AddOperator  = '+' | '-' | OR
term         = factor {MulOperator factor}
MulOperator  = '*' | '/' | DIV | MOD | '&' | AND
literal      = number | string | hexstring | hexchar
              | NIL | TRUE | FALSE | set
factor       = literal | designator [ActualParameters]
              | '(' expression ')' | ('~'|NOT) factor
              | '@' designator
ActualParameters = '(' [ ExpList ] ')'
set             = '{' [ element {[','] element} ] '}'
element         = expression ['..' expression]
FunctionCall    = designator ActualParameters
constructor = [NamedType] '{' [ component {[','] component} ] '}'
component = ident ':' expression
           | '[' ConstExpression ']' ':' expression
           | expression ['..' expression]
hexstring = '$' {hexDigit} '$'
statement = [ assignment | ProcedureCall | IfStatement
            | CaseStatement | LoopStatement
            | ExitStatement | GotoStatement | gotoLabel
            | ReturnStatement | RepeatStatement | ForStatement ]
StatementSequence = statement { [";"] statement}
assignment = designator ':=' expression
ProcedureCall = designator [ ActualParameters ]
IfStatement   = IF expression THEN StatementSequence
               {ElifStatement} [ElseStatement] END
ElifStatement = ELSIF expression THEN StatementSequence
ElseStatement = ELSE StatementSequence
CaseStatement = CASE expression OF ['|'] Case { '|' Case }
               [ ELSE StatementSequence ] END
Case          = [ CaseLabelList ':' StatementSequence ]
CaseLabelList = LabelRange { [','] LabelRange }
LabelRange    = label [ '..' label ]
label         = ConstExpression
WhileStatement = WHILE expression DO StatementSequence END
RepeatStatement = REPEAT StatementSequence UNTIL expression
ForStatement    = FOR ident ':=' expression TO expression
                 [BY ConstExpression]
                 DO StatementSequence END
LoopStatement   = LOOP StatementSequence END
ExitStatement   = EXIT
ReturnStatement = RETURN [ expression ]
ExitStatement   = EXIT
GotoStatement   = 'GOTO' ident
gotoLabel       = ident ':'
ProcedureDeclaration = ProcedureHeading ( [ INLINE | INVAR ] [';'] ProcedureBody
                                         | [';'] EXTERN ['C' [string] | string] )
                                         | ForwardDeclaration

```

```

ProcedureBody      = DeclarationSequence BEGIN StatementSequence
                    [FINALLY StatementSequence] END [ident '.'] ident
DeclarationSequence = { CONST { ConstDeclaration [';'] }
                      | TYPE { TypeDeclaration [';'] }
                      | VAR { VariableDeclaration [';'] }
                      | ProcedureDeclaration [';'] }
FormalParameters = '(' [ FPSection { [';'] FPSection } ] ')'
                  [ ':' ReturnTyp ]
ReturnTyp         = [ POINTER TO | '^' ] NamedType
FPSection         = [CONST] ident { [';'] ident } ':' FormalType
FormalType        = type
ProcedureHeading = ( PROCEDURE | PROC ) [Receiver] identdef [ FormalParameters ]
ForwardDeclaration = ( PROCEDURE | PROC ) '^' [Receiver] identdef [ FormalParameters ]
Receiver = '(' ReceiverParam ':' ReceiverType ')' | ReceiverType '.'
ReceiverParam = ident
ReceiverType = ident
module         = MODULE ident [ MetaParams ] [ ModuleAttributes ] [';']
                { ImportList | WhereDecls | DeclarationSequence }
                [ BEGIN StatementSequence [FINALLY StatementSequence] ]
                END ident [';']
ModuleAttributes ::= '[' Attribute { [';'] Attribute } ']'
Attribute ::= ident [ '=' ConstExpression ]
ImportList = IMPORT import { [';'] import } [';']
import     = [ ident ':' ] ImportPath ident [ MetaActuals ]
ImportPath = { ident '.' }
MetaParams = '(' MetaSection { [';'] MetaSection } ')'
MetaSection = ['CONST' | 'TYPE'] ident { [';'] ident } [ ':' NamedType ]
WhereDecls  = WHERE { WhereDeclaration [';'] }
WhereDeclaration = ident ':' type
MetaActuals = '(' ConstExpression { [';'] ConstExpression } ')'
directive = '<*' ident ( '+' | '-' ) '*>'
directive = '<*' [ scIf | scElsif | scElse | scEnd ] '*>'
scIf      = IF scExpr THEN
scElsif   = ELSIF condition THEN
scElse    = ELSE
scEnd     = END
condition = scTerm { OR scTerm }
scTerm    = scFactor { '&' scFactor }
scFactor  = ident | '(' condition ')' | '~' scFactor

```


Appendix F: Predeclared Types and Procedures

A compact reference, condensed from the language specification [R9]; consult the specification for the precise argument and result types and the halting conditions.

F.1. Predeclared types

Name	Meaning
boolean	truth values, see Chapter 3
char	Latin-1 character
int8 int16 int32 int64	signed integers; <code>shortint</code> , <code>integer</code> , <code>longint</code> and <code>i1..i8</code> are synonyms
uint8 uint16 uint32 uint64	unsigned integers; <code>byte</code> and <code>u1..u8</code> are synonyms
flt32 flt64	floating-point; <code>real</code> , <code>longreal</code> , <code>r4</code> , <code>r8</code> are synonyms
set	set of the integers 0..31
any	opaque type, only usable as a pointer base type; a <code>pointer to any</code> is assignment compatible with all pointer types, like <code>void*</code> in C

F.2. Predeclared function procedures

Name	Function
<code>abs(x)</code>	absolute value of a numeric <code>x</code>
<code>asr(x,n)</code>	arithmetic shift right of a signed <code>x</code>
<code>shl(x,n)</code> <code>shr(x,n)</code>	logical shifts
<code>band(x,y)</code> <code>bor(x,y)</code> <code>bxor(x,y)</code> <code>bnot(x)</code>	bitwise operations on integers
<code>bset(x)</code>	the set with the bits of the unsigned <code>x</code>
<code>cap(x)</code>	the upper-case letter of <code>x</code>
<code>cast(T,x)</code>	bit reinterpretation between numeric or pointer types; with <code>@</code> , a pointer created in place
<code>chr(x)</code>	the character with ordinal <code>x</code>

Name	Function
default(T)	the default value of T (zero, false, nil, 0X)
floor(x)	largest integer $\leq$ the real x, as signed integer
flt(x)	the signed integer x as floating-point value
getenv(T,n) getenv(T,n,d)	the value of environment element n, optionally with default d
len(v)	the length of a fixed-size array
min(T) max(T)	smallest/largest value of a basic, set or enumeration type
min(x,y) max(x,y)	the smaller/larger of two values
obdiv(x,y) obmod(x,y)	Oberon's truncating division and its remainder (Chapter 3)
odd(x)	whether the integer x is odd
ord(x)	the ordinal of a char, boolean or enumeration value
ptroff(p,n)	pointer p displaced by n elements
sigc(x)	bitwise reinterpret an unsigned x as a signed integer of the same width (Chapter 3)
sigl(x)	convert an unsigned x to the next larger signed type, preserving the value (Chapter 3)
usig(x)	bitwise reinterpret a signed x as an unsigned integer of the same width (Chapter 3)
size(T)	size of a simple type T in bytes, as uint32
strlen(s)	length of the string s without the terminating 0X
val(T,x)	value conversion between numeric types, or ordinal to enumeration

F.3. Predeclared proper procedures

Name	Function
assert(x) assert(x,n)	terminate program execution if not x, optionally with code n
halt(n)	terminate program execution with code n
inc(v) inc(v,n) dec(v) dec(v,n)	increment/decrement in place

Name	Function
<code>incl(v,x) excl(v,x)</code>	include/exclude element x in the set v
<code>copy(d,s,n)</code>	copy up to n-1 characters from s^ to d^, always 0X terminated
<code>new(p) new(p,n)</code>	allocate, arrays with n elements (Chapter 8)
<code>newinit(p, ...)</code>	allocate and initialize (Chapter 8)
<code>newgc(p) newgc(p,n)</code>	allocate garbage-collected (Chapter 8), level 5
<code>dispose(p)</code>	deallocate (Chapter 8)
<code>pcall(e,p,...) raise(e)</code>	protected call and exception raise, levels 3+
<code>print(a) println(a)</code>	write a value to the console, mainly for debugging
<code>setenv(n,v)</code>	set environment element n to v
<code>getreg(r,v)</code> <code>putreg(r,v)</code>	read/write a processor register, levels 0 and 1
<code>cli() sti()</code>	disable/enable maskable interrupts, levels 0 and 1
<code>nop()</code>	emit a NOP, for timing loops and pipeline flushes, levels 0 and 1

Which predeclared procedures are available at which language level is summarized in [Appendix G](#).

Appendix G: Language Levels Quick Reference

The level concept is introduced in [Chapter 2](#) and used throughout; this is the summary table.

Level	Name	Focus	Key features added
0	Bootstrap	Hardware init	Inlined code, <code>getreg/putreg/cli/sti</code> . No stack.
1	Systems	Drivers, kernels	Stack frames, recursion, static bound procedures. <code>extern</code> allowed.
2	Abstract	Logic, safety	Interfaces (<code>dispatch</code>), <code>finally</code> .
3	Dynamic	Application data	Heap (<code>new/dispose</code>), pointers to heap, optional exceptions (<code>pcall/raise</code>).
4	Object	Hierarchies	Type extension (inheritance), dynamic dispatch, <code>is</code> , type guards, type case.
5	Managed	Automatic memory	Garbage collection (<code>newgc</code>).

The rules:

- A module declares its level in the module attributes, for example `module Kernel [level=1]`; without the attribute, level 3 is assumed.
- A module may import only modules of the same or a lower level.
- Since a module body runs before the bodies of its importers, lower-level modules are initialized before higher-level ones.

The level is a claim checked by the compiler: a level 1 module cannot use the heap even by accident, and a level 4 module cannot silently start depending on the collector. In the Micron System the attribute does real work (`SYS`, `MIC`, `Kernel`, `FileDir` at level 1; `Display`, `Input` at 4; the rest at 5). TBD

Appendix H: Micron for C and Pascal Programmers

Two lookup tables for readers arriving from C or from Pascal/Oberon. They map constructs, not semantics; where a mapping is approximate, the chapter reference has the caveats. The prose companion to these tables is [Chapter 4](#).

H.1. From C

C	Micron	See
<code>int32_t</code> , <code>uint8_t</code> ...	<code>int32</code> , <code>uint8</code> ...	Chapter 3
unsigned arithmetic wrap	same, in the unsigned family	Chapter 3
implicit int conversions	none; <code>val</code> , <code>sigl</code> , <code>usig</code> , <code>sigc</code> , <code>cast</code>	Chapter 3
<code>struct</code>	record	Chapter 6
<code>union</code>	case variants in a record	Chapter 6
bit fields	bits fields in a record	Chapter 6
<code>enum</code>	enumeration type	Chapter 3
<code>T*</code>	pointer to <code>T</code> ($\wedge T$)	Chapter 6
<code>void*</code>	pointer to any	Chapter 6
<code>p→f</code> , <code>(*p).f</code>	<code>p.f</code> (auto dereference), <code>p$\wedge$.f</code>	Chapter 6
<code>p[i]</code> on a pointer	<code>p$\wedge$[i]</code> on a pointer to array, or <code>ptroff</code>	Chapter 6 , Chapter 12
pointer arithmetic	<code>ptroff(p, n)</code> , typed	Chapter 12
<code>(T)x</code> cast	<code>cast(T, x)</code> , bit reinterpretation	Chapter 12
function pointer	procedure type	Chapter 6
<code>malloc/free</code>	<code>new/dispose</code>	Chapter 8
<code>static</code> (file scope)	non-exported module declaration	Chapter 5
<code>extern</code>	<code>extern</code> procedures/modules	Chapter 13
<code>#include</code>	<code>import</code> (semantic, no text inclusion)	Chapter 5

C	Micron	See
<code>#define</code> constant	<code>const</code> declaration	Chapter 5
<code>#if/#ifdef</code>	source code directives <code><* if *></code>	Chapter 11
<code>inline</code>	<code>inline</code> procedures	Chapter 5
<code>constexpr</code> (C++)	<code>invar</code> procedures	Chapter 11
volatile device access	pointer constructor to a device address	Chapter 12
<code>goto</code>	<code>goto</code> , within a procedure	Chapter 5
<code>x / y</code> , <code>x % y</code> (truncating)	<code>div</code> , <code>mod</code> , same truncating semantics	Chapter 3

H.2. From Pascal and Oberon

Pascal (Turbo/Free)	Oberon-07	Micron	See
Integer, Byte, Cardinal	INTEGER, BYTE	<code>int32</code> , <code>uint8</code> , <code>uint32</code> , both families explicit	Chapter 3
unit / interface / implementation	MODULE, export mark <code>*</code>	module, export marks <code>*</code> and <code>-</code>	Chapter 5
uses	IMPORT	<code>import</code> t, with aliases	Chapter 5
<code>record ... case ... end</code>	records, no variants	records with <code>case</code> variants	Chapter 6
packed record	-	bits fields	Chapter 6
array [1..10] of T	ARRAY 10 OF T (0-based)	<code>array 10 of T</code> (0-based)	Chapter 6
dynamic array (FPC)	pointer to open array	pointer to array of T, no length metadata	Chapter 8
New/Dispose	NEW (GC)	<code>new/dispose</code> ; <code>newgc</code> for GC	Chapter 8
nested procedures	nested procedures	nested procedures	Chapter 5
var parameter	VAR parameter	pointer parameter; <code>@v</code> at the call	Chapter 5
procedural type	procedure type	procedure type	Chapter 6

Pascal (Turbo/Free)	Oberon-07	Micron	See
method (object/class, FPC)	type-bound procedure (Oberon-2)	bound procedures and object methods, both receiver forms	Chapter 7 , Chapter 9
class inheritance (FPC)	type extension	object extension, is, guards, type case	Chapter 9
interfaces (FPC, COM-style)	-	interface types, no reference counting	Chapter 7
generics (FPC)	-	generic modules with meta parameters	Chapter 10
try/finally (FPC)	-	finally section per procedure	Chapter 17
exceptions (FPC)	-	pcall/raise, optional	Chapter 17
div/mod (truncating)	DIV/MOD (floor-style)	div/mod truncating as in Pascal; obdiv/obmod for the Oberon semantics	Chapter 3
absolute (FPC)	SYSTEM.PUT/GET	pointer constructor T{addr}	Chapter 12
asm blocks (FPC)	SYSTEM, code procedures	level 0: inline, getreg /putreg	Chapter 12

The largest non-syntactic differences for both audiences are the strict signed/unsigned separation ([Chapter 3](#)), dynamic arrays without length metadata ([Chapter 8](#)), and the language levels ([Appendix G](#)).

Bibliography

- [R1] R. Keller: **Oberon+ Language Report**. oberon-lang.github.io.
- [R2] N. Wirth, J. Gutknecht: **Project Oberon. The Design of an Operating System, a Compiler, and a Computer**. Edition 2013. projectoberon.net.
- [R3] A. Goldberg, D. Robson: **Smalltalk-80: The Language and its Implementation**. Addison-Wesley, 1983.
- [R4] S. Pemberton, M. Daniels: **Pascal Implementation: The P4 Compiler and Interpreter**. Ellis Horwood, 1982. homepages.cwi.nl/~steven/pascal.
- [R5] R. Keller: **Are We Fast Yet? Micron Implementations**. github.com/rochus-keller/Are-we-fast-yet/tree/main/Micron.
- [R6] N. M. Holm: **Scheme 9 from Empty Space**. Lulu Press, 2007ff. t3x.org.
- [R7] H. Mössenböck, N. Wirth: **The Programming Language Oberon-2**. ETH Zürich, 1992/1996.
- [R8] N. Wirth: **The Programming Language Oberon**. Oberon-07 revision, 2013/2016. projectoberon.net.
- [R9] R. Keller: **The Micron Programming Language Specification**. Distributed with the Micron source repository.
- [R10] **The Oakwood Guidelines for Oberon-2 Compiler Developers**. 1995.