UNITED NORTHERN COMPUTERS

UNC 64

Programmer's Reference Guide

Covers the UNC 16 and the UNC 64
UNC-1 processor · System ROM Rev A
UNC BASIC · the assembler · the memory map

UNC64.COM

UNC 64 Programmer's Reference Guide. Rev A.

United Northern Computers is fictional. The UNC 64 is a fantasy 8-bit computer, emulated for real, and the computer in *UNC 64*, a game in development for Mac, PC and Linux by Viljami Kuosmanen.

This book is the machine's specification: every program listing in it is assembled and run by the emulator's tests. It is free to download, print and share.

unc64.com

Contents

About this book 4

1 Getting started 5

2 Bits, bytes and hex 8

3 The UNC-1 processor 9

4 Addressing modes 12

5 The assembler 14

6 Your first programs 21

7 The screen 26

8 Timing and the raster 40

9 Interrupts 43

10 The keyboard 46

11 The beeper 50

12 The system ROM 56

13 Tape 62

14 UNC BASIC 66

15 The video board 72

A Screen codes and colours 73

B Memory map and I/O registers 75

C Instruction set 78

D ROM reference 83

E Assembler messages 85

F For 6502 programmers 88

G The Machine Room 89

H For hosts: watching the machine, and save states 91

About this book

This is the whole machine. Every register, every address and every instruction is in here, with the exact number of cycles each one takes. Nothing about the UNC 64 is hidden behind a driver or a library. If the computer can do it, this book tells you how.

You don't need to have written machine language before. You do need patience, a pencil, and the willingness to try things and see what happens. You can't damage the computer by typing at it. The worst a program can do is stop, and the reset switch fixes that.

Read chapters 1 to 6 in order. After that, dip into whatever you need. The appendices are for keeping open beside the keyboard.

Conventions

You see	It means
`$C020`	A number in hexadecimal (base 16).
`%1010`	A number in binary (base 2).
`42`	A number in decimal.
`A`, `X`, `Y`	The processor's registers (chapter 3).
`fp`	An address in the fast page, `$00` to `$FF` (chapter 4).
`abs`	Any address, `$0000` to `$FFFF`.
`LDA #$41`	Something you type, exactly as shown.

Program listings are complete. A BASIC listing you type a line at a time, pressing RETURN after each. A machine-language listing you type into the assembler, assemble and run.

1. Getting started

Switching on

start here.
—U

Switch on and the screen clears and prints the start-up message:

```
   UNC 64 SYSTEM ROM  REV A

 16K RAM SYSTEM
 UNC BASIC REV A  14335 BYTES FREE
 TYPE HELP FOR HELP

READY.
-
```

The blinking block is the **cursor**. `READY.` means the computer is listening. It speaks **UNC BASIC**, a language built into the machine: whatever you type, it reads when you press RETURN.

Your first words

Type this, exactly, and press RETURN:

```
PRINT "HELLO"
```

The computer answers:

```
HELLO
READY.
```

`PRINT` puts on the screen whatever follows it. The quotation marks mean "these exact letters". Leave them out and BASIC works the answer out instead: `PRINT 2+2` prints `4`, and `PRINT 6*7` prints `42`.

If you make a mistake, BASIC says so with `?SYNTAX ERROR`. Nothing is broken. Type the line again. While you're typing, DELETE rubs out the last letter.

Your first program

A line that starts with a number isn't done straight away. BASIC keeps it, as a line of your **program**. Type these two lines:

```
10 PRINT "HELLO"
20 GOTO 10
```

Nothing happens yet. Type `LIST`, and BASIC shows you the program it's keeping. Type `RUN`, and it runs it: line 10 prints `HELLO`, line 20 sends it back to line 10, and the screen fills with HELLOs, for ever.

To stop it, press **STOP**. BASIC tells you where it was (`BREAK IN 10` or `BREAK IN 20`) and is ready again.

To change a line, type it again with the same number. To get rid of a line, type just its number. `NEW` forgets the whole program. Number your lines in tens, so there's room to slip new ones in between.

Counting

```
NEW
10 FOR I=1 TO 10
20 PRINT I
30 NEXT I
RUN
```

The lines between `FOR` and `NEXT` run again and again, with `I` counting from 1 to 10. Change line 20 to `20 PRINT I, I*I` and run it again: a table of squares. `I` is a **variable**, a named box holding a number. There are 26 of them, `A` to `Z`.

The border

Now type this:

```
POKE 49184,2
```

The border turns red.

You've just put the number 2 into one particular place in the computer's memory, the one numbered 49184. That place isn't really memory. It's wired to the picture: whatever number it holds is the colour of the border. Try the numbers 0 to 15. Then try this:

```
NEW
10 FOR C=0 TO 15
20 POKE 49184,C
30 NEXT C
40 GOTO 10
RUN
```

Everything the UNC 64 can do, it does through places like that one. Chapter 7 says what the screen's places do, Appendix B lists every one of them, and chapter 14 is the whole of UNC BASIC. The rest of this book is about getting at those places faster than BASIC can.

The UNC 16 and the UNC 64

The UNC 16 and the UNC 64 are the same computer. The UNC 16 has 16K of memory (16,384 bytes), the text screen, the bitmap, the beeper and the tape deck: everything in chapters 1 to 14. The UNC 64 is the UNC 16 fully expanded: the 64K memory expansion, with 48K of memory below the I/O and ROM areas; the video board, with sprites, smooth scrolling and background

layers (chapter 15); and the sound chip, with three voices and a speech synthesizer. They arrive one at a time. The start-up message tells you which you have: the ROM checks for the expansion every time you switch on.

	UNC 16	UNC 64
Memory	16K, `$0000-$3FFF`	48K, `$0000-$BFFF`
Free for your programs	`$0800-$3FFF`, 14K	`$0800-$BFFF`, 46K
Everything else	identical	identical

The monitor

The UNC 16 ships with a green monochrome monitor. Plug in a colour monitor and the border and background colours (chapter 7) show in colour. A program can't tell which monitor is connected, and doesn't need to. On the green screen, each colour appears as a brighter or darker shade of green.

Typing

The keyboard sends capital letters, digits, punctuation, RETURN, DELETE and STOP. Letters always arrive as capitals.

2. Bits, bytes and hex

Skip this chapter if you already think in hexadecimal.

Bits and bytes

The UNC 64 stores everything as **bits**, each one either 0 or 1. Eight bits make a **byte**. A byte holds a number from 0 to 255:

```
bit:     7   6   5   4   3   2   1   0
worth: 128  64  32  16   8   4   2   1
```

`%01000001` has bits 6 and 0 set: 64 + 1 = 65. Bit 7 is the "high" bit and bit 0 the "low" bit.

Hexadecimal

Binary is long-winded, so programmers write bytes in **hexadecimal**: base 16, with the digits `0-9` then `A-F` for 10 to 15. Each hex digit is exactly four bits, so a byte is always two hex digits:

```
%0100 0001  =  $41  =  65
%1111 1111  =  $FF  = 255
```

Learn the sixteen four-bit patterns and you can convert between binary and hex in your head. It's worth it.

Addresses and words

The UNC 64 has 65,536 memory addresses, `$0000` to `$FFFF`. An address takes two bytes, called a **word**. The UNC-1 stores words **low byte first**: the address `$C020` sits in memory as `$20` followed by `$C0`. The low byte is also called the "low half" and the high byte the "page": `$C020` is byte `$20` of page `$C0`. A **page** is 256 bytes.

Negative numbers

A byte can also stand for a number from -128 to 127. Bit 7 is then the sign: `$00-$7F` are 0 to 127, and `$80-$FF` are -128 to -1. To negate a number, flip every bit and add one. So -1 is `$FF` and -128 is `$80`. This is called **two's complement**. The clever part is that the processor doesn't care: adding and subtracting work the same either way. Only you decide what the bits mean.

3. The UNC-1 processor

The UNC-1 is an 8-bit processor running at 1 MHz: one million cycles a second. It reads an instruction from memory, carries it out, and moves on to the next. Every instruction takes a fixed number of cycles, listed in Appendix C. Nothing about the timing is left to chance.

Registers

Register	Size	What it's for
`A`	8 bits	The accumulator. Arithmetic and logic happen here.
`X`	8 bits	Index register. Counting, and offsets into tables.
`Y`	8 bits	Index register. Like X; the `(fp),Y` mode uses it.
`SP`	8 bits	Stack pointer. The next free byte of the stack, in page `$01`.
`PC`	16 bits	Program counter. The address of the next instruction.
`P`	8 bits	Processor status: the flags.

The status flags

```
bit:   7   6   5   4   3   2   1   0
       N   V   1   B   -   I   Z   C
```

Flag	Name	Set when
`C`	Carry	An addition carried out of bit 7, a subtraction did *not* borrow, or a shift pushed a 1 out. Also the "greater or equal" result of a compare.
`Z`	Zero	The result was zero. Also the "equal" result of a compare.
`I`	Interrupt disable	Interrupts are held off (chapter 9). Set at switch-on.
`B`	Break	Only ever seen in a copy of P on the stack: set if `BRK` or `PHP` pushed it, clear if an interrupt did.
`V`	Overflow	A signed addition or subtraction gave a result outside -128 to 127. `BIT` also copies bit 6 into it.
`N`	Negative	Bit 7 of the result was set.

Bit 5 always reads 1. Bit 3 does nothing: the UNC-1 has no decimal mode.

Most instructions that produce a value set N and Z from it. Appendix C lists exactly which flags each instruction changes.

The stack

Page `$01` (`$0100-$01FF`) is the stack. Pushing stores a byte at `$0100 + SP`, then decrements SP. Pulling increments SP, then reads. The stack starts at the top (SP = `$FF`) and grows downward. It holds 256 bytes, and it wraps around without warning if you push too many.

`JSR` pushes the address of the instruction after itself, high byte first. `RTS` pulls it back. Interrupts push PC and then P. `PHA`/`PLA` push and pull A; `PHP`/`PLP` push and pull P.

The instructions by family

Load and store. `LDA`, `LDX`, `LDY` copy a byte from memory into a register and set N and Z. `STA`, `STX`, `STY` copy a register into memory and change no flags.

Transfer. `TAX`, `TAY`, `TXA`, `TYA` copy between registers. `TSX` and `TXS` read and set the stack pointer. All of them except `TXS` set N and Z.

Arithmetic. `ADC` adds memory *and the carry* to A. Clear the carry first with `CLC`. `SBC` subtracts memory and the *inverse* of the carry from A. Set the carry first with `SEC`. Carrying in this way is how you add numbers longer than a byte: the carry out of the low byte goes into the high byte (see "Sixteen-bit arithmetic" in chapter 6).

Logic. `AND`, `ORA` and `EOR` combine A with memory bit by bit. `AND #%00001111` keeps the low four bits. `ORA #%10000000` sets bit 7. `EOR #$FF` flips every bit.

Compare. `CMP`, `CPX` and `CPY` subtract memory from the register without storing the result, and set the flags:

After `CMP M`	Flags
A < M (unsigned)	C clear
A = M	Z set, C set
A > M (unsigned)	C set, Z clear

So `BCC` after a compare means "branch if less than", and `BCS` means "branch if greater than or equal".

Increment and decrement. `INX`, `INY`, `DEX`, `DEY` and `INC`, `DEC` (in memory) add or subtract one, wrapping from `$FF` to `$00` and back. They set N and Z and leave the carry alone.

Shifts and rotates. `ASL` shifts left (bit 0 becomes 0, bit 7 goes to C): a quick multiply by two. `LSR` shifts right (bit 7 becomes 0, bit 0 goes to C): a quick divide by two. `ROL` and `ROR` do the same but shift the old carry in, so a chain of them shifts a number many bytes long.

Bit test. `BIT` ANDs A with memory and sets Z from the result without changing A. It also copies bit 7 of the memory byte into N and bit 6 into V, which makes it the quickest way to test the top two bits of an I/O register.

Branches. `BEQ`, `BNE`, `BCC`, `BCS`, `BMI`, `BPL`, `BVC` and `BVS` jump if one flag is set or clear. The destination is up to 128 bytes back or 127 bytes forward from the next instruction. A branch costs 2 cycles, or 3 if it's taken.

Jumps and subroutines. `JMP` goes anywhere. `JMP ($nnnn)` reads the destination from memory first. `JSR` calls a subroutine; `RTS` returns from it.

Flags. `CLC`, `SEC`, `CLI`, `SEI` and `CLV` clear and set flags directly.

Interrupts and stopping. `BRK` is a deliberate interrupt (chapter 9). `RTI` returns from an interrupt. `NOP` does nothing for 2 cycles. `HLT` stops the processor until reset.

Unassigned opcodes

Of the 256 possible opcodes, 136 are instructions. If the processor meets any of the other 120, it **jams**: it stops at the faulty byte, and only a reset will restart it. This usually means your program ran into data, or into empty memory. Empty memory on a newly switched-on machine is all zeros, and `$00` is `BRK`, so running off the end of a program usually lands in the interrupt handler instead (chapter 9). The ROM's handler returns to the next byte, which is another `BRK`, and the processor crawls through empty memory one `BRK` at a time until it reaches something else.

Reset

At switch-on, or when you press reset, every register is cleared to 0, SP is set to `$FF`, and P to `$24` (interrupts disabled). Memory and every I/O register are cleared to zero. The processor then loads PC from the reset vector at `$FFFC` and starts the system ROM (chapter 12).

4. Addressing modes

An instruction is one byte, the **opcode**, followed by up to two bytes of **operand**. The addressing mode says what the operand means. Not every instruction has every mode: Appendix C lists exactly which ones exist.

Mode	Written	Bytes	Meaning
Implied	`INX`	1	No operand.
Accumulator	`ASL A`	1	Work on A instead of memory.
Immediate	`LDA #$41`	2	The operand *is* the value.
Fast page	`LDA $10`	2	The value at address `$00nn`.
Fast page,X	`LDA $10,X`	2	The value at `$00nn + X`. Stays inside the fast page.
Fast page,Y	`LDX $10,Y`	2	As above, with Y. Only `LDX` and `STX`.
Absolute	`LDA $C020`	3	The value at a full 16-bit address.
Absolute,X	`LDA $0400,X`	3	The value at `$nnnn + X`.
Absolute,Y	`LDA $0400,Y`	3	The value at `$nnnn + Y`.
Indirect	`JMP ($0314)`	3	Only `JMP`: jump to the address stored at `$nnnn`.
(Fast page),Y	`LDA ($FB),Y`	2	Read a pointer from `$nn` and `$nn+1`, add Y, use that address.
Relative	`BNE loop`	2	Branches only: a signed offset from the next instruction.

The fast page

The first 256 bytes of memory, `$0000-$00FF`, are the **fast page**. An instruction that uses a fast-page address needs one byte for the address instead of two, so it's a byte shorter, and it runs a cycle faster:

```
LDA $10      2 bytes, 3 cycles
LDA $0010    3 bytes, 4 cycles   (the same byte, the slow way)
```

Keep your most-used variables there. The screen routines use `$F7-$FE` and UNC BASIC uses `$D0-$F6`; `$00-$CF` and `$FF` are yours.

Indexing in the fast page wraps around inside it: `LDA $F0,X` with X = `$20` reads `$0010`, not `$0110`. Absolute indexing wraps at the end of memory: `$FFF0,X` with X = `$20` reads `$0010`.

Pointers: (fast page),Y

To reach a table whose address you only know while the program runs, store the address in two fast-page bytes, low byte first, and index it with Y:

```
        LDA #$00
        STA $FB         ; pointer low byte
        LDA #$06
        STA $FC         ; pointer high byte: the pointer is $0600
        LDY #5
        LDA ($FB),Y     ; reads $0605
```

The pointer's two bytes must both be in the fast page. A pointer at `$FF` takes its high byte from `$00`.

What things cost

Every instruction's cost is fixed. These rules give every number in Appendix C:

Mode	Read or store	Change memory (`INC`, `ASL`…)
Immediate, accumulator, implied	2	2
Fast page	3	5
Fast page,X / Fast page,Y	4	6
Absolute	4	6
Absolute,X / Absolute,Y	5	(none)
(Fast page),Y	6	(none)

And: branches 2, plus 1 if taken. `JMP` 3, `JMP (ind)` 5. `JSR`, `RTS`, `RTI` and `BRK` 6. `PHA` and `PHP` 3, `PLA` and `PLP` 4. Taking an interrupt costs 6.

Crossing a page boundary never costs extra. A loop takes the same time wherever it sits in memory.

5. The assembler

You could write machine language as a list of hex bytes. Nobody should have to. The assembler reads instructions written as words and turns them into bytes, working out the addresses for you.

A line of assembly

```
label:  LDA #$41        ; a comment
```

- A **label** is a name followed by a colon. It stands for the address of whatever comes next.
- An **instruction** is a mnemonic and its operand. Mnemonics can be in any case.
- A **comment** starts with `;` and runs to the end of the line.

Every part is optional. A line can be just a label, just a comment, or empty.

A **constant** gives a name to a value:

```
BORDER  = $C020
```

A constant has a line to itself: `x: FOO = 1` is an error, because a label and a constant on one line would be two names for two different things.

Names start with a letter or `_` and continue with letters, digits and `_`. Capitals matter: `Loop` and `loop` are different names. `A` (in either case) can't be a name, because as an operand it means the accumulator: `ASL A`.

Local labels start with `@`. A local label belongs to the nearest ordinary label above it, so every routine can have its own `@loop` and `@done`:

```
clear:  LDX #0
@loop:  STA SCREEN,X
        INX
        BNE @loop       ; clear's @loop
        RTS
fill:   LDX #39
@loop:  STA SCREEN,X    ; a different @loop: fill's
        DEX
        BPL @loop
        RTS
```

Only labels with a colon start a new scope; constants don't. In the symbol list and the debugger, a local label shows as `clear@loop`.

Numbers and expressions

Write	Value
`$C020`	Hexadecimal, up to 16 digits (64 bits).

Write	Value
`%0101`	Binary, up to 64 digits.
`42`	Decimal.
`'A'`	The character code of A: 65. `'\''` is a quote and `'\\'` a backslash.
`loop`	A label or constant.
`*`	The address of the current instruction.
`<expr`	The low byte of everything after it.
`>expr`	The high byte of everything after it.
`a + b`, `a - b`, `a * b`, `a / b`	Arithmetic. Division rounds towards zero.
`a % b`	The remainder of `a / b`.
`a & b`, `a | b`, `a ^ b`	Bitwise AND, OR and exclusive OR.
`a << n`, `a >> n`	Shift left or right by `n` bits (0 to 63).
`a < b`, `a <= b`, `a > b`, `a >= b`, `a == b`, `a != b`	Comparisons: 1 if true, 0 if false. Handy in `.assert`.
`a && b`, `a || b`	Logical AND and OR: 1 if both (either) are not 0, else 0.
`-a`, `~a`	Negative; every bit flipped.
`[a + b] * 2`	Square brackets group. Round ones mean indirect addressing, or a function's values.
`sin(a, n)`, `sqrt(x)`, ...	Functions: see below.

The operators bind as they do in C, tightest first:

Binds	Operators
tightest	`-a ~a`
	`* / %`
	`+ -`
	`<< >>`
	`< <= > >=`
	`== !=`
	`&`
	`^`
	`|`
	`&&`
loosest	`||`
	`<expr` and `>expr`: everything after them

So `1 + 2 * 3` is 7 and `$F0 & $3C | 1` is `$31`. Values are worked out in 64 bits, two's complement; only where a value is stored does it have to fit a byte or a word. A hex or binary number can use all 64 bits, so an 8×8 sprite fits in one constant, a row per byte. Past `$7FFFFFFFFFFFFFFF` the top bit makes it negative, so take bytes out with `<` or `& $FF`, not by shifting alone:

```
BUG     = $81423C5A3C993C42     ; the top row is $81
        .rept 8, i
        .byte <([BUG >> [56 - 8 * i]]
        .endr
```

`&&` and `||` work out both sides every time. On comparisons, which are always 0 or 1, `&` and `|` do the same job as `&&` and `||`: `.assert x >= 0 & x < 40` is fine.

`<` and `>` are how you put an address into two registers:

```
        LDX #<message   ; low byte
        LDY #>message   ; high byte
```

They take everything after them, so `#<table+1` is the low byte of `table+1`, which is what you mean when `table+1` crosses a page. To take the byte of only part of an expression, put brackets round it: `[<table] + 1`.

Precedence traps

Two lines that look right and aren't:

```
        .assert MYSET & $FF == 0        ; is MYSET & [$FF == 0]: always 0
        .assert <BOARD == 0             ; is the low byte of [BOARD == 0]
```

C's order puts `==` above `&`, and `<` takes everything after it. Bracket the part you mean: `.assert [MYSET & $FF] == 0` and `.assert [<BOARD] == 0`. The assembler warns about both shapes: a `&`, `|` or `^` with a comparison on one side only, and a `<` or `>` in front of a comparison. A warning doesn't stop the program assembling; the command-line assembler prints it, and the listing (Appendix G) shows it as a note on the line. When the trap makes an `.assert` fail, as both of these do, the warning is added to the error: `assertion failed: MYSET & $FF == 0 (warning: '==' binds tighter than '&': add [ ] to say which you mean)`. A comparison on *both* sides of `&` or `|` is a logical AND or OR and gets no warning, and neither does anything in brackets.

Some signs mean two things, and the place decides. Where the assembler expects a value (at the start, after an operator or after `[`), `*` is the current address, `%` starts a binary number, and `<` and `>` take a byte. After a value, they're operators: multiply, remainder, less than and greater than. So `* - 2` is two bytes back, `2 * *` is twice the address, `%0101` is 5, and `17 % 5` and `17 %5` are both 2.

Functions

Write	Value
`sin(a, n)`	n × sine of the angle a, rounded to the nearest whole number. The angle is in 256ths of a circle: 64 is a right angle, and any a works (it wraps). `sin(64, 127)` is 127.
`cos(a, n)`	n × cosine, the same way: `cos(a, n)` is `sin(a + 64, n)`.
`sqrt(x)`	The square root of x, rounded down: `sqrt(80)` is 8. x can't be negative.
`abs(x)`	x without its sign.
`min(a, b)`, `max(a, b)`	The smaller and the larger.

A name followed by `(` is a function; without the `(` it's an ordinary name, so a label called `min` still works. Case doesn't matter: `SIN(a, n)` is `sin(a, n)`. A `(` that starts an operand still means indirect addressing: `JMP (vec)` jumps through `vec`, and `JMP (max(a, b))` through whichever is larger. Anywhere else, round brackets belong to a function, and grouping is `[ ]`.

The functions use whole numbers only, so a program assembles to the same bytes on every computer. `sin` and `cos` read a table of a quarter circle held to 62 binary places and round halves away from zero, so `sin(-a, n)` is exactly `-sin(a, n)` and `sin(128 - a, n)` is `sin(a, n)`. For every angle and every n up to 1,048,576 either way, the answer is the true value rounded; beyond that it can be one out, in the rare case the true value is within $n/2^{63}$ of a half.

A table of the sine across one screen width, worked out by the assembler and drawn by the program:

```
; A sine wave across the screen, from a table the assembler works out.
SCREEN  = $0400
PTR     = $10

        .org $0800
start:  LDX #39
@col:   LDY wave,X          ; the row for this column
        LDA rowlo,Y         ; point PTR at the row
        STA PTR
        LDA rowhi,Y
        STA PTR+1
        TXA
        TAY                 ; Y = the column
        LDA #'*'
        STA (PTR),Y
        DEX
        BPL @col
        HLT

wave:   .rept 40, col       ; one turn over 40 columns, 12 rows each side of the middle
        .byte 12 - sin(col * 256 / 40, 12)
        .endr
rowlo:  .rept 25, row
        .byte <[SCREEN + row * 40]
        .endr
rowhi:  .rept 25, row
        .byte >[SCREEN + row * 40]
        .endr
```

`col * 256 / 40` turns a column into an angle, and rows grow downwards, so the sine is taken from the middle row, 12.

Directives

Directive	Does
`.org $0800`	Assemble what follows at this address. The default is `$0800`.
`.byte 1, 2, $FF`	Store bytes. `.db` is the same.
`.text "HELLO"`	Store the characters of a string, as their ASCII codes. `.ascii` is the same, and so is a string inside `.byte`. Write in capitals: screen codes have no lowercase, so `.text "abc"` stores `$61-$63`, which show as graphics.
`.asciiz "HELLO"`	Store a string followed by a zero byte.
`.word $C020, loop`	Store words, low byte first. `.dw` is the same.
`.fill 10, $20`	Store 10 copies of `$20`. Without a value, zeros.
`.align 256`	Move on to the next address that's a multiple of 256. Between stored bytes the gap is stored as zeros; next to reserved space it's reserved (see below). `.align 256, $EA` always stores `$EA`s.
`.res 40`	Reserve 40 bytes: the address moves on, but nothing is stored. `.dsb` is the same.
`.rept 8` … `.endr`	Assemble the lines in between 8 times.
`.irp s, B0, B1` … `.endr`	Assemble the lines in between once for each value, with `s` standing for it.
`.include "file.s"`	Assemble another file's lines here.
`.assert expr, "message"`	Stop with the message if `expr` is 0. Without a message, `assertion failed: expr`.

Inside a string, \" is a quote and \\ a backslash: `.text "SAY \"HI\""`. Any other backslash is an error.

Reserving memory. `.res` gives names to memory without filling it in: buffers, tables built at run time, the second half of a screen. Reserved bytes aren't part of the program, so loading it doesn't touch them.

```
        .org $3000
board:  .res 200        ; filled in by the program
next:   .res 1
```

Aligning. `.align` stores its padding only when it's *between stored bytes*: the last thing before it that moves the address stored something (an instruction, `.byte`, `.fill`…), and so does the next thing after it. Otherwise (after `.org` or `.res`, before `.res`, or with nothing stored after it until the next `.org` or the end) it reserves the padding like `.res`, and nothing is loaded there. Labels, constants and `.assert` don't count. With a value, `.align n, value` always stores. So a page-aligned buffer among other buffers costs nothing in the program:

```
board:  .res 200
        .align 256      ; reserved: a gap before and after
charset: .res $400
```

Repeating. `.rept` takes a count the assembler knows at that point, and an optional counter. The counter's name stands for 0, 1, 2 and so on in each copy:

```
        .rept 25, row   ; the address of each screen row
        .word $0400 + row * 40
        .endr
```

The counter is replaced in the copied lines wherever it appears as a whole name, outside strings and comments, so give it a name that isn't used for anything else (and not `A`, `X` or `Y`). `.rept` blocks can be nested. Labels inside a block are defined once per copy, so they'd be defined twice: keep labels outside, use `*`, or paste the counter into the name (below).

`.irp` steps through a list instead of counting. The name stands for each value in turn:

```
        .irp s, BUG0, BUG1, BUG2        ; the same three lines for each sprite
        LDA s+1
        STA s
        DEC s+2
        .endr
```

The values are expressions or strings, separated by commas (a comma inside brackets belongs to the value). The name is replaced the same way as a `.rept` counter. A value that's more than one name, number or string goes in as though bracketed, so with `.irp v, 1 + 2`, `v * 2` is 6. An `.irp` with no values makes no copies. `.irp` and `.rept` nest inside each other, to any depth.

Pasting. Inside a `.rept` or `.irp`, `\name` pastes the counter or value onto the name before it, so each copy can have its own labels:

```
        .rept 25, i
ROW\i:  .res 40         ; ROW0, ROW1, ... ROW24
        .endr
```

The value must be a plain word or number to paste (`\v` with `v = 1 + 2` is an error), and it ends the name: `ROW\i` works, `R\iX` doesn't. Nested blocks paste in turn: `T\i\j` makes `T00`, `T01` and so on.

Including. `.include "file.s"` reads the file's lines as though they were typed in its place. A file can include others, up to 16 deep. The command-line assembler (Appendix G) finds the file next to the program it's assembling. In the Machine Room there are no files, and `.include` says `includes aren't available here`. An error in an included file is reported with the file's name and its own line number.

Checking. `.assert` is checked once every symbol is known, so it can test labels further down:

```
        .assert end <= $1A00, "the code runs into the tables at $1A00"
```

Fast page or absolute?

When an operand is a value below 256 that the assembler already knows, it uses the fast-page form if the instruction has one. To force the absolute form, write the address with three or four hex digits: `LDA $0010`. A label *or a constant* defined *later* in the program is always assembled as absolute, because on its first reading of the program the assembler doesn't know its value yet. So define fast-page variables as constants near the top.

An instruction that has no fast-page form for the way you wrote it gets the absolute form, with the same address. The one that catches people is `fp,Y`: only `LDX` and `STX` have a fast page,Y mode, so `LDA $10,Y` assembles as `LDA $0010,Y`, three bytes and absolute,Y's cost. The listing (Appendix G) marks every place this happens.

Where the program starts

When your program is loaded, it runs from the label `start` if there is one, or from its first byte if not. The Machine Room (and every other way of loading a program) starts it the same way:

PC	`start`
SP	`$FF`: the stack is empty
P	`$24`: I set, so interrupts are off until you `CLI`; the other flags clear
A, X, Y	0

Memory and the I/O registers are as the ROM left them: the screen shows BASIC's start-up message and `READY.` (or, from a tape, `LOAD`'s messages: chapter 13), the border and background are black, and no interrupt sources are enabled.

When something is wrong

The assembler reads the whole program, then lists every line it couldn't assemble, with a plain message. Appendix E lists them all.

```
line 12: STA has no immediate mode
line 19: branch out of range (140 bytes)
line 40: address $0900 is written twice (first by line 7)
```

The last one means two parts of the program, each with its own `.org`, put bytes at the same address. It's reported on the line that writes the address the second time.

A **warning** is for a line that assembles but probably doesn't mean what it says: the precedence traps above. The program is still assembled. The command-line assembler prints `file:line: warning: message`, and the listing shows the message as a note.

6. Your first programs

Each program here is complete. Type it in, assemble it and run it.

Hello

```
; Print a message, then stop.
PRSTR   = $FF09

        .org $0800
start:  LDX #<msg       ; X = low byte of the message's address
        LDY #>msg       ; Y = high byte
        JSR PRSTR       ; the ROM prints up to the zero byte
        HLT

msg:    .text "HELLO FROM THE UNC 64!"
        .byte 13, 0     ; 13 = new line; 0 = the end
```

`PRSTR` is one of the system ROM's routines (chapter 12). `HLT` stops the processor, so it doesn't run on into the message and try to execute the letters.

Loops: filling the screen

The screen is 1,000 bytes of memory starting at `$0400` (chapter 7). Store a character code there and it appears. To fill the screen you need a loop, but X only counts to 255. So this loop stores to four places on each pass:

```
; Fill the screen with stars.
SCREEN  = $0400

        .org $0800
start:  LDA #'*'
        LDX #0
fill:   STA SCREEN,X        ; $0400-$04FF
        STA SCREEN+$100,X   ; $0500-$05FF
        STA SCREEN+$200,X   ; $0600-$06FF
        STA SCREEN+$2E8,X   ; $06E8-$07E7: overlaps the third, ends exactly at the last cell
        INX
        BNE fill            ; X wraps from $FF to 0: 256 passes
        HLT
```

Count the cycles. One pass of the loop is four stores at 5 cycles, `INX` at 2 and a taken `BNE` at 3: 25 cycles. 256 passes, less one for the last branch not being taken, is 6,399 cycles. That's 6.4 thousandths of a second, about a third of a screen refresh (chapter 8). Counting like this is how you'll make programs fast.

Counting in hex

```
; Print every byte value from $00 to $FF.
CHROUT  = $FF03
PRHEX   = $FF0C

        .org $0800
start:  LDX #0
loop:   TXA
        JSR PRHEX       ; print A as two hex digits
        LDA #' '
        JSR CHROUT      ; print a space
        INX
        BNE loop
        HLT
```

The ROM routines keep X and Y as they were, so X survives every call.

Sixteen-bit arithmetic

A byte only counts to 255. For bigger numbers, use two bytes and let the carry link them:

```
; RESULT = NUM1 + NUM2, all 16-bit.
        .org $0800
start:  CLC             ; no carry into the low byte
        LDA num1
        ADC num2        ; add the low bytes; a carry out sets C
        STA result
        LDA num1+1
        ADC num2+1      ; add the high bytes and the carry
        STA result+1
        HLT

num1:   .word 1000
num2:   .word 2345
result: .word 0         ; ends up as 3345 ($0D11)
```

Subtraction is the same with `SEC` and `SBC`.

Multiplying

The UNC-1 has no multiply instruction. You multiply the way you learned at school, but in binary: for each bit of one number, if it's 1, add the other number, then shift.

```
; 8-bit x 8-bit = 16-bit multiply. 200 x 200 = 40000 ($9C40).
M1      = $10           ; first number; becomes the product's low byte
M2      = $11           ; second number
HI      = $12           ; the product's high byte

        .org $0800
start:  LDA #200
        STA M1
        STA M2
        LDA #0          ; A collects the high byte
        LDX #8          ; eight bits
        LSR M1          ; the lowest bit of M1 into the carry
mloop:  BCC noadd
        CLC
        ADC M2          ; that bit was 1: add M2
noadd:  ROR A           ; shift the 16-bit result right...
        ROR M1          ; ...and the next bit of M1 into the carry
        DEX
        BNE mloop
        STA HI
        HLT
```

The trick is that the product's low byte is shifted into M1 from the top as M1's own bits leave at the bottom. Trace it with a pencil for 3 × 5. It's worth the ten minutes.

Subroutines: printing a number in decimal

Printing a byte in decimal means dividing by 100 and then by 10. Division is repeated subtraction. This program puts the work in a subroutine, so you can call it from anywhere:

```
; Print 142 in decimal.
CHROUT  = $FF03
DIV     = $10

        .org $0800
start:  LDA #142
        JSR prdec
        HLT

; PRDEC: print A (0-255) in decimal, without leading zeros.
; Changes A, X and Y.
prdec:  LDY #0          ; Y becomes 1 once a digit is printed
        LDX #100
        JSR digit       ; hundreds
        LDX #10
        JSR digit       ; tens
        CLC
        ADC #'0'        ; what's left is the ones, always printed
        JMP CHROUT      ; CHROUT's RTS returns to our caller

; DIGIT: print how many times X goes into A; leave the remainder in A.
digit:  STX DIV
        LDX #'0'        ; X counts, starting at the character '0'
dloop:  CMP DIV
        BCC dout        ; A < DIV: done
        SBC DIV         ; the carry is already set by CMP
        INX
        JMP dloop
dout:   CPX #'0'
        BNE dprint
        CPY #0
        BEQ ddone       ; a leading zero: don't print it
dprint: PHA             ; keep the remainder
        TXA
        JSR CHROUT
        PLA
        LDY #1
ddone:  RTS
```

`JMP CHROUT` at the end of `prdec` is a **tail call**: instead of `JSR CHROUT` followed by `RTS`, jump straight to it and let its `RTS` do the returning. It saves a byte and 9 cycles.

The stack in a subroutine call

`JSR` pushes the return address; `RTS` pulls it. With SP at `$FF`, the `JSR prdec` at `$0802` does this:

```
before:  SP = $FF
JSR:     $01FF <- $08   (high byte of $0805, the next instruction)
         $01FE <- $05   (low byte)
         SP = $FD
RTS:     pulls $05, then $08: PC = $0805
```

A subroutine that pushes must pull the same number of bytes before it returns, or `RTS` returns to the wrong place.

7. The screen

Screen memory

The screen shows 25 rows of 40 characters. It's drawn from 1,000 bytes of memory at `$0400-$07E7`, one byte per character cell, row by row:

```
address of (column, row) = $0400 + row * 40 + column

row 0:   $0400-$0427
row 1:   $0428-$044F
...
row 24:  $07C0-$07E7
```

The screen is ordinary memory. Read it, write it, and run a program out of it if you like. Whatever is there, the display shows, fifty times a second.

Screen codes

Each byte is a **screen code**. The video chip draws each code as an 8 × 8 **glyph** from the character set, **UNSCII**. The full table is in Appendix A.

Codes	Show
`$00-$0F`	Quarter blocks: bits 0-3 light the top-left, top-right, bottom-left and bottom-right quarter of the cell.
`$10-$1F`	Lines, corners and joints for drawing boxes; diagonals; two shades.
`$20-$5F`	Capital letters, digits and punctuation, in ASCII order.
`$60-$6F`	Eighth blocks, for bar graphs and smooth edges.
`$70-$7F`	Card suits, a ball, a ring, arrows, triangles, a face, a note.
`$80-$FF`	The glyphs of `$00-$7F` in reverse video.
`$A0`	A reversed space, which blinks (unless VMODE bit 3 is set; see below). BASIC's line editor uses it as the cursor.

The quarter blocks are numbered so you can work them out: `$0F` is a solid block, `$03` the top half of a cell, `$0C` the bottom half. With them, the 40 × 25 screen becomes an 80 × 50 grid of dots. There are no lowercase letters; ASCII's lowercase codes are graphics here.

The character set

This program shows every screen code in a 16 × 16 grid. Find a code by its row (the high digit) and column (the low digit):

```
; Show all 256 screen codes in a 16 x 16 grid.
; The codes go straight into screen memory: printing them with CHROUT
; would act on 8, 12 and 13 instead of showing them.
CLS     = $FF00
CHROUT  = $FF03
PRSTR   = $FF09
PRHEX   = $FF0C
SCREEN  = $0400
ROW     = $10           ; pointer to the current grid row
CODE    = $12           ; the next code to show

        .org $0800
start:  JSR CLS
        LDX #<head
        LDY #>head
        JSR PRSTR       ; title and column digits: rows 0-2
        LDA #0
labels: PHA             ; row labels $00-$F0: rows 3-18
        JSR PRHEX
        LDA #13
        JSR CHROUT
        PLA
        CLC
        ADC #16
        BNE labels      ; $F0 + 16 wraps to 0: done
        LDA #<[SCREEN+3*40+4]
        STA ROW         ; the grid starts at column 4 of row 3
        LDA #>[SCREEN+3*40+4]
        STA ROW+1
        LDA #0
        STA CODE
grid:   LDY #0          ; Y steps two columns per code
cell:   LDA CODE
        STA (ROW),Y
        INC CODE
        INY
        INY
        CPY #32
        BNE cell
        CLC
        LDA ROW         ; next row: 40 bytes on
        ADC #40
        STA ROW
        LDA ROW+1
        ADC #0
        STA ROW+1
        LDA CODE
```

```
        BNE grid        ; CODE wraps to 0 after 256 codes
        HLT

head:   .text "UNC 64 CHARACTER SET"
        .byte 13, 13
        .text "    0 1 2 3 4 5 6 7 8 9 A B C D E F"
        .byte 13, 0
```

Border and background

The screen sits inside a coloured **border**. The 25 rows sit on a **background**. Both colours are I/O registers:

Address	Register	Value
`$C020`	BORDER	Colour 0-15.
`$C021`	BACKGROUND	Colour 0-15.

Only the low four bits count. The sixteen colours are in Appendix A. The ROM sets both to 0, black, at start-up. The characters themselves are drawn in the monitor's own colour: phosphor green on the monochrome monitor, white on a colour one.

The border is drawn one raster line at a time, so a program that changes the border colour fast enough can draw stripes across it (chapter 8). The background is a single colour for the whole frame.

UNSCII, the character set

UNSCII is 128 glyphs, built into the character ROM at `$D000-$D3FF`. Each glyph is 8 bytes, one per row from the top, and each bit is a pixel: 1 is ink, 0 is paper, bit 7 is the leftmost. Glyph *n* starts at `$D000 + n * 8`. Here is the letter A, at `$D208`:

```
$D208   %00111000   ..###...
$D209   %01101100   .##.##..
$D20A   %11000110   ##...##.
$D20B   %11000110   ##...##.
$D20C   %11111110   #######.
$D20D   %11000110   ##...##.
$D20E   %11000110   ##...##.
$D20F   %00000000   ........
```

The letters use the top seven rows and the left seven columns, so that neighbouring characters never touch. The graphics glyphs use the whole cell, so they join up.

Your own characters

The video chip doesn't have to read UNSCII. The CHARSET register says where the character set is:

Address	Register	Read and write
$C022	CHARSET	High byte of the character set's address. At reset, $D0: UNSCII.

Copy UNSCII into memory, change the glyphs you want, and point CHARSET at your copy. The set must start on a page boundary, and it takes 1K. A trick of the era is to replace glyphs you don't need: a wall, a tree, a door and a few creatures, and the text screen becomes a game map that the processor can redraw by changing single bytes.

```
; Custom characters: copy UNSCII to RAM, redraw one glyph as an owl.
SCREEN  = $0400
CHARSET = $C022
CHARROM = $D000
MYSET   = $3800         ; 1K at the top of a UNC 16's memory
OWL     = $7F           ; the glyph to redraw: the note

        .org $0800
start:  LDX #0
copy:   LDA CHARROM,X       ; copy all 1,024 bytes, a quarter per pass
        STA MYSET,X
        LDA CHARROM+$100,X
        STA MYSET+$100,X
        LDA CHARROM+$200,X
        STA MYSET+$200,X
        LDA CHARROM+$300,X
        STA MYSET+$300,X
        INX
        BNE copy
        LDX #7
def:    LDA owl,X           ; eight rows, bottom row first
        STA MYSET+OWL*8,X
        DEX
        BPL def
        LDA #>MYSET
        STA CHARSET         ; the video chip now reads our set
        LDX #39
        LDA #OWL
row:    STA SCREEN+12*40,X  ; a row of owls
        DEX
        BPL row
        HLT

owl:    .byte %01000010
        .byte %01111110
        .byte %11011011
        .byte %11111111
        .byte %01100110
        .byte %01111110
        .byte %00100100
        .byte %00000000
```

The screen doesn't change until the last moment, when CHARSET switches: every glyph in the copy is the same as the ROM's except the owl.

A change to a glyph shows on every cell that uses it, at once. Change one glyph eight times a second and every flame on the screen flickers; change it a frame at a time and the water ripples. The processor touches eight bytes, not hundreds.

What the copy costs. The loop above copies 1K in 256 passes of 45 cycles: about 11,500 cycles, more than half a frame. Do it once, at start-up, before the game's first frame.

Glyph `$00` as the empty cell. A game that keeps its board in memory, a byte per cell, usually uses 0 for "empty". Redraw glyph `$00` as whatever an empty cell looks like (paper, a dot, a dark floor tile) and the board can be copied straight to the screen, a byte at a time, with no translating. Glyph `$00` is a blank quarter-block cell in UNSCII, so nothing else is lost.

A steady cursor

The video chip blinks code `$A0` by itself, about once a second, so the cursor BASIC's line editor shows while it waits for you to type (chapter 12) needs no program to blink it. Printing never draws it, so a program only sees `$A0` on the screen if BASIC was waiting when it started, or if it stores one itself. A game that wants a solid reversed block instead sets VMODE bit 3:

Address	Register	Bit 3
`$C000`	VMODE	0: `$A0` blinks (the default, and what the ROM expects). 1: `$A0` is a steady reversed space, like every other reversed character.

Set it with `ORA`, so the other VMODE bits stay as they are:

```
        LDA VMODE
        ORA #%00001000  ; bit 3: no blink
        STA VMODE
```

256 glyphs

Reverse video costs half the character set. For a game that wants more glyphs, set bit 0 of VMODE:

Address	Register	Bit 0
`$C000`	VMODE	0: 128 glyphs, codes `$80-$FF` in reverse video (the default). 1: 256 separate glyphs from a 2K set, no reverse video, no blinking cursor.

In 256-glyph mode, the set is 2K long and glyph *n* starts at CHARSET × 256 + *n* × 8. The ROM's own printing still works, but the line editor's cursor is whatever you draw for code `$A0`.

You need your own set. UNSCII is only 1K. With CHARSET at `$D0`, glyphs `$80-$FF` would come from `$D400-$D7FF`, which reads `$FF`: every one of them a solid block. So 256-glyph mode starts with a 2K set in RAM. This program builds one, UNSCII in the bottom half and its reversed copy in the top half, and draws a box with the line glyphs `$10-$1A`:

```
; A box in 256-glyph mode, from a 2K set in RAM.
SCREEN  = $0400
VMODE   = $C000
CHARSET = $C022
CHARROM = $D000
MYSET   = $3000             ; 2K: $3000-$37FF
CORNER  = SCREEN+10*40+18   ; the box's top-left cell

        .org $0800
start:  LDX #0
copy:   .rept 4, page       ; glyphs $00-$7F: UNSCII. $80-$FF: the same, reversed.
        LDA CHARROM+page*$100,X
        STA MYSET+page*$100,X
        EOR #$FF
        STA MYSET+$400+page*$100,X
        .endr
        INX
        BNE copy
        LDA #>MYSET
        STA CHARSET
        LDA #%00000001      ; VMODE bit 0: 256 glyphs
        STA VMODE
        LDX #4
draw:   .rept 5, row        ; five rows of five cells
        LDA box+row*5,X
        STA CORNER+row*40,X
        .endr
        DEX
        BPL draw
        HLT

box:    .byte $12, $10, $18, $10, $13   ; top: corners and a T
        .byte $11, ' ', $11, ' ', $11   ; sides
        .byte $16, $10, $1A, $10, $17   ; middle: joints and a cross
        .byte $11, ' ', $11, ' ', $11
        .byte $14, $10, $19, $10, $15   ; bottom
```

The line glyphs run through the middle of their cells and reach the edges, so the eleven of them join into any box: `$10` across, `$11` down, `$12-$15` the corners, `$16-$19` the T joints and `$1A` the cross.

A first animation

```
; Move a block along row 12, one cell per frame.
SCREEN  = $0400
FRAME   = $C006

        .org $0800
start:  LDX #0
loop:   LDA #$0F            ; a solid block
        STA SCREEN+12*40,X
        LDA FRAME           ; wait for the frame counter to change
wait:   CMP FRAME
        BEQ wait
        LDA #' '            ; rub the block out
        STA SCREEN+12*40,X
        INX
        CPX #40
        BNE loop
        LDX #0
        JMP loop
```

The frame counter at `$C006` goes up by one every time the screen is redrawn (chapter 8), so waiting for it to change paces the program at fifty steps a second, exactly. Without the wait, the block would cross the screen in a fraction of a frame and you'd see nothing.

The bitmap

Text and block graphics go a long way, but sometimes you want to draw: a line at any angle, a circle, a single dot. For that the video chip has a **bitmap**: the screen as 320 × 200 separate pixels, each one bit, ink or paper. It needs no upgrade; every UNC 16 has it.

Address	Register	Bits
`$C000`	VMODE	Bit 1: 1 shows the bitmap instead of the text. Bit 2: **mixed mode**, see below. Bit 0 is the 256-glyph switch and bit 3 the steady cursor.
`$C023`	BITMAP	High byte of the bitmap's address. At reset, `$20`: the bitmap is `$2000`-`$3F3F`.

The bitmap is 8,000 bytes of ordinary memory, 40 bytes to a line, line after line from the top. Each byte is eight pixels side by side, and bit 7 is the leftmost:

```
byte = BITMAP * 256 + y * 40 + x / 8
bit  = 7 - (x AND 7)
```

So the pixel at (100, *57*) is in byte `$2000 + 57 * 40 + 12` = `$28F4`, bit 3. Set that bit and the pixel lights; clear it and it goes dark. Pixels are drawn in the same ink and paper as text: phosphor green on black on the green monitor, and the BACKGROUND colour behind white ink on a colour one.

```
; Plot one pixel at (100, 57) on the bitmap, then stop.
VMODE   = $C000
BITMAP  = $C023
PTR     = $10
PIXEL   = $2000+57*40+100/8 ; the byte: $28F4
MASK    = $80/16            ; bit 7 - (100 AND 7) = bit 3: $08

        .org $0800
start:  LDA #$20
        STA BITMAP          ; the bitmap at $2000 (where it is after reset anyway)
        STA PTR+1
        LDA #0
        STA PTR
        TAY
        LDX #31             ; paper: 31 whole pages, $2000-$3EFF
clear:  STA (PTR),Y
        INY
        BNE clear
        INC PTR+1
        DEX
        BNE clear
        LDY #$3F            ; and the last 64 bytes, $3F00-$3F3F:
last:   STA $3F00,Y         ; 8,000 in all, and not a byte past the bitmap
        DEY
        BPL last
        LDA PIXEL
        ORA #MASK           ; light one bit, leave its seven neighbours alone
        STA PIXEL
        LDA #%00000010      ; VMODE bit 1: show the bitmap
        STA VMODE
        HLT
```

The `ORA` is the important part: a byte holds eight pixels, so to light one you change one bit and keep the other seven. `AND` with the mask inverted turns one off; `EOR` flips it.

Mixed mode. With VMODE bits 1 and 2 both set, the bitmap shows only its top 168 lines, and below them the last four rows of the text screen, rows 21 to 24, appear as usual. That's how UNC BASIC lets you draw and type at the same time (chapter 14). The split is at raster line 219 (chapter 8): lines 51-218 show bitmap lines 0-167, and lines 219-250 show text rows 21-24.

The bitmap's bottom 32 lines are still in memory; they're just covered up. In mixed mode nothing shows them, so those 1,280 bytes, BITMAP × 256 + `$1A40` to BITMAP × 256 + `$1F3F`, are free RAM for tables and variables. With the 192 bytes after the bitmap in its last page, that's 1,472 bytes: `$3A40-$3FFF` at the default address. Clear the bitmap with care, then: 8,000 bytes, not 32 whole pages, as the listing above does.

Where the bitmap can be. BITMAP can point anywhere. Point it past the memory a machine has (`$40` or above on a UNC 16) and the video chip reads `$FF`: a screen of solid ink.

What it costs. 8,000 bytes is half of a UNC 16. With the bitmap at `$2000`, programs have `$0800-$1FFF`, 6K, below it. A UNC 64 can put it at `$A000` (that's where BASIC puts it) and keep 38K free underneath. Drawing in hi-res is one of the best reasons there is to save up for the 64K expansion.

Shapes at any x: pre-shifted sprites. A shape eight pixels wide drawn at an x that's a multiple of 8 covers one byte per line. Anywhere else, it straddles two bytes, and shifting it into place as you draw costs eight shifts per line per byte. The trick of the era is to do the shifting once. At start-up, build eight copies of the shape, each one shifted a pixel further right and two bytes wide; a game that moves in 2-pixel steps needs only four. To draw at x, pick the copy for x AND 7, and store its bytes at x / 8 and x / 8 + 1 on each line. Draw with `EOR` and the same call rubs the shape out again, leaving the background as it was; draw with `ORA` and rub out with `AND` and the inverted mask. The tables cost memory (eight copies of an 8 × 8 shape is 128 bytes) and save thousands of cycles a frame. Or let the hardware do the shifting: see the shifter, below.

Tearing. The beam draws the picture from line 51 down, every frame. FRAME goes up at line 0, so a program that starts drawing when FRAME changes races the beam: if the beam reaches a shape before the program has finished moving it, the top of the picture shows the new frame and the bottom the old, or the shape shows half rubbed out. The cure is to change the picture where the beam isn't. Start the frame's drawing from a raster interrupt below the part you draw (chapter 9): in mixed mode, line 219, where the bitmap ends. From there, the beam doesn't reach the bitmap again until line 51 of the next frame, 144 lines (about 9,200 cycles) later.

The shifter

Pre-shifted copies cost memory, and shifting as you draw costs time, more the further the shape is from a byte boundary. The UNC 64 has a third way, the one the arcade boards of the late 1970s used: a **shifter**, a little circuit that moves a byte's pixels across in no time at all, by any amount from 0 to 7.

Address	Register	Read	Write
`$C050`	SHIFT_IN		Push a byte into the window.
`$C051`	SHIFT_AMOUNT	The amount, 0-7.	How many pixels to move right. Only bits 0-2 count, so storing x itself sets x AND 7.
`$C052`	SHIFT_OUT	The window moved right by SHIFT_AMOUNT: the next byte for the bitmap.	
`$C053`	SHIFT_FLIP		Push a byte with its pixels in reverse order: see "Mirror images".

The shifter holds a **window** of 16 pixels: the last two bytes pushed, side by side, the newer one on the right. Each push moves the newer byte to the left half, drops the older one, and puts the new byte on the right. SHIFT_OUT is eight pixels of the window, starting SHIFT_AMOUNT pixels left of the newest byte:

```
window    = (byte pushed before) * 256 + (byte pushed last)
SHIFT_OUT = (window >> SHIFT_AMOUNT) AND $FF
```

To draw a row of a shape at *x*, set SHIFT_AMOUNT to *x*, push the row's byte and read SHIFT_OUT: that's the byte moved right by *x* AND 7, the part that goes in byte *x* / 8. Push a 0 and read again: the part that spilled over, for byte *x* / 8 + 1. The next row's push drops the old byte out of the window, so every row starts clean. A shape 16 pixels wide pushes its two bytes and then a 0, reading SHIFT_OUT after each: three bytes on the screen.

```
; Draw an 8 x 8 sprite at any x with the shifter: five balls on line 60.
VMODE     = $C000
SHIFT_IN  = $C050
SHIFT_AMT = $C051
SHIFT_OUT = $C052
LINE60    = $2000+60*40     ; where line 60 of the bitmap starts
PTR       = $10             ; the byte under the sprite's top-left corner
XLO       = $12             ; the sprite's x, 0-312
XHI       = $13
N         = $14             ; which ball

        .org $0800
start:  LDA #%00000010      ; show the bitmap: after switching on, it's all paper
        STA VMODE
        LDA #0
        STA N
next:   LDX N
        LDA xlo,X
        STA XLO
        LDA xhi,X
        STA XHI
        JSR sprite
        INC N
        LDA N
        CMP #5
        BNE next
        HLT

; Draws the ball with its top-left corner at (XHI:XLO, 60).
sprite: LDA XLO
        STA SHIFT_AMT       ; the shifter keeps bits 0-2: x AND 7
        LDA XHI             ; x / 8: three 16-bit shifts right
        LSR A
        LDA XLO
        ROR A
        LSR A
        LSR A
        CLC
        ADC #<LINE60        ; PTR = LINE60 + x / 8
        STA PTR
        LDA #>LINE60
        ADC #0
        STA PTR+1
        LDX #0
@row:   LDY #0
        LDA ball,X
        STA SHIFT_IN        ; the sprite's row into the window
```

```
        LDA SHIFT_OUT       ; the part that lands in the first byte
        ORA (PTR),Y
        STA (PTR),Y
        STY SHIFT_IN        ; push paper after it (Y is 0)
        LDA SHIFT_OUT       ; the part that spills into the next byte
        INY
        ORA (PTR),Y
        STA (PTR),Y
        LDA PTR             ; down a line: 40 bytes on
        CLC
        ADC #40
        STA PTR
        BCC @same
        INC PTR+1
@same:  INX
        CPX #8
        BNE @row
        RTS

xlo:    .byte <3, <77, <160, <254, <312
xhi:    .byte >3, >77, >160, >254, >312
ball:   .byte %00111100
        .byte %01111110
        .byte %11111111
        .byte %11111111
        .byte %11111111
        .byte %11111111
        .byte %01111110
        .byte %00111100
```

Every row takes 69 cycles (73 when the pointer crosses a page), whatever x is: about 600 cycles a ball. The shifter's own share is two pushes and two reads, 16 cycles a row. Doing the same shift with `LSR` and `ROR` costs from nothing at x AND 7 = 0 to over 80 cycles at 7, so a game's frame would take longer whenever a sprite sat at the wrong place.

Drawing and rubbing out. The listing draws with `ORA`, which lights the shape's pixels and leaves the rest alone. Draw with `EOR` instead, and drawing the same shape at the same place again rubs it out, leaving the background as it was. To clear just the shape's pixels, `AND` each byte with SHIFT_OUT `EOR #$FF`.

Mirror images. SHIFT_FLIP pushes a byte back to front: bit 7 becomes bit 0, and the leftmost pixel the rightmost. Push a shape's rows through SHIFT_FLIP instead of SHIFT_IN and it's drawn facing the other way, from the same bytes, so a ship that flies left and right needs one copy. A shape wider than 8 pixels pushes its bytes right-hand byte first. With SHIFT_AMOUNT at 0, SHIFT_FLIP then SHIFT_OUT reverses any byte's bits in 8 cycles.

At the right edge. Byte x / 8 + 1 of the last column is the first byte of the next line. At x = 312 nothing spills (the shift is 0), but from 313 to 319 the spill would light pixels at the left edge one line down. Keep x to 312 or less, or skip the spill byte in the last column.

One shifter. The window is shared. If a program draws with the shifter both in its main loop and in an interrupt handler, the handler's pushes land in the middle of the main loop's rows. Draw in one place, or set I around each row.

8. Timing and the raster

How the picture is made

The display draws the picture as **raster lines** from top to bottom, 50 times a second. The UNC 64's timing is fixed and simple:

Processor clock	1,000,000 cycles a second
One raster line	64 cycles
One frame	312 lines = 19,968 cycles
Frame rate	1,000,000 / 19,968 = about 50.08 frames a second

Every instruction your program runs takes up part of a line. A frame has room for about 5,000 instructions. Keep that number in mind: it's your budget for everything a game does between one picture and the next.

The lines of a frame

Lines	What's drawn
0-15	Vertical blank: nothing. The beam returns to the top.
16-50	Top border.
51-250	The 25 text rows, 8 lines each. Row *r* covers lines 51 + 8*r* to 58 + 8*r*.
251-311	Bottom border.

Reading the raster

Address	Register	Read
`$C001`	RASTER	The current line, low 8 bits.
`$C002`	RASTER_HI	Bit 0: bit 8 of the line, as it was when RASTER was last read (lines 256-311).
`$C006`	FRAME	The frame counter. Goes up by one at line 0 of each frame, wrapping from 255 to 0.

These are read-only. Writing them does nothing.

Read RASTER first. The line can change between two reads. So reading RASTER also *latches* bit 8: RASTER_HI returns the bit as it was at that moment, and keeps returning it until RASTER is read again. Read RASTER, then RASTER_HI, and the two always belong to the same line. The other way round, RASTER_HI gives the bit from your previous RASTER read, which may be a frame old.

```
        LDA RASTER      ; low 8 bits, and latch bit 8
        LDX RASTER_HI   ; bit 8 of the same line
```

When a frame's work overruns. A game loop paced by FRAME ("wait for it to change, do a frame's work, wait again") runs at 50 steps a second only while the work fits in a frame. If one step takes longer, FRAME has already moved on by the time the loop comes back to wait, so the

wait ends at once or at the next change, and the frames in between are simply skipped. Nothing warns you: the game just runs slower. To notice, keep the FRAME value you started from and compare: if it has gone up by more than one, you dropped frames. Changing the border colour at the start and end of the work (as in "Raster bars" below) shows how much of the frame it uses.

Raster bars

The border colour is recorded line by line. More precisely, each line takes the colour BORDER held when the last instruction to finish on that line finished. A program that keeps changing BORDER draws horizontal bars:

```
; Change the border colour as the raster moves down the screen.
RASTER  = $C001
BORDER  = $C020

        .org $0800
start:  SEI
loop:   LDA RASTER
        LSR A
        LSR A
        LSR A           ; a new colour every 8 lines
        STA BORDER
        JMP loop
```

The loop takes 4 + 2 + 2 + 2 + 4 + 3 = 17 cycles, so it runs about four times a line, and the bars come out steady.

Waiting for the right line

To start something at an exact line, wait for the raster to reach it:

```
; A red band in the border alongside text rows 10-12.
RASTER  = $C001
BORDER  = $C020
TOP     = 51+10*8       ; first line of row 10
BOTTOM  = 51+13*8       ; first line after row 12

        .org $0800
start:  SEI
frame:  LDA #TOP
wtop:   CMP RASTER
        BNE wtop
        LDA #2          ; red
        STA BORDER
        LDA #BOTTOM
wbot:   CMP RASTER
        BNE wbot
        LDA #0          ; black
        STA BORDER
        JMP frame
```

RASTER holds only the low 8 bits, so a line number above 255 matches twice a frame: line 300 and line 44 both read `$2C`. Check RASTER_HI too, or use lines 0-255. Better still, let the hardware wait for you: that's the raster interrupt (chapter 9).

Timing, exactly

Two facts make cycle-exact programs possible on the UNC 64. Every instruction always takes the same number of cycles, wherever it is and whatever it reads. And a line is always 64 cycles. A loop of known length started at a known line will be at a known place on the screen on every frame, forever.

9. Interrupts

An **interrupt** makes the processor drop what it's doing, run a handler, and then carry on as though nothing had happened. The UNC 64 can interrupt on a raster line, or at the start of every frame.

What happens

Before each instruction, the processor checks the interrupt request line. If a request is waiting and the I flag is clear, it:

1. pushes PC, high byte first,
2. pushes P, with B clear,
3. sets I, so the handler isn't interrupted in turn,
4. loads PC from the IRQ vector at `$FFFE`.

This takes 6 cycles. The ROM's handler at that vector does one thing, `JMP ($0314)`, which takes 5 more. So your handler starts 11 cycles after the instruction boundary where the interrupt was taken. It runs with I set, and it ends with `RTI`, which pulls P and then PC, picking up exactly where the program left off.

The processor saves only PC and P. If your handler uses A, X or Y, push them first and pull them before `RTI`.

The interrupt registers

Address	Register	Read	Write
`$C003`	RASTER_CMP	The compare line.	Set it: a raster interrupt is requested when the raster reaches this line. Lines 0-255 only.
`$C004`	IRQ_ENABLE	Enabled sources.	Bit 0: raster. Bit 1: vertical blank.
`$C005`	IRQ_STATUS	Sources that have fired.	Write a 1 to a bit to acknowledge it.

A source sets its bit in IRQ_STATUS whenever its event happens, enabled or not, so you can also poll for it. The interrupt request line is held for as long as any bit is set in both IRQ_STATUS and IRQ_ENABLE.

Always acknowledge. If your handler doesn't clear its bit, the request is still there when `RTI` returns. The processor takes it again straight away, and your program never runs another instruction.

Hooking the interrupt

`$0314-$0315` holds the address of the handler, low byte first. At start-up the ROM points it at a handler that acknowledges everything and returns. To install your own, set I so that no interrupt arrives halfway through, store the new address, then clear I:

```
; Raster split: blue border alongside the text, black above and below.
RASTER_CMP = $C003
IRQ_ENABLE = $C004
IRQ_STATUS = $C005
BORDER     = $C020
IRQVEC     = $0314

        .org $0800
start:  SEI
        LDA #<split
        STA IRQVEC
        LDA #>split
        STA IRQVEC+1
        LDA #51         ; first text line
        STA RASTER_CMP
        LDA #1          ; raster interrupts on
        STA IRQ_ENABLE
        CLI
idle:   JMP idle        ; the main program has nothing to do

split:  PHA
        LDA RASTER_CMP
        CMP #51
        BNE bottom
        LDA #6          ; blue alongside the text
        STA BORDER
        LDA #251        ; next interrupt: below the text
        STA RASTER_CMP
        JMP done
bottom: LDA #0          ; black above and below
        STA BORDER
        LDA #51
        STA RASTER_CMP
done:   LDA #1
        STA IRQ_STATUS  ; acknowledge the raster interrupt
        PLA
        RTI
```

The handler moves RASTER_CMP between two lines, so one interrupt source does two jobs per frame. The main program is free to do anything else meanwhile.

The vertical blank interrupt

Bit 1 of IRQ_ENABLE requests an interrupt at the start of every frame, as the raster returns to line 0. It's the natural heartbeat for a game: move things, play the next note, read the keyboard, all once per frame while the beam is out of sight.

BRK

`BRK` is a one-byte instruction that interrupts the processor on purpose. It works like an interrupt, but it pushes P with B set, and it happens even when I is set. It goes through the same vector, so the same handler runs. `RTI` returns to the instruction after the `BRK`.

To tell the two apart, look at the B bit in the copy of P on the stack. At the very start of the handler, before you push anything:

```
        TSX
        LDA $0101,X     ; the P that was pushed
        AND #%00010000  ; B: non-zero means BRK
```

NMI

The vector at `$FFFA` is reserved for a non-maskable interrupt. Nothing in the UNC 64 generates one.

10. The keyboard

The keyboard registers

Address	Register	Read	Write
`$C010`	KBD	The last key in bits 0-6. Bit 7 is set while it hasn't been taken.	
`$C011`	KBD_STROBE	The same as KBD, then clears bit 7.	Clears bit 7.
`$C012`	HELD	The key that is down now, or 0. See below.	

When you press a key, its code goes into KBD with bit 7 set. Reading `$C010` leaves it there. Reading `$C011` takes it: the read returns the value *with* bit 7 still set, and then bit 7 clears, so the program can wait for the next one. That makes `LDA KBD_STROBE` and `BPL` a key poll in one read: if N is set, A holds a new key (and it's already taken); if not, there was none.

Key	Code
Letters	`A-Z`, capitals only: `$41-$5A`
Digits and punctuation	Their ASCII codes, `$20-$5F`
RETURN	13
DELETE	8
STOP	3
Cursor left, right, up, down	28, 29, 30, 31

UNC BASIC looks for STOP before every statement it runs, and stops the program with `BREAK IN` and the line number. Your own machine-language programs see STOP as code 3, like any other key, and can do what they like with it. BASIC's line editor ignores the cursor keys; games are welcome to them. (Printed with CHROUT or stored on the screen, codes 28-31 are line-drawing glyphs, like every code below 32.)

The keyboard remembers one key. If a program doesn't take a key before the next one arrives, the first is lost.

Holding a key down. Hold a key and, after a pause of about half a second, it repeats, ten or more times a second, each repeat arriving in KBD as a new press. Only the last key pressed repeats: hold LEFT, then press SPACE, and the LEFT presses stop, even though LEFT is still down. For typing that's right. For a game it's awkward: a ship steered by KBD hesitates before it moves, then moves in jerks, and stops when the player fires.

HELD. `$C012` reads the code of the key that is down right now, or 0 when none is. If several are down, it's the one pressed most recently; let that one go and HELD shows the one before. HELD doesn't repeat and doesn't need taking: read it once a frame and move while it's not 0. KBD and KBD_STROBE work exactly as before, so a game can steer with HELD and still take single presses (fire, pause) from KBD.

```
        LDA HELD        ; $C012
        CMP #28         ; cursor left held?
        BNE @notleft
        DEC SHIPX       ; one step every frame it stays down
@notleft:
```

Several keys at once: the matrix

HELD shows one key. A game often wants more: turn while thrusting, and fire while doing both. The **keyboard matrix** shows every key at once, as nine rows of eight bits. A key's bit is 1 while it's down and 0 when it's up.

Address	Register	Read
`$C060-$C068`	KEYS	Rows 0-8 of the keyboard matrix. Writing does nothing.

Row	Address	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7
0	`$C060`	SPACE	!	"	#	$	%	&	'
1	`$C061`	(	)	*	+	,	-	.	/
2	`$C062`	0	1	2	3	4	5	6	7
3	`$C063`	8	9	:	;	<	=	>	?
4	`$C064`	@	A	B	C	D	E	F	G
5	`$C065`	H	I	J	K	L	M	N	O
6	`$C066`	P	Q	R	S	T	U	V	W
7	`$C067`	X	Y	Z	[	\	]	^	_
8	`$C068`	LEFT	RIGHT	UP	DOWN	RETURN	DELETE		STOP

Rows 0-7 are the codes `$20-$5F` in order: the key with code *c* is bit *c* AND 7 of row (*c* - `$20`) / 8. Row 8 holds the cursor keys in bits 0-3 (code - 28), RETURN, DELETE, and STOP in bit 7, where `BIT $C068` puts it in N: `BMI` and the player wants out. Bit 6 of row 8 is always 0.

```
; Read held keys from the matrix: A, D and SPACE, all at once.
KEYS    = $C060             ; row 0 of the keyboard matrix
SCREEN  = $0400

        .org $0800
start:  LDX #' '
        LDA KEYS+4          ; row 4: @ A B C D E F G
        AND #%00000010      ; bit 1: A
        BEQ @a
        LDX #'A'
@a:     STX SCREEN
        LDX #' '
        LDA KEYS+4
        AND #%00010000      ; bit 4: D
        BEQ @d
        LDX #'D'
@d:     STX SCREEN+1
        LDX #' '
        LDA KEYS            ; row 0: SPACE ! " # $ % & '
        AND #%00000001      ; bit 0: SPACE
        BEQ @f
        LDX #'*'
@f:     STX SCREEN+2
        JMP start
```

Hold A and SPACE together and both show; let go of A and press D, and the `D` and the `*` show. The loop takes about 50 cycles, so the corner catches up with the keys hundreds of times a frame.

The matrix goes by key code, like KBD, so a shifted key shows as the character it types: shift and 1 together light `!`, not `1`. It doesn't repeat and needs no taking: read the rows a game needs once a frame. KBD, KBD_STROBE and HELD work exactly as before, and taking a key from KBD_STROBE doesn't let it go in the matrix.

Reading keys yourself

```
; Show each key's character in the top-left corner of the screen.
KBD     = $C010
STROBE  = $C011

        .org $0800
start:  LDA KBD
        BPL start       ; bit 7 clear: no key yet
        BIT STROBE      ; take the key
        AND #$7F        ; the code without bit 7
        STA $0400
        JMP start
```

`BPL` tests bit 7 for free, because `LDA` copies bit 7 into N. `BIT STROBE` reads the strobe without disturbing A.

Reading keys with the ROM

`GETKEY` at `$FF06` waits for a key and returns its code in A:

```
; Echo every key twice.
CHROUT  = $FF03
GETKEY  = $FF06

        .org $0800
start:  JSR GETKEY
        JSR CHROUT
        JSR CHROUT
        JMP start
```

11. The beeper

The UNC 64's speaker has two positions, in and out. Move it in and out at a steady rate and you hear a tone. There are two ways to move it: let the **timer** do it, which costs the processor nothing, or flip it yourself, from a loop with every cycle counted.

The registers

Address	Register	Read	Write
`$C030`	BEEPER		Any value flips SPEAKER bit 1.
`$C031`	SPEAKER	Bits 0 and 1 as written. Bit 5: the timer's output. The rest 0.	Bit 0: the timer's **gate** (1: it runs). Bit 1: speaker **data** (1: the timer reaches the speaker).
`$C032`	TIMER_LO	As written.	The timer's divisor, low byte.
`$C033`	TIMER_HI	As written.	The divisor, high byte. Writing it gives the timer TIMER_HI × 256 + TIMER_LO.
`$C034`	NOISE	A pseudo-random byte. See "Noise".	

At switch-on they're all 0: the timer stopped, the speaker in, silence.

How the speaker is wired

The **timer** counts the processor's clock, 1,000,000 cycles a second, and puts out a square wave: high for the first half of every *divisor* cycles, low for the second half. The speaker is out while this is true:

```
speaker out = DATA AND (GATE ? timer output : 1)
```

GATE is SPEAKER bit 0 and DATA is SPEAKER bit 1. As a table:

SPEAKER	The speaker
`%00`	In: silence. As at switch-on.
`%01`	In. The timer runs, but nothing reaches the speaker.
`%11`	Follows the timer: a tone.
`%10`	Out. With the gate off, the timer's output is held high, so DATA alone sets the speaker.

BEEPER is a shortcut: any write to it flips DATA. With the gate off, as it is unless a program turns it on, each flip moves the speaker in or out: playing the speaker by hand, flip by flip. With a tone playing, a flip turns DATA off and the tone stops, until the next flip.

Tones with the timer

The timer plays a tone of

```
pitch = 1,000,000 / divisor  Hz
```

so the divisor for a pitch is 1,000,000 / pitch. Write the low byte to TIMER_LO, then the high byte to TIMER_HI: TIMER_HI is the one that counts, so the timer never runs on half a divisor. Then store `%11` in SPEAKER. The tone plays until you change it, and the processor gets on with something else. A tone from the timer costs no cycles at all.

Note	Divisor	Note	Divisor
C (middle C)	3822	F#	2703
C#	3608	G	2551
D	3405	G#	2408
D#	3214	A (440 Hz)	2273
E	3034	A#	2145
F	2863	B	2025

For the octave above, halve the divisor; for the one below, double it. A divisor of 0 means 65,536, the lowest the timer goes: 15.3 Hz, more a rattle than a note. Divisors below 50 make pitches above 20,000 Hz, which nobody hears, and 1 holds the output high: no wave at all. With an odd divisor the two halves can't be equal, so the output is high for (divisor + 1) / 2 cycles and low for the rest.

```
; Play a tune on the timer: the first two lines of Frere Jacques.
FRAME   = $C006
SPEAKER = $C031
TIMER_LO = $C032
TIMER_HI = $C033
LEFT    = $10           ; frames left to wait

        .org $0800
start:  LDX #0
note:   LDA tune+2,X    ; the note's length in frames; 0 ends the tune
        BEQ done
        STA LEFT
        LDA tune,X
        STA TIMER_LO
        LDA tune+1,X
        STA TIMER_HI    ; the new divisor takes over at the next half-wave
        LDA #%11        ; gate on, speaker data on: the timer plays
        STA SPEAKER
        JSR wait
        LDA #%01        ; speaker data off for a moment: the notes don't run together
        STA SPEAKER
        LDA #2
        STA LEFT
        JSR wait
        INX
        INX
        INX
        JMP note
done:   LDA #0          ; gate off, data off: silence
        STA SPEAKER
        HLT

wait:   LDA FRAME       ; wait LEFT frames
@tick:  CMP FRAME
        BEQ @tick
        DEC LEFT
        BNE wait
        RTS

C4      = 3822          ; 1,000,000 / 261.6 Hz
D4      = 3405
E4      = 3034
F4      = 2863
G4      = 2551
tune:   .word C4
        .byte 18
        .word D4
```

```
        .byte 18
        .word E4
        .byte 18
        .word C4
        .byte 18
        .word C4
        .byte 18
        .word D4
        .byte 18
        .word E4
        .byte 18
        .word C4
        .byte 18
        .word E4
        .byte 18
        .word F4
        .byte 18
        .word G4
        .byte 38
        .word E4
        .byte 18
        .word F4
        .byte 18
        .word G4
        .byte 38
        .byte 0, 0, 0
```

Each note sets the divisor, plays for its length, then turns DATA off for two frames so that two notes the same don't run together. The timer itself never stops until `done`.

Changing note. A new divisor doesn't cut the wave short. The timer finishes the half-wave it's in and starts the new one from there, so one note follows another without a click. To start a wave afresh at a known moment, turn the gate off and on again: a wave always starts with its high half, on the cycle the gate goes on.

Music in the background. Because the timer plays by itself, a program touches it only when a note changes. Put the tune's next step in the vertical blank interrupt (chapter 9), once a frame, and the main program never knows there's music playing.

SPEAKER bit *5* reads the timer's output, and reads 1 while the gate is off. A program can watch it to time something: with a divisor of 1,000 it changes every 500 cycles.

The hard way: flipping the speaker yourself

The timer plays square waves, one pitch at a time. Everything else (a click, a chirp that slides, two notes trilled) you play by hand: with the gate off, any write to `$C030` flips the speaker.

A tone is all timing. One full wave needs two flips. If a loop flips the speaker every n cycles, the pitch is 1,000,000 / (2n) Hz.

```
; A steady tone near 440 Hz, the A above middle C.
BEEPER  = $C030
N       = 226

        .org $0800
start:  SEI             ; an interrupt would put the timing out
tone:   STA BEEPER      ; 4 cycles
        LDX #N          ; 2
wait:   DEX             ; 2
        BNE wait        ; 3 when taken, 2 the last time
        JMP tone        ; 3
```

One trip round `tone` costs 4 + 2 + *5N* - 1 + 3 = *5N* + 8 cycles. With *N* = 226 that's 1,138 cycles per flip, or 2,276 per wave: 1,000,000 / 2,276 = 439.4 Hz. Every note played by hand is worked out like this, and while it plays the processor does nothing else. That's what the timer is for.

Noise

Address	Register	Read
`$C034`	NOISE	A pseudo-random byte, from the cycle it's read on. Writing does nothing.

Read NOISE and you get a byte worked out from the exact cycle the reading instruction starts on. Two reads one cycle apart give unrelated bytes. It costs what any load costs, 4 cycles for `LDA NOISE`, and needs no setting up.

It isn't truly random, on purpose. Run the same program, with the same keys pressed on the same cycles, and it reads the same bytes: that's what lets a recorded game be replayed and checked. What makes it unpredictable is the player. The cycle a key goes down is never the same twice, so a read of NOISE just after a key press is as good as a dice throw.

Use it to **seed** RND (chapter 12), so that every game is different:

```
        JSR GETKEY      ; "PRESS ANY KEY"
        LDA NOISE
        STA SEED        ; $020B
        LDA NOISE
        ORA #1          ; never 0: the generator would stay at 0
        STA SEED+1
```

Or use it as a **fast random source**: a byte in 4 cycles, where RND takes 53. Every read in a loop is unrelated to the one before, however regular the loop.

Noise you can hear. Flip the speaker when a NOISE bit says so, and you get a hiss: an explosion, surf, a rocket.

```
hiss:   LDA NOISE
        BPL @skip       ; flip half the time
        STA BEEPER
@skip:  DEX
        BNE hiss
```

Which cycle? As far as the UNC 64's registers are concerned, a read or a write happens on the first cycle of the instruction that does it. The timer, SPEAKER bit *5* and NOISE all go by that cycle, so a program does exactly the same thing every time it runs.

12. The system ROM

The top 8K of memory, `$E000-$FFFF`, is the **system ROM**: the program that runs when you switch on. It's read-only; writing to it does nothing. It's written in UNC assembly, and its source is the best long example of UNC-1 programming you'll find.

What it does at start-up

1. Sets the stack pointer to `$FF`.
2. Points the interrupt hook at `$0314` to its own handler.
3. Sets the border and background to black.
4. Checks for the 64K expansion by writing `$55` to `$4000` and reading it back. Sets RAMTOP at `$0203` to `$40` (16K) or `$C0` (64K).
5. Empties BASIC's program and variables, and seeds RND.
6. Clears the screen and prints the start-up message, with the bytes free for BASIC.
7. Clears I, and hands over to UNC BASIC, which prints `READY.` and waits for a line (chapter 14).

It all takes about 23,000 cycles: a little over a frame.

Calling the ROM

Call the ROM's routines with `JSR` through the **jump table** at `$FF00`. The addresses in the table will stay the same in every future ROM, even when the routines behind them move.

Address	Routine	In	Out	Changes
`$FF00`	CLS		Screen cleared, cursor at the top left	A, X
`$FF03`	CHROUT	A = character	Character printed	nothing
`$FF06`	GETKEY		A = key code. Waits for a key.	A
`$FF09`	PRSTR	X = low byte, Y = high byte of a string ending in 0	String printed. Up to 255 characters.	A, Y
`$FF0C`	PRHEX	A = byte	Two hex digits printed	A
`$FF0F`	PRDEC	X = low byte, Y = high byte	The number printed in decimal, 0 to 65535, no leading zeros	A, X
`$FF12`	RND		A = a pseudo-random byte	A
`$FF15`	LOAD	X = low byte, Y = high byte of a name ending in 0; empty for the first program	Doesn't return: the program from the tape starts, or BASIC prints the error (chapter 13)	everything
`$FF18`	NUMSTR	X = low, Y = middle, A = high byte of a number from 0 to 16,777,215	Eight ASCII digits, leading zeros and all, in NUMBUF (`$03D0-$03D7`). Prints nothing. A = where the first digit that isn't a leading zero is, 0-7 (7 for 0)	A, X, Y
`$FF1B`	PRNUM	A = width: 0, or 1-8	The last NUMSTR's number printed: with no leading zeros for width 0, or in exactly *width* digits, padded with zeros	A, X

Address	Routine	In	Out	Changes
$FF1E	RNDSEED	X = low byte, Y = high byte	SEED set; 0 becomes the start-up seed	A
$FF21	READY		Doesn't return: BASIC's READY., with the BASIC program and its variables as they were	everything

The routines don't promise to keep the flags.

Printing doesn't draw a cursor. CHROUT, PRSTR, PRHEX, PRDEC and PRNUM put characters on the screen and move CURX and CURY on, and that's all: the screen shows exactly what you printed. GETKEY doesn't draw one either. The blinking cursor you see at READY. belongs to BASIC's line editor ("The line editor's cursor", below).

PRDEC prints a 16-bit number as an unsigned decimal: a score, a count, an address. It uses BASIC's workspace in the fast page ($E4-$E5 and $F0-$F1), so keep your own variables out of $D0-$FE when you call it, as with the other routines.

RND steps the same generator BASIC's RND uses, a 16-bit xorshift held in SEED ($020B-$020C), and returns the new high byte of SEED. The ROM seeds it at start-up, so a program that wants a different game each time should give it a new seed first, with RNDSEED: two NOISE reads when the player presses a key to start (chapter 11), say. Calling RND from machine language moves BASIC's sequence on too.

READY is the way back to BASIC from a machine-language program that has nowhere to RTS to: one that LOAD started, say. It empties the stack, turns interrupts on (BASIC runs with them on), and goes to BASIC's READY., on a new line if the cursor isn't at the start of one. Nothing in memory is cleared: the BASIC program, its variables and the screen stay as they were, and RUNFLG goes to 0, direct mode. Called from a routine that a running BASIC program SYSed, it stops the program there, as END would.

RNDSEED sets it: X and Y become SEED, low byte in X. The xorshift can't work from 0 (it would give 0 for ever), so a seed of 0 becomes the start-up seed, $ACE1.

```
        JSR GETKEY      ; $FF06: "PRESS A KEY TO START"
        LDX NOISE       ; $C034: depends on when it was pressed
        LDY NOISE
        JSR RNDSEED     ; $FF1E
        JSR RND         ; $FF12
        AND #31         ; 0-31
```

Numbers of any width: NUMSTR and PRNUM

PRDEC stops at 65,535, and prints no leading zeros. A score wants more of both, so the work is split in two. **NUMSTR** turns a 24-bit number, up to 16,777,215, into eight ASCII digits in **NUMBUF** ($03D0-$03D7), the highest first and with leading zeros: 420 becomes 00000420. It doesn't print, so the digits are there for you to use as you like: copy them to the screen yourself, compare them, print them twice. It returns in A the position in NUMBUF of the first digit that isn't a leading zero: 5 for 420, 0 for 16,777,215, and 7 for 0, so the last 0 is always counted.

PRNUM prints the last NUMSTR's digits at the cursor, through CHROUT. A is the width. 0 prints the number as PRDEC would, with no leading zeros. 1 to 8 print exactly that many digits, the number padded on the left with zeros. A number with more digits than the width is printed whole, never cut short: 123456 at width 3 prints `123456`. A width over 8 counts as 8.

```
; Show the score at the top right, six digits wide.
NUMSTR  = $FF18
PRNUM   = $FF1B
CURX    = $0200
CURY    = $0201

        .org $0800
start:  LDA #34         ; column 34 of row 0: six digits reach the right edge
        STA CURX
        LDA #0
        STA CURY
        LDX score       ; low byte
        LDY score+1     ; middle byte
        LDA score+2     ; high byte
        JSR NUMSTR
        LDA #6          ; six digits wide
        JSR PRNUM       ; prints 000420
        HLT

score:  .byte <420, >420, 420 >> 16
```

NUMSTR uses BASIC's `$E4-$E6` and PRNUM its `$F0`, so the rule for `$D0-$FE` holds for them too. NUMBUF keeps the digits until the next NUMSTR.

Keeping score fast

NUMSTR is the easy way to show a number, not the fast way. A six-digit score typically takes about 1,250 cycles to convert and 900 more to print: a tenth of a frame, every time it changes. That isn't NUMSTR being careless. The UNC-1 has no decimal mode, by design (chapter 3), so turning binary into decimal always means dividing by ten, and on the UNC-1 that's repeated subtraction: hundreds of cycles for even a small number.

So a score that changes every frame should never be binary in the first place. Keep it as a row of decimal digits, one byte each, 0 to 9, the highest first, and add points the way you would on paper: add to one digit, and if it reaches 10, take 10 off and carry one into the digit to its left. Only the digits that change get written to the screen.

```
; Keep score as a row of digits: 420, 580, then 7 points 143 times.
SCREEN  = $0400
SCORE   = $10           ; six digits, 0-9 each, the highest first: $10-$15
ROW     = SCREEN+12*40+17 ; where the score shows: row 12, columns 17-22

        .org $0800
start:  LDX #5          ; 000000
clear:  LDA #0
        STA SCORE,X
        LDA #'0'
        STA ROW,X
        DEX
        BPL clear
        LDA #4          ; 420 is 4 hundreds...
        LDX #3
        JSR add
        LDA #2          ; ...and 2 tens
        LDX #4
        JSR add
        LDA #5          ; 580: 5 hundreds...
        LDX #3
        JSR add
        LDA #8          ; ...and 8 tens. 001000
        LDX #4
        JSR add
        LDY #143        ; 7 points, 143 times: 002001
more:   LDA #7
        LDX #5
        JSR add
        DEY
        BNE more
        HLT

; ADD: add A (0-9) to digit X (5 the ones, 4 the tens... 0 the hundred thousands),
; carrying into the digits to its left, and show each digit that changes.
; Past 999999 the score rolls over to 000000. Changes A and X.
add:    CLC
        ADC SCORE,X
        CMP #10
        BCC last        ; under 10: no carry, and this is the last digit to change
        SBC #10         ; CMP left the carry set
        STA SCORE,X
        ORA #'0'        ; the digit's character
        STA ROW,X
        LDA #1          ; carry one...
        DEX             ; ...into the digit to the left
        BPL add
```

```
        RTS
last:   STA SCORE,X
        ORA #'0'
        STA ROW,X
        RTS
```

A call that changes one digit takes 34 cycles, `JSR` and `RTS` included, and each carry adds 30. Even 999,999 + 1, which carries through all six digits and rolls over, costs under 200: a hundredth of a frame. Points worth more than 9 are one call a digit, like 420 above, or a loop.

When the score has to be a number too (to compare with the high score, say), compare the digits from the highest down: the first pair that differs decides.

CHROUT in detail

CHROUT prints at the cursor and moves the cursor on. Some codes do something instead of printing:

Code	Does
13	New line.
12	Clear the screen, cursor to the top left.
8	Back one column, unless already at the left edge. Doesn't rub out.
anything else	Stores the code at the cursor, even block graphics and reverse video.

After column 39 the cursor moves to the next line. Below row 24 the screen scrolls up one row. Nothing is drawn at the cursor and nothing there is rubbed out: a new line leaves the rest of the old line as it was.

The line editor's cursor

The cursor is part of BASIC's line editor, not of printing: it's there only while the machine is waiting for you to type. At `READY.` and at an `INPUT`, the line editor shows a cursor (code `$A0`, a reversed space, which the video chip blinks: chapter 7) where the next key will go, keeping the character that was there in CURSAV (`$020F`). It moves the cursor along as you type, and when it stops waiting it takes it away and puts the character back: on RETURN, and on STOP. So by the time `RUN` starts a program, or `LOAD` starts the tape, the cursor has gone, and a running program has a screen with no cursor on it unless it's waiting at `INPUT`.

A program started from the Machine Room starts on BASIC's screen just as BASIC left it, `READY.` and all (chapter 5), but without the cursor: the loader takes it away, putting back what was under it. Clear the screen first if you want it empty.

What it costs

Routine	Cycles, including the `JSR`
CLS	about 6,400
CHROUT	about 125, on any row
CHROUT, when it scrolls	about 19,000: nearly a whole frame

Routine	Cycles, including the `JSR`
GETKEY, with a key waiting	27
PRHEX	about 285
PRDEC	about 320 for one digit, about 1,700 for 65535
RND	53
NUMSTR	about 530 for 0, 680 for 420, 1,000 for a 16-bit number, 1,250 for a six-digit one; at most about 2,520, for 9,999,999
PRNUM	about 140 a digit printed, and 25 for each leading zero it skips
RNDSEED	32

Printing is slow. A game that redraws the screen with CHROUT every frame won't keep up. Write to screen memory directly.

Memory the ROM uses

Address	Name	Holds
`$F7-$F8`	SRC	Pointer, used by scrolling.
`$F9-$FA`	DST	Pointer, used by scrolling.
`$FB-$FC`	PTR	Pointer to the cursor's row, as the last CHROUT left it.
`$FD-$FE`	STRP	Pointer, used by PRSTR.
`$0200`	CURX	Cursor column, 0-39.
`$0201`	CURY	Cursor row, 0-24.
`$0202`	TMPA	Scratch.
`$0203`	RAMTOP	High byte of the first address past memory: `$40` or `$C0`.
`$020F`	CURSAV	The character under the line editor's cursor.
`$0314-$0315`	IRQVEC	The interrupt handler's address.
`$03D0-$03D7`	NUMBUF	NUMSTR's eight digits.

UNC BASIC also uses `$D0-$F6` in the fast page and most of `$0204-$03C8`; Appendix D lists every byte.

Move the cursor by storing to CURX and CURY. The next CHROUT prints there.

Keep your programs and data out of `$0200-$03FF`, and out of `$D0-$FE` when you call the ROM or return to BASIC. Future ROMs will need more of both.

The vectors

Address	Vector	Points to
`$FFFA`	NMI	Reserved.
`$FFFC`	Reset	The start-up code.
`$FFFE`	IRQ and BRK	`JMP ($0314)`.

13. Tape

Programs come on cassettes. The tape deck plugs into the back of the computer, and the ROM's `LOAD` reads a program off a tape into memory and starts it. A cassette holds as many programs as fit on it, one after another, each with a name.

Loading a program

Put the cassette in the deck, then type `LOAD` and press RETURN:

```
LOAD
PRESS PLAY ON TAPE
SEARCHING
FOUND SNAKE
LOADING
```

1. `LOAD` switches the deck's motor on. If PLAY isn't down, it asks for it with `PRESS PLAY ON TAPE`, and waits.
2. Press PLAY on the deck. The tape moves, and `LOAD` prints `SEARCHING` and reads the tape until it finds the start of a program.
3. Each program starts with a header that holds its name. `LOAD` prints `FOUND` and the name.
4. `LOADING`: the program goes into memory, at the address its header gives. The border shows the bytes going past, a stripe of colour for each one: the **loading stripes**.
5. At the end, `LOAD` checks the program's checksum, switches the motor off and puts the border back as it was. Then it starts the program.

`LOAD` alone takes the first program it finds. `LOAD "SNAKE"` takes the first one called `SNAKE`: it prints `SEARCHING FOR SNAKE`, and `FOUND` for every program it passes on the way. The name has to match exactly, up to 16 characters; `LOAD ""` is the same as `LOAD`.

A program that lands on the BASIC program **replaces the resident program**: if any of its bytes fall between `$0800` and the BASIC program's end marker (PRGEND, Appendix D), `LOAD` forgets the BASIC program (as `NEW` does) before it loads. A program that loads clear of it, wholly above the end marker or wholly below `$0800`, leaves the BASIC program and its variables as they were. A machine-language program starts at its entry address in the start state of chapter 5: SP = `$FF`, I set, A, X and Y 0. The screen still shows the `LOAD` messages, so a program clears it itself if it wants a clean screen.

A machine-language program doesn't come back to BASIC by itself. A game never needs to. A tape that only brings something for BASIC to use, numbers or a routine to `SYS`, loads clear of the BASIC program and ends with a jump to READY (`$FF21`, chapter 12): you get `READY.` back, and your program is still there to `RUN`.

```
; A data tape: ten numbers for BASIC at 12288, then READY.
PRSTR   = $FF09
READY   = $FF21

        .org $3000      ; 12288, well above a BASIC program
data:   .byte 3, 1, 4, 1, 5, 9, 2, 6, 5, 3
start:  LDX #<msg       ; the entry: say where the numbers are
        LDY #>msg
        JSR PRSTR
        JMP READY       ; back to BASIC, program and variables kept
msg:    .asciiz "DATA AT 12288-12297"
```

Put it on a tape (`unc-asm data.s --tape data.tap`), type a program that reads the numbers with `PEEK`, then `LOAD`. The numbers arrive, `DATA AT 12288-12297` and `READY.` follow, and `LIST` shows your program, untouched.

Loading takes time. The tape gives about 600 bytes a second, so a 1.6K game takes under three seconds, and one of 6K about ten. The header and the checksum add 29 bytes, a twentieth of a second.

Stops LOAD	Message
The STOP key	`BREAK`
STOP on the deck, or the end of the tape (the deck lets PLAY up by itself)	`BREAK`
The checksum is wrong, a byte was missed, or the program isn't one this ROM can run	`?LOAD ERROR`
The program wouldn't fit between `$0400` and the top of memory	`?OUT OF MEMORY`

All of them switch the motor off and put the border back. `LOAD "WRONG"` on a tape without a `WRONG` reads to the end of the tape, prints `FOUND` for everything on it, and ends with `BREAK` when the deck stops. The tape is then at its end: take it out and put it in again to start from the beginning.

While `LOAD` prints, it switches the motor off, so that no byte goes past while the screen is busy.

From machine language, call `LOAD` through the jump table (chapter 12): X and Y hold the address of the name, a string ending in 0 (an empty one takes the first program). It never returns: it starts the program, or prints its message and goes to BASIC's `READY.`.

The deck

The deck has two keys. **PLAY** stays down while the tape plays. **STOP/EJECT** lets PLAY up; pressed again, it ejects the tape. A cassette goes in at its start.

The tape moves only while PLAY is down **and** the computer runs the motor. That's how `LOAD` can stop the tape and start it again without your help. At the end of the tape, the deck lets PLAY up by itself.

The deck isn't part of the computer. Reset doesn't eject the tape or move it; it only stops the motor.

The tape registers

Address	Name	Read	Write
$C040	TAPE	Bit 0: PLAY is down. Bit 1: a tape is in. Bit 2: the motor is on. Bit 6: overrun. Bit 7: a byte is ready	Bit 2: the motor, 1 on, 0 off. The other bits do nothing
$C041	TAPEDATA	The byte off the tape. Reading it takes it: bits 6 and 7 of TAPE clear	Nothing

While the tape moves, a new byte arrives every **1,664 cycles**: 12 a frame, about 601 a second. It goes into TAPEDATA and sets bit 7 of TAPE. Take it before the next one comes. A byte that arrives while bit 7 is still set replaces the one waiting, which is lost, and sets bit 6, the **overrun** flag, until you next read TAPEDATA.

`BIT TAPE` tests both at once: N is "a byte is ready" and V is "a byte was lost". 1,664 cycles is plenty of time for a loop that takes each byte as it comes, but not for a `CHROUT` that scrolls the screen (chapter 12): switch the motor off first, as `LOAD` does.

```
; Read the tape yourself: the next 256 bytes to $3000
TAPE    = $C040
TDATA   = $C041
BORDER  = $C020
        .org $0800
start:  LDA #4
        STA TAPE        ; motor on
        LDX #0
wait:   BIT TAPE        ; N: a byte is ready
        BPL wait
        LDA TDATA       ; take it
        STA $3000,X
        STA BORDER      ; your own loading stripes
        INX
        BNE wait
        LDA #0
        STA TAPE        ; motor off
        HLT
```

Run it with a tape in the deck and PLAY down. Without PLAY, it waits for ever: press it, and the program carries on.

The tape format

A program on tape is a header, the program's bytes and a checksum. Format version 1:

Byte	Size	Holds
0	4	`UNCT`: the mark `LOAD` searches for
4	1	Format version: 1
5	1	Type: 1 = machine language, started at its entry
6	16	Name: capitals, padded with spaces

Byte	Size	Holds
22	2	Load address, low byte first
24	2	Length, low byte first
26	2	Entry address, low byte first
28	length	The program
28 + length	1	Checksum: every byte from the version to the end of the program, combined with `EOR`

The checksum makes all the bytes after `UNCT`, checksum included, `EOR` together to 0. That's how `LOAD` checks them: it `EOR`s each byte into a running total as it reads it, and the total has to end at 0.

A tape is these programs one after another, so a tape's first four bytes are always `UNCT` and the fifth is the format version. A version-1 ROM gives `?LOAD ERROR` for a version it doesn't know rather than guess.

14. UNC BASIC

UNC BASIC is built into the system ROM. It's ready the moment you switch on, and it's small on purpose: whole numbers only, 26 variables, and a handful of statements you can learn in an evening. What it lacks in size it makes up for in reach: `PEEK`, `POKE` and `SYS` get at every part of the machine.

Direct mode and programs

A line typed **without** a number runs straight away. That's **direct mode**, and it's how you try things out. A line typed **with** a number is stored as part of the **program** in memory, and runs when you type `RUN`.

Put several statements on one line with a colon between them:

```
FOR I=1 TO 3:PRINT I:NEXT
```

`READY.` means BASIC has finished and is waiting for you.

The line editor

You type	BASIC
`10 PRINT "HI"`	Stores line 10, replacing any old line 10.
`10`	Deletes line 10.
`NEW`	Forgets the whole program and every variable.
`LIST`	Shows the program, in line-number order.

Line numbers run from 0 to 65535. Lines can be up to 120 characters: three rows of the screen. While you type, DELETE rubs out the last character, but not past the left edge of the screen.

The blinking cursor shows where the next character goes. It's there only while BASIC is waiting for you to type, at `READY.` or at an `INPUT`, and it goes as soon as you press RETURN or STOP, putting back whatever was under it. A running program's screen has no cursor unless the program is waiting at `INPUT` (chapter 12).

BASIC stores each keyword as a single byte, a **token**, so programs take less memory and run faster. `LIST` spells them out again. Spaces are kept as you typed them. `?` is short for `PRINT`, and lists as `PRINT`.

Commands

Command	Does
`RUN`	Clears the variables and runs the program from its first line.
`LIST`	Lists the whole program.
`LIST 100`	Lists line 100.
`LIST 100-200`	Lists lines 100 to 200. `LIST -200` and `LIST 100-` work too.
`NEW`	Forgets the program and the variables.

Command	Does
`HELP`	Shows the keywords on the screen.
`LOAD`	Loads the first program on the tape, and starts it (chapter 13).
`LOAD "`*name*`"`	The same, for the first program called *name*.

Commands work inside programs too: a program can `LIST` itself.

Statements

Statement	Does
`PRINT` *items*	Prints strings in quotes and the values of expressions. `;` between items prints them side by side; `,` moves on to the next tenth column. A `;` or `,` at the very end means no new line. `PRINT` alone prints an empty line.
`INPUT` *v*	Prints `?` and waits for you to type a number and press RETURN. Puts it in variable *v*. Anything that isn't a whole number gets `?REDO` and another try.
`INPUT "`*prompt*`";`*v*	The same, with your own prompt first.
`LET` *v*`=`*expr*	Puts the value of *expr* into variable *v*. `LET` is optional: `A=A+1` is the same.
`IF` *expr* `THEN` *line*	Goes to *line* if *expr* is true (not zero).
`IF` *expr* `THEN` *statement*	Runs the rest of the line if *expr* is true. If it's false, the rest of the line, colons and all, is skipped.
`GOTO` *line*	Carries on from *line*. In direct mode, starts the program there.
`GOSUB` *line*	Like `GOTO`, but remembers where it came from. Up to 16 deep.
`RETURN`	Goes back to just after the last `GOSUB`.
`FOR` *v*`=`*a* `TO` *b*	Starts a loop: *v* is set to *a*.
`FOR` *v*`=`*a* `TO` *b* `STEP` *c*	The same, counting by *c* (which can be negative).
`NEXT` *v*	Adds the step to *v*. If *v* hasn't passed *b*, goes back to just after the `FOR`. `NEXT` alone means the innermost loop. Up to 8 loops inside each other.
`POKE` *address*`,`*value*	Puts *value* (0-255) into memory at *address*.
`REM` *anything*	A remark. BASIC ignores the rest of the line.
`END`	Ends the program.
`CLS`	Clears the screen.
`SYS` *address*	Calls a machine-language routine (see below).
`GR`	Graphics: turns on the bitmap with four text rows below it, clears both, and puts the cursor on the first text row.
`TEXT`	Back to the text screen, cleared.
`PLOT` *x*`,`*y*	Draws the pixel at (*x*, *y*).
`LINE` *x1*`,`*y1*`,`*x2*`,`*y2*	Draws a straight line, both ends included.
`CIRCLE` *x*`,`*y*`,`*r*	Draws a circle of radius *r* around (*x*, *y*).
`INK` *n*	How the drawing commands draw: 1 lights pixels (the start), 0 rubs them out, 2 flips them.

A `FOR` loop always runs at least once, and the loop ends when the variable passes the limit: `FOR I=1 TO 10` leaves `I` at 11. Starting a new `FOR` on a variable that's already looping starts that loop over.

`GOTO` and `GOSUB` take any expression, so `GOTO 100+N*10` works.

Numbers

UNC BASIC knows whole numbers from -32768 to 32767, the range of two bytes. There are no fractions: `7/2` is 3, and `-7/2` is -3. Arithmetic that goes past the end wraps round without a word: `32767+1` is -32768.

You can type any number from 0 to 65535. Above 32767, it wraps to a negative number: `49184` is stored as -16352. That sounds like a problem and isn't, because an address is just two bytes either way: `POKE 49184,2` and `POKE -16352,2` reach the same place. Only `PRINT` shows the difference.

A number starting with `$` is in hexadecimal: `POKE $C020,2` is the same as `POKE 49184,2`.

Variables

The 26 variables are the letters `A` to `Z`. Each holds one number, and starts at 0. `RUN` and `NEW` set them all back to 0. Changing the program or pressing STOP doesn't, so after a STOP you can `PRINT` a variable to see how far the program got.

Expressions

From first to last:

Operators	Meaning
`( )`	Grouping.
`-`	Negative (in front of a value).
`* /`	Multiply, divide.
`+ -`	Add, subtract.
`= <> < > <= >=`	Compare. True is -1, false is 0.
`AND`	Both: bit by bit.
`OR`	Either: bit by bit.

Because true is -1 (every bit set), `AND` and `OR` work on comparisons the way you'd hope: `IF X>0 AND X<10 THEN 200`. `IF` counts any value that isn't 0 as true.

Functions

Function	Gives
`PEEK(`*address*`)`	The byte at *address*, 0-255.
`POINT(`*x*`,`*y*`)`	1 if the pixel at (*x*, *y*) is lit, 0 if not, or off the screen.
`RND(`*n*`)`	A random number from 0 to *n*-1. 0 if *n* is 0 or less.
`ABS(`*n*`)`	*n* without its minus sign.

RND gives the same series every time you switch on. That's a feature: a game that uses it plays the same way twice, so its best scores can be compared.

STOP

The STOP key stops a running program. BASIC prints BREAK IN and the line it was on, and goes back to READY.. STOP also gets you out of INPUT and LIST, and throws away a line you're halfway through typing.

Errors

When BASIC can't go on, it prints a message starting with ?, adds IN and the line number if a program was running, and goes back to READY..

Message	Means
?SYNTAX ERROR	BASIC can't read the line: a misspelled keyword, a missing bracket, a stray character.
?UNDEFINED LINE	GOTO or GOSUB to a line that doesn't exist.
?NEXT WITHOUT FOR	A NEXT with no loop running, or for a variable no loop uses.
?RETURN WITHOUT GOSUB	A RETURN with nowhere to return to.
?DIVISION BY ZERO	Dividing by 0.
?OUT OF MEMORY	The program fills memory, or GOSUBs go more than 16 deep, or FORs more than 8, or GR finds the program in the bitmap's way, or a program on tape wouldn't fit.
?LOAD ERROR	The program on the tape is damaged, or isn't one this ROM can run (chapter 13).

Graphics

GR turns the screen over to the bitmap (chapter 7): 320 pixels across, numbered 0 to 319 from the left, and 200 down, 0 to 199 from the top. You see the top 168 lines; below them, four rows of text keep READY. and your typing in view. TEXT goes back to the text screen.

```
10 GR
20 CIRCLE 250,40,22
30 LINE 250,10,250,2:LINE 250,70,250,78
40 LINE 220,40,212,40:LINE 280,40,288,40
50 LINE 229,19,223,13:LINE 271,19,277,13
60 LINE 229,61,223,67:LINE 271,61,277,67
70 LINE 60,100,160,100:LINE 160,100,160,160
80 LINE 50,100,110,55:LINE 110,55,170,100
90 CIRCLE 110,80,6
100 LINE 95,160,95,125:LINE 95,125,115,125:LINE 115,125,115,160
110 LINE 125,110,150,110:LINE 150,110,150,135
120 LINE 150,135,125,135:LINE 125,135,125,110
130 LINE 60,100,60,160:LINE 0,160,319,160
```

A sun over a house: a circle and eight rays, two walls, a roof, a round attic window, a door, a window, and the ground.

Anything off the edges is simply not drawn, so a circle can hang half off the screen and a line can start anywhere. `POINT` tells you whether a pixel is lit, which is how a game finds out that something has hit a wall.

`INK 2` flips pixels instead of lighting them, so drawing the same thing twice leaves the screen exactly as it was. That's how to move a shape over a background: draw it, wait, draw it again in the same place to take it away, then draw it in the new one. Every command draws each pixel once, so even circles and lines come off cleanly.

Memory. The bitmap goes in the top 8K of memory: `$2000-$3F3F` on a UNC 16, `$A000-$BF3F` on a UNC 64. While it's on, your program has to fit below it, which on a UNC 16 leaves 6K. `GR` gives `? OUT OF MEMORY` if the program is already too long, and so does typing a line that would grow it into the bitmap. After `TEXT` the memory is BASIC's again (the picture stays there until something writes over it).

Speed. The drawing itself is machine code in the ROM. `PLOT` takes about 0.6 ms, most of it BASIC reading the two numbers. A line 50 pixels long takes about 15 ms, and one corner to corner across the whole screen about 80 ms. A circle of radius 10 takes about 15 ms, and one of radius 90 about 0.1 s. `GR` itself takes about 0.1 s, most of it clearing 8,000 bytes.

Machine language: SYS

`SYS` *address* calls the machine-language routine at *address*, as `JSR` would. When the routine ends with `RTS`, BASIC carries on with the next statement. This is where BASIC hands over to the rest of this book.

You can even build a routine with `POKE`s. These six bytes are `LDA #2`, `STA $C020`, `RTS`:

```
10 POKE 12288,40:POKE 12289,2
20 POKE 12290,51:POKE 12291,32:POKE 12292,192
30 POKE 12293,3
40 SYS 12288
```

The border turns red again, this time by machine language. Appendix C gives every instruction's bytes.

A routine you call with `SYS` can use A, X and Y freely. It must leave the stack as it found it, and leave `$D0-$FE` and `$0200-$03FF` alone.

Where everything is

The program is stored from `$0800` upwards, line after line. Each line is a length byte (the whole line, in bytes), the line number (two bytes, low first), the text with its tokens, and a 0. A length of 0 marks the end of the program. Everything above that, up to the top of memory, is free; the start-up message counts it.

Variables, the `FOR` and `GOSUB` stacks and the line you're typing live in `$0200-$03FF`, and BASIC's pointers in `$D0-$F6`. Appendix D lists them. Two of them are worth knowing: `$0204` is 0 in direct mode and not 0 while a program runs, and `$0205-$0206` holds the number of the line being run.

Machine-language programs also load at `$0800` unless they say otherwise, on top of any BASIC program. To keep both, put your machine language higher up: `$3000` (12288) on a UNC 16 leaves 10K for BASIC below it. `LOAD` keeps the BASIC program when a tape loads clear of it (chapter 13).

Tokens

Token	Keyword	Token	Keyword	Token	Keyword
`$80`	PRINT	`$8B`	END	`$96`	CIRCLE
`$81`	LET	`$8C`	CLS	`$97`	INK
`$82`	INPUT	`$8D`	SYS	`$98`	THEN
`$83`	IF	`$8E`	RUN	`$99`	TO
`$84`	GOTO	`$8F`	LIST	`$9A`	STEP
`$85`	GOSUB	`$90`	NEW	`$9B`	PEEK
`$86`	RETURN	`$91`	HELP	`$9C`	RND
`$87`	FOR	`$92`	GR	`$9D`	ABS
`$88`	NEXT	`$93`	TEXT	`$9E`	AND
`$89`	POKE	`$94`	PLOT	`$9F`	OR
`$8A`	REM	`$95`	LINE	`$A0`	POINT
				`$A1`	LOAD

`LOAD` came last, so the tokens before it kept their numbers.

Speed

BASIC reads each line afresh every time it runs it, and works out every number in it from its digits every time. That makes it slow. An empty `FOR` loop takes about half a millisecond a pass, so 1,000 passes take half a second. A line that does real work, like `POKE 1024+X,32:X=X+1:POKE 1024+X,81`, takes about 6 milliseconds: a third of a frame. A game that moves a dozen things around the screen in BASIC will manage a few frames a second.

Some things help: fewer, longer lines; variables instead of long numbers (`B=49184` once, then `POKE B,2`); `GOTO`s to lines near the start of the program, which BASIC finds sooner. When that isn't enough, it's time for machine language.

15. The video board

In preparation. The video board is an upgrade card, one of the expansions that make a UNC 16 a UNC 64. It adds new video modes: hardware sprites, smooth scrolling and background layers. Everything in chapter 7, UNSCII and custom characters included, works without it.

Appendix A. Screen codes and colours

Screen codes

UNSCII, the character ROM. Rows give the high hex digit, columns the low one. The graphics glyphs are shown with the nearest typewriter symbol; the real glyphs are 8 × 8 pixels.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	empty	▘	▝	▀	▖	▌	▞	▛	▗	▚	▐	▜	▄	▙	▟	█
1	─	│	┌	┐	└	┘	├	┤	┬	┴	┼	╱	╲	╳	▒	░
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	▁	▂	▃	▄	▅	▆	▇	▏	▎	▍	▌	▋	▊	▉	▔	▕
7	♠	♥	♦	♣	•	○	←	↑	→	↓	◢	◣	◤	◥	☺	♪

- `$00-$0F`: bit 0 lights the top-left quarter of the cell, bit 1 the top-right, bit 2 the bottom-left and bit 3 the bottom-right.
- `$10-$1A`: the lines run through the middle of the cell, two pixels thick, and reach its edges, so neighbouring cells join into boxes.
- `$60-$66` fill the bottom 1 to 7 eighths of the cell; `$67-$6D` the left 1 to 7 eighths. `$6E` is the top eighth and `$6F` the right eighth.
- `$80-$FF`: the glyph of the code minus `$80`, in reverse video. `$C1` is a reversed A. `$A0`, the reversed space, blinks about once a second: it's the cursor. VMODE bit 3 holds it steady (chapter 7). In 256-glyph mode (chapter 7) there's no reverse video and no blinking.

The glyphs themselves are in `rom/unscii.txt`, drawn as rows of `#` and `.`.

Colours

Number	Colour	Hex (colour monitor)
0	Black	`#000000`
1	White	`#e8e4da`
2	Red	`#9a3b2e`
3	Cyan	`#5fb3a8`
4	Purple	`#7a3f8a`
5	Green	`#4f9a4f`
6	Blue	`#2c3d9a`
7	Yellow	`#e2c84a`
8	Orange	`#a8632a`
9	Brown	`#5a3e1a`
10	Light red	`#d0705f`
11	Dark grey	`#3a3a3a`
12	Grey	`#6b6b6b`

Number	Colour	Hex (colour monitor)
13	Light green	`#9fe08a`
14	Light blue	`#6f8fd8`
15	Light grey	`#a9a9a9`

On the monochrome monitor each colour shows as green, as bright as the colour is light.

Appendix B. Memory map and I/O registers

Memory map

Range	Size	What
`$0000-$00FF`	256	**Fast page.** `$D0-$F6` UNC BASIC; `$F7-$FE` the screen routines' pointers.
`$0100-$01FF`	256	**Stack.**
`$0200-$03FF`	512	**System variables** and BASIC's workspace (Appendix D). `$0314-$0315` is the IRQ vector. The rest is reserved for future ROMs.
`$0400-$07E7`	1,000	**Screen memory.** 40 × 25 screen codes, row by row.
`$07E8-$07FF`	24	Free.
`$0800-$3FFF`	14K	**Your programs.** BASIC programs start at `$0800`; machine-language programs load there too unless they say otherwise. While the bitmap is on, it takes 8,000 bytes from BITMAP × 256: at reset, `$2000-$3F3F`.
`$4000-$BFFF`	32K	**Expansion memory**, with the 64K expansion only. Without it, reads give `$FF` and writes vanish.
`$C000-$CFFF`	4K	**Input/output.**
`$D000-$D3FF`	1K	**UNSCII character ROM.** Read-only.
`$D400-$DFFF`	3K	Unused: reads `$FF`.
`$E000-$FFFF`	8K	**System ROM.** Read-only.

I/O registers

Address	Name	Read	Write	Chapter
`$C000`	VMODE	Video mode	Bit 0: 256-glyph character set. Bit 1: bitmap. Bit 2: mixed (bitmap above, text rows 21-24 below). Bit 3: steady cursor (`$A0` doesn't blink). All 0 at reset	7
`$C001`	RASTER	Current raster line, bits 0-7; latches bit 8 for RASTER_HI		8
`$C002`	RASTER_HI	Bit 0: raster line bit 8, as latched by the last read of RASTER		8
`$C003`	RASTER_CMP	Compare line	Compare line (0-255)	9
`$C004`	IRQ_ENABLE	Enabled sources	Bit 0 raster, bit 1 vertical blank	9
`$C005`	IRQ_STATUS	Sources that have fired	1 bits acknowledge	9
`$C006`	FRAME	Frame counter		8
`$C010`	KBD	Last key; bit 7 set while not taken		10
`$C011`	KBD_STROBE	As KBD (bit 7 included), then clears bit 7	Clears bit 7	10

Address	Name	Read	Write	Chapter
$C012	HELD	The key down now (the latest pressed), or 0. Rev A draft		10
$C020	BORDER	Border colour	Border colour, 0-15	7
$C021	BACKGROUND	Background colour	Background colour, 0-15	7
$C022	CHARSET	Character set page	Character set page; $D0 at reset	7
$C023	BITMAP	Bitmap page	Bitmap page; $20 at reset	7
$C030	BEEPER		Any value flips SPEAKER bit 1	11
$C031	SPEAKER	Bit 0 gate, bit 1 data, bit 5 the timer's output. Rev A draft	Bit 0: timer gate. Bit 1: speaker data; BEEPER flips it	11
$C032	TIMER_LO	As written. Rev A draft	Timer divisor, low byte	11
$C033	TIMER_HI	As written. Rev A draft	Timer divisor, high byte: loads the divisor (0 is 65,536)	11
$C034	NOISE	Pseudo-random byte from the cycle count. Rev A draft		11
$C040	TAPE	Bit 0: PLAY down. Bit 1: tape in. Bit 2: motor on. Bit 6: overrun. Bit 7: byte ready	Bit 2: motor on	13
$C041	TAPEDATA	The byte off the tape; reading takes it (clears TAPE bits 6 and 7)		13
$C050	SHIFT_IN		Pushes a byte into the shifter's window. Rev A draft	7
$C051	SHIFT_AMOUNT	Shift, 0-7. Rev A draft	Shift (bits 0-2)	7
$C052	SHIFT_OUT	The window shifted right by SHIFT_AMOUNT, low byte. Rev A draft		7
$C053	SHIFT_FLIP		Pushes a byte with its bits reversed. Rev A draft	7
$C060-$C068	KEYS	Keyboard matrix, rows 0-8: 1 while a key is down. Rev A draft		10
$C3F0-$C3FF		Reserved	Reserved	

Every other address in $C000-$CFFF reads $FF and ignores writes. Blank cells: reads give $FF, writes are ignored.

Timing

Clock	1 MHz
Cycles per raster line	64
Raster lines per frame	312
Cycles per frame	19,968

Frames per second	about 50.08
Vertical blank	lines 0-15
Text area	lines 51-250
Tape	a byte every 1,664 cycles (12 a frame) while PLAY is down and the motor on

Appendix C. Instruction set

The UNC-1 processor has *55* operations in 136 opcodes. Cycle costs are fixed: a taken branch adds one cycle, and nothing else varies. There is no decimal mode and no page-crossing penalty.

Opcode map

Rows are the high nibble, columns the low nibble. Blank cells are unassigned: the processor jams on them.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	NOP	HLT	RTS	RTI	JSR abs	JMP abs	JMP (abs)	BEQ	BNE	BCC	BCS	BMI	BPL	BVC	BVS
1x	TAX	TAY	TXA	TYA	TXS	TSX	INX	INY	DEX	DEY	PHA	PLA	PHP	PLP	CLC	SEC
2x	CLI	SEI	CLV						LDA #	LDA fp	LDA fp,X	LDA abs	LDA abs,X	LDA abs,Y	LDA (fp),Y	
3x		STA fp	STA fp,X	STA abs	STA abs,X	STA abs,Y	STA (fp),Y		ADC #	ADC fp	ADC fp,X	ADC abs	ADC abs,X	ADC abs,Y	ADC (fp),Y	
4x	SBC #	SBC fp	SBC fp,X	SBC abs	SBC abs,X	SBC abs,Y	SBC (fp),Y		AND #	AND fp	AND fp,X	AND abs	AND abs,X	AND abs,Y	AND (fp),Y	
5x	ORA #	ORA fp	ORA fp,X	ORA abs	ORA abs,X	ORA abs,Y	ORA (fp),Y		EOR #	EOR fp	EOR fp,X	EOR abs	EOR abs,X	EOR abs,Y	EOR (fp),Y	
6x	CMP #	CMP fp	CMP fp,X	CMP abs	CMP abs,X	CMP abs,Y	CMP (fp),Y		LDX #	LDX fp		LDX abs		LDX abs,Y		LDX fp,Y
7x	LDY #	LDY fp	LDY fp,X	LDY abs	LDY abs,X					STX fp		STX abs				STX fp,Y
8x		STY fp	STY fp,X	STY abs					CPX #	CPX fp		CPX abs				
9x	CPY #	CPY fp		CPY abs						INC fp	INC fp,X	INC abs				
Ax		DEC fp	DEC fp,X	DEC abs						ASL fp	ASL fp,X	ASL abs				ASL A
Bx		LSR fp	LSR fp,X	LSR abs				LSR A		ROL fp	ROL fp,X	ROL abs				ROL A
Cx		ROR fp	ROR fp,X	ROR abs				ROR A		BIT fp		BIT abs				
Dx																
Ex																
Fx																

Instructions

Syntax	Opcode	Bytes	Cycles	Operation	Flags
`BRK`	$00	1	6	Break: push PC and status (B set), jump through $FFFE	I
`NOP`	$01	1	2	No operation	-
`HLT`	$02	1	2	Halt the processor until reset	-
`RTS`	$03	1	6	Return from subroutine	-
`RTI`	$04	1	6	Return from interrupt: pull status, then PC	all
`JSR $nnnn`	$05	3	6	Jump to subroutine: push the return address	-
`JMP $nnnn`	$06	3	3	Jump	-
`JMP ($nnnn)`	$07	3	5	Jump	-

Syntax	Opcode	Bytes	Cycles	Operation	Flags
`BEQ label`	$08	2	2 (+1 if taken)	Branch if equal (Z set)	-
`BNE label`	$09	2	2 (+1 if taken)	Branch if not equal (Z clear)	-
`BCC label`	$0A	2	2 (+1 if taken)	Branch if carry clear	-
`BCS label`	$0B	2	2 (+1 if taken)	Branch if carry set	-
`BMI label`	$0C	2	2 (+1 if taken)	Branch if minus (N set)	-
`BPL label`	$0D	2	2 (+1 if taken)	Branch if plus (N clear)	-
`BVC label`	$0E	2	2 (+1 if taken)	Branch if overflow clear	-
`BVS label`	$0F	2	2 (+1 if taken)	Branch if overflow set	-
`TAX`	$10	1	2	A to X	N Z
`TAY`	$11	1	2	A to Y	N Z
`TXA`	$12	1	2	X to A	N Z
`TYA`	$13	1	2	Y to A	N Z
`TXS`	$14	1	2	X to stack pointer	-
`TSX`	$15	1	2	Stack pointer to X	N Z
`INX`	$16	1	2	X + 1	N Z
`INY`	$17	1	2	Y + 1	N Z
`DEX`	$18	1	2	X - 1	N Z
`DEY`	$19	1	2	Y - 1	N Z
`PHA`	$1A	1	3	Push A	-
`PLA`	$1B	1	4	Pull A	N Z
`PHP`	$1C	1	3	Push status (with B set)	-
`PLP`	$1D	1	4	Pull status	all
`CLC`	$1E	1	2	Clear carry	C
`SEC`	$1F	1	2	Set carry	C
`CLI`	$20	1	2	Clear interrupt disable	I
`SEI`	$21	1	2	Set interrupt disable	I
`CLV`	$22	1	2	Clear overflow	V
`LDA #$nn`	$28	2	2	Load A	N Z
`LDA $nn`	$29	2	3	Load A	N Z
`LDA $nn,X`	$2A	2	4	Load A	N Z
`LDA $nnnn`	$2B	3	4	Load A	N Z
`LDA $nnnn,X`	$2C	3	5	Load A	N Z

Syntax	Opcode	Bytes	Cycles	Operation	Flags
`LDA $nnnn,Y`	$2D	3	5	Load A	N Z
`LDA ($nn),Y`	$2E	2	6	Load A	N Z
`STA $nn`	$31	2	3	Store A	-
`STA $nn,X`	$32	2	4	Store A	-
`STA $nnnn`	$33	3	4	Store A	-
`STA $nnnn,X`	$34	3	5	Store A	-
`STA $nnnn,Y`	$35	3	5	Store A	-
`STA ($nn),Y`	$36	2	6	Store A	-
`ADC #$nn`	$38	2	2	A + M + C	N V Z C
`ADC $nn`	$39	2	3	A + M + C	N V Z C
`ADC $nn,X`	$3A	2	4	A + M + C	N V Z C
`ADC $nnnn`	$3B	3	4	A + M + C	N V Z C
`ADC $nnnn,X`	$3C	3	5	A + M + C	N V Z C
`ADC $nnnn,Y`	$3D	3	5	A + M + C	N V Z C
`ADC ($nn),Y`	$3E	2	6	A + M + C	N V Z C
`SBC #$nn`	$40	2	2	A - M - (1 - C)	N V Z C
`SBC $nn`	$41	2	3	A - M - (1 - C)	N V Z C
`SBC $nn,X`	$42	2	4	A - M - (1 - C)	N V Z C
`SBC $nnnn`	$43	3	4	A - M - (1 - C)	N V Z C
`SBC $nnnn,X`	$44	3	5	A - M - (1 - C)	N V Z C
`SBC $nnnn,Y`	$45	3	5	A - M - (1 - C)	N V Z C
`SBC ($nn),Y`	$46	2	6	A - M - (1 - C)	N V Z C
`AND #$nn`	$48	2	2	A AND M	N Z
`AND $nn`	$49	2	3	A AND M	N Z
`AND $nn,X`	$4A	2	4	A AND M	N Z
`AND $nnnn`	$4B	3	4	A AND M	N Z
`AND $nnnn,X`	$4C	3	5	A AND M	N Z
`AND $nnnn,Y`	$4D	3	5	A AND M	N Z
`AND ($nn),Y`	$4E	2	6	A AND M	N Z

Syntax	Opcode	Bytes	Cycles	Operation	Flags
ORA #$nn	$50	2	2	A OR M	N Z
ORA $nn	$51	2	3	A OR M	N Z
ORA $nn,X	$52	2	4	A OR M	N Z
ORA $nnnn	$53	3	4	A OR M	N Z
ORA $nnnn,X	$54	3	5	A OR M	N Z
ORA $nnnn,Y	$55	3	5	A OR M	N Z
ORA ($nn),Y	$56	2	6	A OR M	N Z
EOR #$nn	$58	2	2	A XOR M	N Z
EOR $nn	$59	2	3	A XOR M	N Z
EOR $nn,X	$5A	2	4	A XOR M	N Z
EOR $nnnn	$5B	3	4	A XOR M	N Z
EOR $nnnn,X	$5C	3	5	A XOR M	N Z
EOR $nnnn,Y	$5D	3	5	A XOR M	N Z
EOR ($nn),Y	$5E	2	6	A XOR M	N Z
CMP #$nn	$60	2	2	Compare A with M	N Z C
CMP $nn	$61	2	3	Compare A with M	N Z C
CMP $nn,X	$62	2	4	Compare A with M	N Z C
CMP $nnnn	$63	3	4	Compare A with M	N Z C
CMP $nnnn,X	$64	3	5	Compare A with M	N Z C
CMP $nnnn,Y	$65	3	5	Compare A with M	N Z C
CMP ($nn),Y	$66	2	6	Compare A with M	N Z C
LDX #$nn	$68	2	2	Load X	N Z
LDX $nn	$69	2	3	Load X	N Z
LDX $nnnn	$6B	3	4	Load X	N Z
LDX $nnnn,Y	$6D	3	5	Load X	N Z
LDX $nn,Y	$6F	2	4	Load X	N Z
LDY #$nn	$70	2	2	Load Y	N Z
LDY $nn	$71	2	3	Load Y	N Z
LDY $nn,X	$72	2	4	Load Y	N Z
LDY $nnnn	$73	3	4	Load Y	N Z
LDY $nnnn,X	$74	3	5	Load Y	N Z
STX $nn	$79	2	3	Store X	-
STX $nnnn	$7B	3	4	Store X	-
STX $nn,Y	$7F	2	4	Store X	-
STY $nn	$81	2	3	Store Y	-
STY $nn,X	$82	2	4	Store Y	-
STY $nnnn	$83	3	4	Store Y	-

Syntax	Opcode	Bytes	Cycles	Operation	Flags
CPX #$nn	$88	2	2	Compare X with M	N Z C
CPX $nn	$89	2	3	Compare X with M	N Z C
CPX $nnnn	$8B	3	4	Compare X with M	N Z C
CPY #$nn	$90	2	2	Compare Y with M	N Z C
CPY $nn	$91	2	3	Compare Y with M	N Z C
CPY $nnnn	$93	3	4	Compare Y with M	N Z C
INC $nn	$99	2	5	M + 1	N Z
INC $nn,X	$9A	2	6	M + 1	N Z
INC $nnnn	$9B	3	6	M + 1	N Z
DEC $nn	$A1	2	5	M - 1	N Z
DEC $nn,X	$A2	2	6	M - 1	N Z
DEC $nnnn	$A3	3	6	M - 1	N Z
ASL $nn	$A9	2	5	Shift left; bit 7 to C	N Z C
ASL $nn,X	$AA	2	6	Shift left; bit 7 to C	N Z C
ASL $nnnn	$AB	3	6	Shift left; bit 7 to C	N Z C
ASL A	$AF	1	2	Shift left; bit 7 to C	N Z C
LSR $nn	$B1	2	5	Shift right; bit 0 to C	N Z C
LSR $nn,X	$B2	2	6	Shift right; bit 0 to C	N Z C
LSR $nnnn	$B3	3	6	Shift right; bit 0 to C	N Z C
LSR A	$B7	1	2	Shift right; bit 0 to C	N Z C
ROL $nn	$B9	2	5	Rotate left through C	N Z C
ROL $nn,X	$BA	2	6	Rotate left through C	N Z C
ROL $nnnn	$BB	3	6	Rotate left through C	N Z C
ROL A	$BF	1	2	Rotate left through C	N Z C
ROR $nn	$C1	2	5	Rotate right through C	N Z C
ROR $nn,X	$C2	2	6	Rotate right through C	N Z C
ROR $nnnn	$C3	3	6	Rotate right through C	N Z C
ROR A	$C7	1	2	Rotate right through C	N Z C
BIT $nn	$C9	2	3	Test bits: Z = A AND M is zero, N = M bit 7, V = M bit 6	N V Z
BIT $nnnn	$CB	3	4	Test bits: Z = A AND M is zero, N = M bit 7, V = M bit 6	N V Z

Appendix D. ROM reference

Jump table

Address	Routine	In	Out	Changes
`$FF00`	CLS		Screen cleared, cursor at the top left	A, X
`$FF03`	CHROUT	A = character; 13 new line, 12 clear, 8 back	Character printed; no cursor drawn	nothing
`$FF06`	GETKEY		A = key code, bit 7 clear	A
`$FF09`	PRSTR	X/Y = address of a zero-terminated string	String printed	A, Y
`$FF0C`	PRHEX	A = byte	Two hex digits printed	A
`$FF0F`	PRDEC	X/Y = a 16-bit number, low byte in X	Printed in unsigned decimal	A, X; `$E4-$E5`, `$F0-$F1`
`$FF12`	RND		A = pseudo-random byte; SEED advanced	A
`$FF15`	LOAD	X/Y = address of a zero-terminated name; empty: the first program	Doesn't return: starts the program, or BASIC prints `BREAK` or the error	everything; `$D0-$D3`, `$02E4-$030F`
`$FF18`	NUMSTR	X/Y/A = a 24-bit number: low byte in X, middle in Y, high in A	Eight ASCII digits with leading zeros in NUMBUF; A = index of the first that isn't a leading zero (7 for 0). Prints nothing	A, X, Y; `$E4-$E6`
`$FF1B`	PRNUM	A = width: 0 none, 1-8 digits (over 8: 8)	NUMBUF printed: no leading zeros (0), or padded with zeros to the width; never cut short	A, X; `$F0`
`$FF1E`	RNDSEED	X/Y = a 16-bit seed, low byte in X	SEED = X/Y; 0 becomes `$ACE1`, the start-up seed	A
`$FF21`	READY		Doesn't return: SP = `$FF`, I clear, RUNFLG 0, then BASIC's `READY.` on a new line. Memory, the BASIC program and its variables untouched	everything

The next free slot is `$FF24`.

Vectors

Address	Vector	ROM Rev A value
`$FFFA-$FFFB`	NMI	Reserved
`$FFFC-$FFFD`	Reset	Start-up code
`$FFFE-$FFFF`	IRQ and BRK	`JMP ($0314)`

System variables

Address	Name	Holds
$D0-$DB		UNC BASIC graphics: the pixel, its byte's address, circle centre, error term. LOAD uses $D0-$D3
$DC-$F6		UNC BASIC: LIST range, text and line pointers, expression accumulator, scratch
$F7-$F8	SRC	ROM pointer
$F9-$FA	DST	ROM pointer
$FB-$FC	PTR	ROM pointer: the cursor's row
$FD-$FE	STRP	ROM pointer: PRSTR's string
$0200	CURX	Cursor column
$0201	CURY	Cursor row
$0202	TMPA	ROM scratch
$0203	RAMTOP	$40 (16K) or $C0 (64K)
$0204	RUNFLG	0 in direct mode; not 0 while a BASIC program runs
$0205-$0206	CURLIN	Number of the BASIC line being run
$0207	FORSP	Bytes in use on the FOR stack
$0208	GOSP	Bytes in use on the GOSUB stack
$0209-$020A	PRGEND	Address of the program's end marker
$020B-$020C	SEED	RND state, shared by BASIC's RND and the jump table's RND; RNDSEED sets it
$020D	INKM	The current INK
$020E	GFXS	Graphics scratch
$020F	CURSAV	The character under the line editor's cursor, put back when it stops waiting
$0210-$025F	GOSTK	GOSUB stack: 16 frames of 5 bytes
$0260-$0293	VARS	Variables A-Z, 2 bytes each, low byte first
$0294-$02E3	FORSTK	FOR stack: 8 frames of 10 bytes
$02E4-$02F3	TNAME	LOAD: the name asked for, padded with spaces
$02F4	TNAMED	LOAD: not 0 if a name was given
$02F5-$030C	THDR	LOAD: the header as read, from the version byte to the entry address
$030D	TCHK	LOAD: the bytes so far, EORed
$030E	TOVR	LOAD: $40 if a byte was missed
$030F	TBORD	LOAD: the border colour before LOAD
$0314-$0315	IRQVEC	Interrupt handler address
$0320-$0398	IBUF	The line being typed, 120 characters and a 0
$03A0-$03C8	INBUF	INPUT's line, 40 characters and a 0
$03D0-$03D7	NUMBUF	NUMSTR's eight ASCII digits, the highest first, with leading zeros

Appendix E. Assembler messages

Message	Meaning
`unknown instruction 'LDQ'`	Not a UNC-1 mnemonic. Check the spelling against Appendix C.
`unknown directive '.foo'`	Not one of the directives in chapter 5.
`STA has no immediate mode`	The instruction exists, but not in the mode you wrote. Appendix C lists every mode each instruction has.
`INC has no absolute,X mode`	As above, for an indexed address.
`JMP needs an operand`	The instruction needs an address or value.
`LDA can't operate on A`	Only the shifts and rotates take `A` as an operand.
`undefined symbol 'x'`	A name that was never defined as a label or constant.
`undefined local label '@x' (after 'loop')`	No `@x` between the label `loop` and the next ordinary label. Local labels don't reach across ordinary ones.
`undefined local label '@x' (there's no global label before it)`	A local label used before the program's first ordinary label, and not defined there.
`'x' is defined twice`	Two labels or constants with the same name.
`'@x' is defined twice (after 'loop')`	The same local label twice under one ordinary label.
`'A' can't be a name: as an operand, A means the accumulator`	A label or constant called `A` or `a`. Call it something else.
`a constant can't have a label: put 'FOO = ...' on its own line, without 'x:'`	A label and a constant on one line, like `x: FOO = 1`.
`address $0900 is written twice (first by line 7)`	Two parts of the program store bytes at the same address. Reported on the line that writes it second.
`unknown escape '\n'`	In a string or character, a backslash goes only before `"`, `'` or `\`.
`bad shift count 70`	A shift by less than 0 or more than 63 bits.
`bad align count 0`	`.align` needs a number from 1 to 65536.
`bad reserve count -1`	`.res` needs a count from 0 to 65536.
`bad rept count -1`	`.rept` needs a count from 0 to 65536.
`the .rept count must be known here: undefined symbol 'n'`	`.rept` counts are worked out as the assembler reaches them, so define the count above.
`'X' is a register: name the counter something else`	A `.rept` counter or `.irp` name called `A`, `X` or `Y` would replace the register names in the copies.
`bad counter name '1x'`	A `.rept` counter or `.irp` name must be a name.
`'.rept' without '.endr'`	A `.rept` block that never ends. `'.irp' without '.endr'` is the same for `.irp`.
`'.endr' without '.rept' or '.irp'`	An `.endr` with no `.rept` or `.irp` above it.
`an .irp value is empty`	Two commas in a row, or one at the end, in an `.irp` list.
`'\i' pastes a .rept or .irp name, but there's none called 'i' here`	A `\name` outside a block that names it, or a pasted name that runs on: `R\iX` looks for a name `iX`.

Message	Meaning
`can't paste '1+2' into a name`	Only a value that's a plain word or number can be pasted with \.
`unknown function 'foo'`	A name followed by (that isn't one of the functions in chapter 5.
`sin takes 2 values: sin(angle, size)`	A function given too many or too few values.
`sqrt of a negative number (-4)`	`sqrt` needs 0 or more.
`missing ')'`	A function's (without its).
`round brackets are for indirect addressing and functions: group with [ ]`	A (that isn't at the start of an operand or after a function name.
`the program expands to more than 1000000 lines`	`.rept` and `.include` made the program enormous; probably a count is wrong.
`'.include' needs a file name in quotes`	Write `.include "file.s"`.
`can't read 'file.s'`	The included file isn't there (it's looked for next to the program).
`includes nest too deep at 'file.s' (does it include itself?)`	More than 16 includes inside each other.
`includes aren't available here`	The Machine Room has no files to include.
`assertion failed: expr`	An `.assert` without a message found its expression 0. With a message, the message is shown instead.
`#256 doesn't fit in a byte`	An immediate value outside -128 to 255.
`300 doesn't fit in a byte`	A `.byte` value outside -128 to 255.
`70000 doesn't fit in a word`	A `.word` value outside -32768 to 65535.
`$1234 is outside the fast page`	A fast-page or `(fp),Y` operand above `$FF`.
`$10000 is outside memory`	An address above `$FFFF`.
`address $10000 is outside memory`	The program runs past `$FFFF`.
`branch out of range (200 bytes)`	Branches reach 128 bytes back and 127 forward. Branch round a `JMP` instead.
`bad number '...'`	A number that can't be read: a `$` or `%` with no digits after it, or one too big (over 64 bits, or a decimal over 9223372036854775807).
`bad character literal`	A character in quotes must be exactly one character: `'A'`.
`bad fill count -1`	`.fill` needs a count from 0 to 65536.
`division by zero`	An expression divides by zero.
`missing value`	An expression ends where a value should be.
`missing ']'`	A [without its].
`unexpected '!' in expression`	A character that has no meaning in an expression.
`unexpected text in '...'`	An expression with something left over, like `1 2`.

Warnings

A warning doesn't stop the program assembling. It's printed by the command-line assembler and shown as a note in the listing.

Message	Meaning
`'==' binds tighter than '&': add [ ] to say which you mean`	A `&`, `|` or `^` with a comparison on one side only: `F & $FF == 0` is `F & [$FF == 0]`. Write `[F & $FF] == 0`.
`'<' takes the low byte of the whole comparison: add [ ] to say which you mean`	`<BOARD == 0` is the low byte of `[BOARD == 0]`. Write `[<BOARD] == 0`. The same for `>` and the high byte.

Appendix F. For 6502 programmers

If you know the 6502, you know the UNC-1. Mnemonics, registers, flags, the stack and the addressing modes will all feel familiar. These are the differences:

	6502	UNC-1
Opcodes		Different numbering. 6502 binaries don't run.
Zero page	Zero page	Called the **fast page**. Same idea.
Decimal mode	`SED`, `CLD`, BCD arithmetic	None. Bit 3 of P does nothing.
Page-crossing penalty	+1 cycle on some indexed reads and branches	None. Costs are fixed.
Absolute,X / Absolute,Y read	4 (+1)	5, always
(Indirect),Y read	5 (+1)	6, always
Indexed indirect `(zp,X)`	Yes	None
`JSR`	Pushes the return address minus one	Pushes the real return address
`JMP ($xxFF)`	Wraps within the page	Reads the right bytes
`BRK`	2 bytes, 7 cycles	1 byte, 6 cycles; `RTI` returns to the next byte
Interrupt entry	7 cycles	6 cycles
`HLT`	None	Stops until reset
Undefined opcodes	Do various things	Jam the processor
`BIT #imm`, `STZ`, `PHX` etc.	65C02 only	None
`INC abs,X`, `DEC abs,X`, shifts `abs,X`	Yes	None
`STY abs,X`, `STX abs,Y`	Yes	None: only the fast-page forms, `STY fp,X` and `STX fp,Y`
`LDA zp,Y` and others with no zero page,Y mode	Assemblers vary	Assembled as absolute,Y without a word: three bytes, absolute,Y's cost (chapter 5)

Fixed costs make the UNC-1 slightly slower than a 6502 on indexed reads, and much easier to count.

Appendix G. The Machine Room

The Machine Room is the UNC 64 in a browser, with an assembler and a debugger beside it. It runs the real emulator core compiled to WebAssembly, so it runs exactly the machine this book describes.

The screen

The monitor shows the machine at 50 frames a second. Each refresh of your screen runs the frames the machine is owed, at most two, so the picture keeps time without bursts; on a 60 Hz screen, one refresh in six shows no new frame. Click it, and your keys go to the machine: the cursor keys as codes 28-31, Ctrl+C as STOP, and every key's release to HELD (chapter 10). Under it:

- **Colour monitor**: switch between the green monochrome monitor and a colour one.
- **RAM**: 16K (UNC 16) or 64K (with the expansion). Changing it switches the machine off and on.
- **Mute**: silence the speaker. The page remembers it.

The speaker plays through your computer's sound (chapter 11). Browsers allow sound only after you click or press a key, so it starts at your first one.

The assembler

Type a program, or pick one from **Sample**, then **Assemble & run** (Ctrl+Enter). The machine is switched off and on, the program is loaded, and it runs from `start` in the start state of chapter 5. If there are errors, they're listed under the editor. Click one to jump to its line. Tab types a tab. There are no files here, so `.include` can't be used.

Samples: Hello, Raster bars, Count in hex, Character set, Custom characters, Keyboard, Tune, The reserved range.

The debugger

Control	Key	Does
Run / Pause	F8	Run the machine, or stop it where it is.
Step	F7	Run one instruction.
Step over	F6	Run one instruction, but run a `JSR`'s whole subroutine as one step.
Power cycle		Switch the machine off and on. The program is gone.

The status shows RUNNING, PAUSED, HALTED, `JAMMED ON $nn` (the opcode it jammed on) or `BREAK AT $nnnn`. Below it are the registers, the flags (lit when set), the cycle count and the current raster line.

Disassembly shows the instructions around PC, with labels and the source line each one came from. Click a line to set or clear a breakpoint there. Running stops *before* that instruction.

Memory shows 256 bytes from any address. Type an address, or jump to the fast page, the screen, your program, I/O or the ROM. Bytes that changed since the display last refreshed are highlighted.

Stack shows the bytes on the stack, from SP up.

The command-line assembler

The same assembler runs from a terminal:

```
cargo run -p unc-asm -- prog.s -o prog.bin [--range 0800-0FFF] [--fill FF] [--sym prog.sym] [--lst prog.lst]
cargo run -p unc-asm -- prog.s --tape prog.tap [--name NAME]
```

It writes a flat binary of the range: by default from the lowest to the highest byte the program emits, with gaps filled with `$FF`. Reserved bytes (`.res`) are gaps too. It prints errors as `prog.s:12: message`, or `file.s:3: message` for a line in an included file. `.include "file.s"` finds the file in the same folder as `prog.s`.

- `--sym FILE` writes the symbol table, one `NAME = $XXXX` line per label and constant, in address order. Local labels appear as `global@local`.
- `--lst FILE` writes a listing: each source line with its address, the bytes it made (up to three on the line, the rest on lines below) and its line number. `.rept` copies are listed one by one. Where an operand that fits the fast page was assembled as absolute, the line ends with a note: `; ** LDA has no fast page,Y mode: absolute,Y`, or `; ** defined later: absolute`.

```
0800  2D 10 00      2  start: LDA PTR,Y  ; ** LDA has no fast page,Y mode: absolute,Y
0803  6F 10         3   LDX PTR,Y
```

- `--tape FILE` writes a tape image (chapter 13): one machine-language program from the lowest to the highest byte the program emits (gaps are 0), entry at `start`, named NAME or after the file. The image is the bytes on the tape, so `cat a.tap b.tap > both.tap` makes a tape with both programs.

`-o` can be left out when you only want the symbols, the listing or the tape.

Appendix H. For hosts: watching the machine, and save states

Watching the machine

`Machine::observe()` in Rust, `observe()` on the game's machine node, says what the processor has done since the last call, then starts the counts again from 0:

Field	What it counts
`reads`, `writes`	The processor's reads and writes of each I/O address `$C000-$C0FF`: 256 counts each. `INC $C020` is one of each.
`reserved_read`	Whether anything read `$C3F0-$C3FF` (Appendix B's reserved range).
`frames`	Frames that ended.
`screen_writes`	Writes to screen memory, `$0400-$07E7`.
`bitmap_writes`	Writes to the 8,000 bytes from BITMAP × 256, wherever BITMAP points at the time of the write, and whether or not the bitmap is showing.

Only the processor's own reads and writes count, whoever's program it's running: BASIC's `PEEK` and `POKE` count, because the ROM does them with `LDA` and `STA`. The host's own `peek`, `poke` and `load` don't, so a debugger or a check that looks at memory doesn't change what it measures.

The counts are the host's bookkeeping. No program can read them, and nothing the machine does depends on them, so a program runs the same, cycle for cycle, whether anything watches it or not. Keeping them costs one comparison on each memory access, too little to measure. A reset doesn't clear them; `Machine::new` starts them at 0. A few calls a second is plenty: a count only wraps after 2^{32} events.

In the game, `observe()` returns a dictionary: `reads` and `writes` (PackedInt32Array, 256 each), `reserved_read` (bool), and `frames`, `screen_writes` and `bitmap_writes` (int).

Save states

`Machine::save_state()` returns the whole computer as bytes, and `Machine::load_state(bytes)` puts it back exactly: the same bytes run the same future, cycle for cycle. In the game they're `save_state()`, which returns a PackedByteArray, and `load_state(bytes)`, which returns false for bytes it can't use.

The format, numbers low byte first:

Bytes	Holds
4	`UNCS`
1	Format version: 1
1	RAM: 16 or 64 (K)
18	The processor: A, X, Y, SP, PC, P, the cycle count, whether it's running, halted or jammed (and on what), and the IRQ line
65 + keys held	The I/O: VMODE, the raster compare, the interrupt registers, FRAME, KBD and the keys held, the keyboard matrix, the colours, CHARSET, BITMAP, the timer, the speaker, the shifter

Bytes	Holds
4	The computer's side of the tape interface: the motor, TAPEDATA, byte ready, overrun
4	Cycles into the current frame
312	The border colour on each line of the last frame (for the picture)
16K or 48K	RAM, from `$0000` to the top of memory

A UNC 16 with no key down saves to 16,793 bytes; with the 64K expansion, 49,561.

Not saved: the tape deck (the cassette in it, how far it has wound and the PLAY key), the host's `observe` counts, breakpoints, and sound already played. The deck isn't part of the computer, just as a reset doesn't touch it. `load_state` leaves the deck as the host has it and takes only the computer's side of the interface from the save, so a host that wants the tape back puts it in again. That's safe at any moment: a `LOAD` that was reading carries on with whatever tape it finds, and stops with `BREAK` or `?LOAD ERROR` if it isn't the same one.

`load_state` checks everything before it changes anything. Bytes that aren't a save state, a version this build doesn't know, a RAM size that isn't 16 or 64, or a state that's been cut short or damaged are refused, and the machine carries on as it was. A newer layout gets a new version number; an old one is never guessed at. After a load, the `observe` counts start from 0 and the speaker's sound starts afresh.