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

PART No. UNC-64-PRG-A · FIRST PRINTING, 1983

RADIO AND TELEVISION INTERFERENCE

Like every computer, the UNC 64 generates radio frequency energy. If the picture on a nearby television set crawls while the computer is on, and stops when you switch it off, move one of them. A metre is usually enough.

LOOKING AFTER YOUR COMPUTER

Keep it out of direct sunlight, away from radiators, and away from drinks. Do not cover the ventilation slots. Wipe the case with a slightly damp cloth, never with solvents. Switch off before you plug anything in or pull anything out.

LOOKING AFTER YOUR CASSETTES

Keep them in their boxes, and away from the monitor, loudspeakers, telephones and anything else with a magnet in it. Wind a tape back to the start before you put it away. Make a copy of anything you would be sorry to lose, and keep the copy somewhere else.

THIS BOOK

Describes System Rev A. Later revisions only add. Write in the margins: that is what they are for.

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 UNC BASIC 9

3 Bits, bytes and hex 18

4 The UNC-1 processor 19

5 Addressing modes 22

6 The assembler 24

7 Your first programs 31

8 The screen 36

9 Timing and the raster 51

10 Interrupts 55

11 The keyboard 59

12 Sound 63

13 The system ROM 72

14 Tape 80

15 The NIECE 87

16 The monitor 99

A Screen codes and colours 102

B Memory map and I/O registers 104

C Instruction set 109

D ROM reference 114

E Assembler messages 117

F For 6502 programmers 120

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, or never stop, and STOP and RESTORE together fix that (chapter 2); the RESET button, next to the power switch, fixes anything, and your program is still there.

Read chapters 1 to 7 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 4).
`fp`	An address in the fast page, `$00` to `$FF` (chapter 5).
`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.
_
```

As `READY.` appears, the machine beeps once, short and high: it's 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, or put it right where it stands: the cursor keys move the cursor back up onto the line, you type over the mistake, and RETURN gives BASIC the line again as the screen now shows it. 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, or `LIST` it, move the cursor up onto it, change it there and press RETURN (chapter 2, "The screen editor"). 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 2, next, is the whole of UNC BASIC. Chapter 8 says what the screen's places do, and Appendix B lists every one of them. 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, the AUNT with its three voices, and the tape deck: everything in chapters 1 to 14. The UNC 64 is the UNC 16 fully expanded, with two more parts in it: the 64K memory expansion, with 48K of memory below the I/O and ROM areas; and the NIECE, a graphics chip with sixteen colours and memory of its own (chapter 15). They are separate parts, and a machine can have either without the other. The start-up message tells you how much memory yours has (the ROM counts it every time you switch on); `PRINT PEEK(49487)` says 23 if the NIECE is in (chapter 15).

	UNC 16	UNC 64
Memory	16K, `$0000–$3FFF`	48K, `$0000–$BFFF`
Free for your programs	`$0800–$3FFF`, 14K	`$0800–$BFFF`, 46K
Graphics	text and the bitmap, in one colour	the same, and the NIECE behind them: 160 × 100 in sixteen colours
Everything else	identical	identical

The monitor and the video signal

The UNC monitor is a colour monitor. Inside its tube are three electron guns, red, green and blue, and the machine drives each of them down a wire of its own. What it sends down them is set by **SW1**, a block of four DIP switches on the main board, by the NIECE's socket. To reach it, switch the machine off and take out the four screws: the keyboard lifts up on its hinge. A switch is ON when it's pushed away from you, toward the back of the machine.

Switch 3 is COLOUR: each gun gets its own share of every colour, and the sixteen colours (Appendix A) show as they are. With switch 3 OFF, the machine sends only how bright each point of the picture is, its luminance, and switches 1 and 2 say which guns it goes to. Switch 4 isn't connected.

1	2	3	The guns	The picture
OFF	OFF	OFF	green alone	green, every colour a shade of it: as the machine leaves the factory
ON	OFF	OFF	red and green	amber
OFF	ON	OFF	all three	paper white
ON	ON	OFF	all three, over a blue ground	white on blue
any	any	ON	each its own share	colour

Why would anyone switch colour off? Three guns never land on quite the same spot, and forty columns of text are sharpest drawn by one: that's why the machine leaves the factory set to green, and why plenty of owners keep it there for typing, NIECE or not. The border and background colours (chapter 8) and the NIECE's sixteen (chapter 15) show as themselves only in colour; with switch 3 OFF, each is a lighter or darker shade. A program can't read SW1, and doesn't need to: the picture it makes is the same, and only the signal that carries it changes.

Typing

The keyboard sends letters, digits, punctuation, RETURN, DELETE, INSERT (shift and DELETE), the four cursor keys and STOP. As the machine is switched on, every letter arrives as a capital, whether you hold SHIFT or not: there is no lock key to forget. The machine has small letters too, and a program that wants them typed says so with one POKE (chapter 11). RESTORE, at the top right, sends nothing: it interrupts the processor (chapter 10), and with STOP held it brings BASIC back from anything (chapter 2).

2. 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.

This is UNC BASIC as System ROM Rev A has it. A later ROM may add keywords, after the last token (`$A7`); it won't change these, so a program stored by this ROM means the same to every later one.

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.

Small letters. BASIC is written in capitals, and as the machine is switched on capitals are all the keyboard sends. A program can ask the keyboard for small letters (`POKE 49171,1`: CASE, chapter 11), and BASIC then takes them for capitals everywhere except inside quotes and after `REM`. So `print "Hello"` runs as `PRINT "Hello"`, and that's how it lists: the keyword in capitals, the words in the quotes as you typed them. Variables are the same in either case: `a` is `A`. STOP and RESTORE put the keyboard back to capitals.

Lines and programs

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.

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 13).

The screen editor

The screen is the editor. The cursor goes anywhere on it, and when you press RETURN after moving it, BASIC reads the line the cursor is on, from the screen, exactly as it shows. It doesn't matter who put the line there: you just now, LIST a minute ago, or a program.

A line you simply type, without touching the cursor keys or INSERT, is taken as you typed it, from where the cursor began to where it is now, whatever else is on the row. So after a game has left its picture all over the screen, CLS typed in the middle of it is still CLS.

Key	Does
Cursor keys	Move the cursor a cell. Left from a row's first column goes to the end of the row above, and right from its last to the start of the row below. They stop at the screen's corners, and nothing scrolls.
A character	Goes where the cursor is, over whatever was there, and the cursor moves on.
DELETE	Rubs out the character before the cursor, and the rest of the line moves left to close the gap.
INSERT (shift and DELETE)	Opens a space at the cursor: the rest of the line moves right.
RETURN	Gives BASIC the line. If the cursor has been moved: the line it is on, all of it, wherever in it the cursor is. If not: what you typed. The cursor goes to the start of the row under the line.
STOP	Leaves the line as it looks and gives BASIC nothing.

So to change a line of your program, LIST it, move the cursor up onto it, put it right, and press RETURN:

```
LIST 20
20 PRINT "HELLO WORLF"
READY.
```

Cursor up twice and along to the F, type D, press RETURN. Line 20 is changed, and the cursor is on READY.'s row: move down to an empty row before you type the next thing, or you'll be typing over what's there. (Press RETURN on READY. itself and BASIC reads READ Y., and says ?OUT OF DATA.)

Change a line's number and press RETURN, and you have a second copy of the line under the new number: the old one is still in the program. Press RETURN on a line you typed and ran a moment ago, and it runs again.

Lines of more than one row. A line that reaches the right edge carries on along the next row, up to three rows: 120 characters. The editor remembers which rows are one line, so RETURN on any of them takes them all, DELETE and INSERT move characters across the edge, and DELETE at a row's first column goes back up onto the row above. When a line you're typing in the middle of the screen needs another row, the rows below move down to make room (the bottom one drops off the screen); on the bottom row, the screen scrolls up instead. At the end of the third row the cursor stays put, and INSERT on a line whose three rows are full does nothing.

What BASIC gets from the screen is the line without the spaces after its end, and with reverse video read as plain characters. A listed line that comes to more than 120 characters (every ? you typed lists as PRINT) shows on four rows, and the fourth isn't part of it: change it on the screen

and it's stored without its end.

INPUT uses the same editor. Once you move the cursor it reads from just after its prompt to the end of the line, so you can go back and put a number right before RETURN. Move the cursor to some other line and press RETURN, and that line is what INPUT reads, and most likely ?REDOs.

After a game. A game steered with the cursor keys often ends with one held down. Whenever BASIC prints READY., it waits for the cursor keys to be let go and forgets a cursor key that hasn't been taken yet, so the cursor is under READY. when you look, not half-way up the screen.

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.
MON	The monitor: memory in hex, and machine language listed back (chapter 16). Q there comes back to READY., with the program as it was.
HELP	A screen of help: every keyword, in groups, with what it does. It takes the whole screen, so it leaves graphics first, as TEXT does.
LOAD	Loads the first program on the tape (chapter 14). A BASIC program replaces yours, with its variables cleared, and BASIC says READY.; machine language starts.
LOAD "*name*"	The same, for the first program called *name*.
SAVE "*name*"	Records the program on the tape as *name*: PRESS RECORD AND PLAY ON TAPE, then SAVING *name* (chapter 14).
SAVE	The same, with no 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, with the screen editor. 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).

Statement	Does
`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.
`DATA` *items*	Numbers for `READ`, separated by commas. Running the line does nothing: `DATA` takes the rest of its line, so put it last on a line, or on a line of its own.
`READ` *v*,*w*...	Puts the next `DATA` item into *v*, the one after into *w*, and so on, in the order the items appear in the program, from the first `DATA` line to the last. `?OUT OF DATA` when there are none left.
`RESTORE`	The next `READ` starts again from the first `DATA` item. `RUN` and `NEW` do the same.
`END`	Ends the program.
`CLS`	Clears the screen. With `GR` on, the picture too, and the cursor goes to the first text row, as after `GR`.
`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.
`GR` *n*	The same with *n* text rows below the picture: `GR 0` for the whole screen, `GR 1` for one line of status at the bottom.
`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.

A `DATA` item is an expression too, worked out when it's read: `DATA 10, -3, $C020, 4*8`. That's how a type-in carries its tables (the levels, the shapes, the notes of a tune) without a `POKE` for every byte:

```
10 FOR I=0 TO 7:READ B:POKE 12288+I,B:NEXT I
20 DATA 24,60,126,255,24,24,24,24
```

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`. Chapter 3 explains hexadecimal.

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.
`GET`	The key waiting, as its code, and takes it; 0 if no key is waiting. It doesn't wait, so a game can keep moving: `K=GET:IF K=0 THEN 100`.

`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 gives up a line you're halfway through typing: it stays on the screen, and BASIC never sees it.

STOP and RESTORE. STOP is a key, and it only works on a program that looks at the keys. BASIC does, before every statement. A machine-language routine you called with `SYS` may not, and then STOP does nothing at all. Hold STOP and press RESTORE: the screen clears, the border and background go black, any sound stops, and BASIC says `READY.`, with your program and its variables still there. It works on anything short of a jammed processor (chapter 4); for that, press RESET, next to the power switch, which does the same from anything. RESTORE on its own does nothing, so you can't hit it by accident. Chapter 10 says exactly what it resets.

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 `GOSUB`s go more than 16 deep, or `FOR`s 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 14). After a BASIC program that's damaged, there's no program.
`?OUT OF DATA`	A `READ` with no `DATA` items left. An item that isn't an expression is a `?SYNTAX ERROR`; either way, the line number is the `READ`'s.

Graphics

`GR` turns the screen over to the bitmap (chapter 8): 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. `CLS` clears the picture and the text rows and stays in graphics; `TEXT` goes back to the text screen.

More picture, less text. `GR` *n* gives the picture all but *n* rows of text, each row eight lines: `GR 1` shows lines 0 to 191 with one row under them, where `PRINT` goes, for a score or a status line; `GR 0` shows all 200 lines and no text at all. *n* can be any expression, up to 25. `PRINT` still works with no rows showing; you just can't see it. So when a program ends, or you stop it, and fewer than two rows are showing, BASIC puts back `GR`'s four before it says `READY.`: there has to be somewhere to see it and somewhere to type. In direct mode that makes `GR 0` and `GR 1` the same as `GR`; they're for programs.

```
10 GR 1
20 FOR R=10 TO 90 STEP 10:CIRCLE 160,96,R:NEXT
30 PRINT "NINE RINGS. STOP TO END.";
40 GOTO 40
```

```
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 17 ms, and one corner to corner across the whole screen about 95 ms. A circle of radius 10 takes about 17 ms, and one of radius 90 about 0.12 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.

Those bytes, `$0800` to the end marker, are exactly what `SAVE` puts on tape and `LOAD` puts back. There's no address in them, only lengths, so a program saved on a UNC 64 loads on a UNC 16, if it fits in 14K.

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 14).

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
				`$A1`	LOAD
				`$A2`	SAVE
				`$A3`	DATA

Token	Keyword	Token	Keyword	Token	Keyword
				$A4	READ
				$A5	RESTORE
				$A6	GET
				$A7	MON

LOAD and SAVE came after the functions, DATA, READ, RESTORE and GET after them, and MON last, so the tokens before each kept their numbers. A later ROM adds keywords after $A7 the same way: a program stored by this ROM means the same to every later one.

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); GOTOs to lines near the start of the program, which BASIC finds sooner. When that isn't enough, it's time for machine language.

3. 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.

4. 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 10). 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 7).

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 10). `RTI` returns from an interrupt. `NOP` does nothing for 2 cycles. `HLT` stops the processor until a reset, or until RESTORE's interrupt wakes it (chapter 10).

Unassigned opcodes

Of the 256 possible opcodes, 136 are the instructions this book describes. Run into one of the other 120 and, as a rule, the processor **jams**: it stops at the faulty byte, and only a reset will restart it. Press RESET, on the monitor: your program is still there (chapter 13). 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 10). 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 `$24` (interrupts disabled), and every I/O register is cleared to zero. Switching on clears memory too; RESET leaves it as it is. The processor then loads PC from the reset vector at `$FFFC` and starts the system ROM, which can tell the two apart (chapter 13).

While the processor is stopped, halted or jammed, the rest of the machine goes on: the raster keeps scanning, the frame counter counts and the border keeps its colour, line after line. Only the processor waits: for the reset, or, if it's halted, for RESTORE (chapter 10).

5. 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.

6. 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).
`%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 its listing 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       ; a turn in 40 columns, 12 rows each side of 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`. Screen codes are ASCII, so the bytes show on the screen as the letters you wrote, small or capital.
`.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 finds the file next to the program it's assembling; the Machine Room finds it among the project's files, by its path (`.include "lib/print.s"`). 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 assembler's listing 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 14), 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.

7. 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 13). `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 8). 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, ends 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 9). 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.

8. 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–$07`, `$0F`	Quarter blocks: bits 0-2 light the top-left, top-right and bottom-left quarter of the cell; `$0F` is all four.
`$08–$0E`	A cell filled from the bottom: one to seven eighths of it, for bar graphs.
`$10–$1F`	Lines, corners and joints for drawing boxes; diagonals; two shades.
`$20–$7E`	ASCII: punctuation, digits, capital letters, small letters. A character's screen code is its ASCII code.
`$7F`	A diamond.
`$80–$FF`	The glyphs of `$00–$7F` in reverse video. (In 256-glyph mode, 128 more glyphs: see "256 glyphs".)
`$A0`	A reversed space, which blinks (unless VMODE bit 3 is set; see below). BASIC's screen editor uses it as the cursor.

The quarter blocks are numbered so you can work them out: `$03` is the top half of a cell, `$05` its left half, `$0F` a solid block. The seven with the bottom-right quarter lit (and not all four) aren't in this half: each is one of `$01–$07` reversed, so for a pattern `n` from 8 to 14, by the same bits with bit 3 the bottom-right, the code is `n` exclusive-or `$8F`. With them, the 40 × 25 screen becomes an 80 × 50 grid of dots. The eighths do the same for heights: a column of cells is 200 levels tall, `$0B` is the bottom half of a cell, and reversed (`$88–$8E`) the same seven glyphs fill a cell from the top.

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 picture is drawn one raster line at a time, border and background alike, so a program that changes either colour as the beam goes down can draw stripes across it (chapter 9).

UNSCII, the character set

UNSCII is 256 glyphs, built into the character ROM at `$D000`–`$D7FF`: the 128 the screen uses as it's switched on, and 128 more for 256-glyph mode (below). 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. Small letters with tails (g, j, p, q, y) hang into the eighth row. 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's first half 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 diamond

        .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 `$A0` by itself, about once a second, so the cursor BASIC's screen editor shows while it waits for you to type (chapter 13) needs no program to blink it. Printing never draws it, so a program only sees `$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 screen editor's cursor is whatever you draw for code `$A0`.

The ROM's second half. With CHARSET at `$D0`, as at reset, glyphs `$80–$FF` come from the second kilobyte of the character ROM, `$D400–$D7FF`, and they're pictures and signs that the first half has no room for:

Codes	Show
`$80–$86`	The quarter blocks the first half leaves out, the ones with the bottom-right quarter lit: quarter pattern `n` (8 to 14) is at `n + $78`.
`$87–$8F`	More eighths, for bar graphs and smooth edges: `$87–$8D` fill the left 1 to 7 eighths of the cell; `$8E` is the top eighth and `$8F` the right.
`$90–$9F`	Card suits, a ball, a ring, arrows left, up, right and down, four triangles, a face, a note.

Codes	Show
`$A0-$A6`	`$08-$0E` reversed: a cell filled from the top, all but its bottom 1 to 7 eighths.
`$A7-$BF`	`$87-$9F` reversed: the other eighths, and the pictures cut out of a solid cell.
`$C0-$CA`	Double lines, corners, joints and a cross, in the order of the single ones at `$10-$1A`.
`$CB-$D6`	Round corners; squares, stripes across and down, and a dark shade; and a ball four cells big, in its four quarters.
`$D7-$E6`	Letters with accents: Å Ä Ö å ä ö Ø ø Æ æ É é Ü ü ß ñ.
`$E7-$F6`	More signs: £ ¥ § ° ± × ÷ π µ ½ ¼ ¾ ¿ ¡ « ».
`$F7-$FF`	Things to play with: a ship, an alien, a man, a tree, a house, a key, a star, a bomb, a skull.

Appendix A has the whole table. One store gets you all of them, at the price of reverse video:

```
LDA #1
STA VMODE       ; $C000: 256 glyphs
LDA #$91
STA $0400       ; a heart, top left
```

Reverse video is still there if you only want it for the pictures: `$A7-$BF` are `$87-$9F` reversed, and `$A0-$A6` the eighths of `$08-$0E`. The cursor's code, `$A0`, is one of them (the bottom eighth, reversed), so the screen editor's cursor is a steady block with a gap along its foot.

A set of your own, 2K. For 256 glyphs of your own, the set is 2K in RAM, and you fill both halves. This program builds one, UNSCII's first half in the bottom and its reversed copy in the top (which is what the screen shows without 256-glyph mode), 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       ; $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 9), 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 reset puts it 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 2). The split is at raster line 219 (chapter 9): lines 51-218 show bitmap lines 0-167, and lines 219-250 show text rows 21-24.

How many rows. Four is what TEXTROWS holds at reset. Store another number there and that many rows of text show under the bitmap, the last ones of the screen, with eight lines fewer of picture for each: 1 leaves bitmap lines 0-191 and shows row 24 alone, and 0 leaves the whole bitmap and no text, the same picture as bit 1 without bit 2. It takes 25 at most, which is all text. BASIC's GR *n* stores it for you (chapter 2). The video chip looks at TEXTROWS on every line, as it does at everything, so a program that changes it as the beam goes down can have text in the middle of the picture too (chapter 9).

Address	Register	Read and write
$C024	TEXTROWS	In mixed mode, the rows of text under the bitmap: 0 to 25. 4 at reset.

The bitmap's bottom 32 lines are still in memory; they're just covered up. In mixed mode with its four rows 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, a line at a time, from what memory holds as it gets there (chapter 9). A program that changes the picture while the beam is drawing it 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, for that one frame. The cure is to change the picture where the beam isn't. FRAME goes up as the beam leaves the picture, at line 251, and from there the beam doesn't draw again until line 51 of the next frame: 112 lines, 7,168 cycles. A game that waits for FRAME and does its drawing first has all of that. Work that takes longer can keep going below the beam (draw only what the beam has already passed) or wait for it: in mixed mode, from line 219, where the bitmap ends, the bitmap is out of the beam's way for 144 lines.

One bitmap, one bit per pixel, in the screen's ink and paper. The NIECE (chapter 15), if the machine has one, draws in sixteen colours behind it.

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.

Address	Register	Read	Write
`$C051`	SHIFT_AMOUNT	The amount, 0-7.	How many pixels to move right. Only bits 0-2 count, so storing *x* itself sets *x* AND 7.
`$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: at power-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.

SHIFT_IN and SHIFT_FLIP read `$FF`; SHIFT_OUT and SHIFT_AMOUNT ignore writes (SHIFT_AMOUNT's bits 3-7 read 0). The window starts at 0 at switch-on.

9. 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 as the beam leaves the picture, at the start of line 251, 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 10).

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.

The picture, line by line

The beam draws each line of the picture as it finishes the line, from whatever the machine holds at that moment: screen memory and the character set, the bitmap, VMODE, CHARSET, BITMAP, BACKGROUND, and with the NIECE its page as it stands on the line's last cycle. So everything is a raster split, not only the border: change BACKGROUND, VMODE or the character set at line 150 and the lines from 150 down show the change, in the same frame. A line takes what the machine held when the last instruction to finish on it finished, as the border does. Each line is drawn whole: there are no changes part way along a line.

```
; A raster split: the background turns red halfway down the text rows, every frame.
RASTER  = $C001
BG      = $C021
SPLIT   = 51+12*8       ; the first line of row 12

        .org $0800
start:  SEI
frame:  LDA #0          ; black from the top of the frame
wtop:   CMP RASTER
        BNE wtop
        LDA #0
        STA BG
        LDA #SPLIT
wsplit: CMP RASTER
        BNE wsplit
        LDA #2          ; red from row 12 down
        STA BG
        JMP frame
```

The beam never waits for the processor, and the processor never waits for the beam: drawing steals no cycles, ever. A frame is always 19,968 cycles of your program's.

FRAME goes up, and the vertical blank interrupt comes (chapter 10), as the beam leaves the picture: at the start of line 251, the first line of the bottom border. From there to line 51 of the next frame, 112 lines, nothing in the picture is drawn, so that's when a game changes it. What the beam drew in a frame is what the monitor shows.

10. 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 once every frame, as the beam leaves the picture at the start of line 251, when FRAME goes up. 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, 112 lines

before it's back at the top of the picture.

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 RESTORE key

The processor has a second interrupt line, the **non-maskable interrupt**, and one thing is wired to it: the RESTORE key. RESTORE isn't read like the other keys. It isn't in KBD or the matrix (chapter 11). Pressing it interrupts the processor, once for each press, however long it's held.

An NMI is taken like an IRQ, with three differences. It doesn't ask the I flag: `SEI` can't keep it out. It goes through the vector at `$FFFA`, not `$FFFE`. And it wakes a processor stopped by `HLT`.

1. The processor finishes the instruction it's on.
2. It pushes PC (high byte first), then P with the B bit clear.
3. It sets I.
4. It jumps to the address stored at `$FFFA-$FFFB`.

That takes 6 cycles. The ROM's handler there does `JMP ($0316)`, 5 more, and `$0316-$0317`, NMIVEC, is yours to change, as IRQVEC is. `RTI` goes back to where the program was.

Woken from a `HLT`, the processor pushes the address of the `HLT` itself, so `RTI` runs the `HLT` again and the processor stops again: an NMI doesn't set a halted program going by accident. A **jammed** processor (chapter 4) stays jammed. Only a reset helps that: the RESET button.

STOP and RESTORE. The handler the ROM puts in NMIVEC looks at the STOP key (bit 7 of the matrix's row 8). If STOP isn't down, it returns: RESTORE alone does nothing. If it is, the handler never returns to your program. It:

1. Points IRQVEC and NMIVEC back at the ROM's own handlers.
2. Sets the border and background to black, VMODE to 0 (the text screen, 128 glyphs, the blinking cursor) and CHARSET to `$D0`, the ROM's characters.
3. Turns every interrupt source off (IRQ_ENABLE 0) and acknowledges what's waiting, and puts the keyboard back to capitals (CASE 0).
4. Stops the speaker (SPEAKER 0) and the tape's motor.
5. With the NIECE: NCTRL 0, which takes its page out of the picture (chapter 15).
6. Clears the screen, empties the stack, clears I, and goes to BASIC's `READY.`.

Memory isn't touched. The BASIC program and its variables are as they were, and so is your machine language, wherever it is: `SYS` will call it again. So STOP and RESTORE is the way out of a program that STOP alone can't reach, one that never looks at the keyboard or has halted, and you lose nothing but the picture.

A program that wants RESTORE for itself points NMIVEC at its own handler, which ends with `RTI`. While it holds the vector, STOP and RESTORE is its business too, and the RESET button is the only way out it can't close.

```
NMIVEC  = $0316
        LDA #<pause
        STA NMIVEC
        LDA #>pause
        STA NMIVEC+1    ; RESTORE now calls pause, at any moment
```

An NMI can arrive between any two instructions, in the middle of anything, even inside your IRQ handler. Keep the handler short, and save what it uses.

11. 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.
`$C012`	HELD	The key that is down now, or 0. See below.	
`$C013`	CASE	Bit 0, as written.	Bit 0: 1 and letters arrive as they're typed. 0 at reset. See "Small letters".

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`, `$41-$5A`. With CASE set, small letters too: `a-z`, `$61-$7A`
Digits and punctuation	Their ASCII codes, `$20-$7E`
RETURN	13
DELETE	8
INSERT (shift and DELETE)	20
STOP	3
RESTORE	none: it interrupts the processor (chapter 10)
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 screen editor moves its cursor with the cursor keys and opens a space with INSERT (chapter 2); while a program runs they're the program's, and 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.

Small letters. As the machine is switched on, CASE is 0 and every letter arrives as a capital, with SHIFT or without: a program that asks whether the key is `A` gets one answer. Store 1 in CASE and letters arrive as they're typed: small (`a` is `$61`), or capitals with SHIFT, in KBD and in HELD. There is no key that changes it; it's a store, by the program that wants small letters, and STOP and RESTORE store 0 again (chapter 10). In BASIC, `POKE 49171,1`. The signs `` ` { | } ~ `` (`$60`, `$7B-$7E`) arrive either way.

```
        LDA #1
        STA CASE        ; $C013: letters as typed
```

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:
```

HELD shows one key; for two or more at once (move and fire), read the keyboard matrix, below. The repeat delay and rate belong to the computer the emulator runs on, not to the UNC 64.

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	INSERT	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, INSERT, and STOP in bit 7, where `BIT $C068` puts it in N: `BMI` and the player wants out. A letter is one key in the matrix, small or capital, whatever CASE says, and the five signs above `$5F` have no place in it.

```
; 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.

The keyboard of the computer the emulator runs on may not report every combination of three or more keys at once; that's its limit, not the UNC 64's.

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
```

12. Sound

The UNC makes sound two ways. The **speaker**, the beeper, is one bit: in or out. The **AUNT**, the Audio Unit, is three voices with a shape, a loudness and a filter: an instrument. The beeper comes first here, as it does for most people; the AUNT has the second half of the chapter.

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 speaker is wired as the IBM PC's was: a counter chip, a gate and a data bit.

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. Then the ROM beeps once as `READY.` appears, 880 Hz for 300 ms (chapter 13), and puts them back to 0.

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 off 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 10), 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 / (2*n*) 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 + 5*N* - 1 + 3 = 5*N* + 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 13), 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.

The emulator plays the speaker through the computer it runs on: the Machine Room, the showcase and the game take each frame's speaker positions and make sound of them, averaged over each sample and gently filtered so that a speaker at rest, in or out, is silent, at a modest volume. The browser pages have a Mute switch, and start their sound at the first click or key, as browsers require.

The AUNT

The AUNT is the UNC's Audio Unit: a chip of its own beside the speaker, in every UNC from the 16 up. It has **three voices**. Each one has a pitch, a waveform, and an envelope that brings the note up and lets it die away. One **filter** takes the top or the bottom off any of them. They all come out of the same loudspeaker as the beeper, together.

It lives at `$C0C0-$C0FF`. Each voice is eight registers; voice 1 starts at `$C0C0` (49344), voice 2 at `$C0C8` (49352), voice 3 at `$C0D0` (49360).

Offset	Register	Write
+0	FREQ_LO	The pitch, low byte.
+1	FREQ_HI	The pitch, high byte. FREQ is the pitch in Hz × 16.777: A at 440 Hz is 7,382.
+2	WIDTH	The pulse wave's high part, in 256ths. 128 is a square.
+3	WAVE	Bit 0 triangle, bit 1 sawtooth, bit 2 pulse, bit 3 noise, bit 4 the wave table. Bit 7: short noise.
+4	CONTROL	Bit 0: the **gate**. Bit 1: sync. Bit 2: ring.
+5	ATTACK_DECAY	The attack's rate in the high four bits, the decay's in the low four.
+6	SUSTAIN_RELEASE	The level held in the high four bits, the release's rate in the low four.
+7	TRIM	Bits 0-3: sixteenths taken off the voice's level.

And the chip's own:

Address	Register	Read	Write
`$C0D8`	CUTOFF	As written.	The filter's corner, 0-255: 30 Hz at 0, an octave higher every 32, about 7,500 Hz at 255.
`$C0D9`	FILTER	As written.	Bits 0-2: voice 1, 2, 3 goes through the filter. Bits 4-7: resonance.
`$C0DA`	VOLUME	As written.	Bits 0-3: volume. Bit 4 low-pass, bit 5 band-pass, bit 6 high-pass. Bit 7: voice 3 silent.
`$C0DC`	VOICE3	Voice 3's waveform at this moment, its top 8 bits.	

Address	Register	Read	Write
`$C0DD`	ENV3	Voice 3's envelope at this moment, 0-255.	
`$C0DF`	SOUND	`$16`.	
`$C0E0-$C0FF`	WAVETABLE	As written.	A waveform of your own: 32 steps, each 0-255.

The voices' registers read back what was written. At switch-on every register is 0, and the AUNT is silent.

A first note. Nothing sounds until the volume is up, a waveform is chosen and the gate is on. From `READY.`:

```
POKE 49370,15
POKE 49347,2
POKE 49350,240
POKE 49344,214
POKE 49345,28
POKE 49348,1
```

Volume 15; a sawtooth; hold at full level; A (7,382 is 28 × 256 + 214); and the gate. It sounds until `POKE 49348,0` lets it go.

The gate and the envelope. A note isn't on or off: it has a shape. While the gate is 1 the voice's level climbs to full at the **attack** rate, falls to the **sustain** level at the **decay** rate, and stays. When the gate goes to 0 it falls to nothing at the **release** rate. The rates run from 0, the fastest, to 15:

Rate	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Attack	2 ms	8	16	24	38	56	68	80	100	250	500	800	1 s	3	5	8

A decay or a release at the same rate takes three times as long, and slows as it nears the bottom, as a struck string does. Sustain is a level, not a time: 15 is full, 0 is nothing. With sustain 0 a note dies by itself, gate or no gate: a pluck, a drum. To strike the same voice again, take the gate off and put it on.

The waveforms. Triangle is soft and hollow, a flute. Sawtooth is bright and buzzing, strings and brass. Pulse depends on WIDTH: at 128 a hollow square, and thinner and more nasal as WIDTH moves away from it; change WIDTH a little every frame and the sound moves. Noise is a hiss, and it has a pitch like the others: FREQ sets how fast it hisses, and with FREQ 0 it doesn't hiss at all. Short noise repeats after 127 steps: a metallic buzz. The wave table is a shape you draw yourself, 32 levels in WAVETABLE, played round once for each cycle of the note.

The filter. A voice goes through the filter if its bit in FILTER is set; the others go round it. VOLUME says what comes out of it: the low-pass lets through what's below CUTOFF, the high-pass what's above, the band-pass what's near it. Resonance makes the filter ring at its corner. With the first note still sounding:

```
POKE 49369,1
POKE 49370,31
POKE 49368,128
```

Voice 1 through the filter; volume 15 with the low-pass (16 more); the corner at 480 Hz. The buzz goes and the note stays. Move CUTOFF while a note sounds and the sound opens and closes.

A drum is noise that dies by itself:

```
POKE 49347,8
POKE 49345,60
POKE 49349,8
POKE 49350,0
POKE 49348,0
POKE 49348,1
```

Sync and ring tie a voice to the one before it (voice 1 to voice 3). With sync, the voice starts its wave again each time the other finishes one. With ring, the other voice turns this one's triangle upside down as it goes, which makes bells and gongs. Neither needs the other voice to be heard: give it a pitch and no waveform.

Voice 3 read back. VOICE3 and ENV3 show voice 3's waveform and envelope as they are at the cycle of the read. Run voice 3 slowly with a triangle, silence it with VOLUME bit 7, and VOICE3 is a number rising and falling by itself: add it to another voice's FREQ once a frame and that voice has a vibrato. With noise it is a second source of random bytes.

From machine language the AUNT is ordinary stores. This plays a note and lets it go half a second later:

```
; A for half a second on voice 1 of the AUNT.
AUNT    = $C0C0
FREQ_LO = 0
FREQ_HI = 1
WAVE    = 3
CONTROL = 4
SUSTAIN_RELEASE = 6
VOLUME  = $C0DA
FRAME   = $C006
A440    = 7382

        .org $0800
start:  LDA #15
        STA VOLUME
        LDA #2                  ; sawtooth
        STA AUNT+WAVE
        LDA #$F8                ; hold at full level; release at rate 8
        STA AUNT+SUSTAIN_RELEASE
        LDA #<A440
        STA AUNT+FREQ_LO
        LDA #>A440
        STA AUNT+FREQ_HI
        LDA #1
        STA AUNT+CONTROL        ; the gate on
        LDX #25                 ; 25 frames: half a second
@frame: LDA FRAME
@wait:  CMP FRAME
        BEQ @wait
        DEX
        BNE @frame
        LDA #0
        STA AUNT+CONTROL        ; and off: the note dies away
        HLT
```

A tune is this once a frame for each voice: a routine on the vertical blank (chapter 10) that counts frames, and at each new note writes FREQ, takes the gate off and puts it on. The AUNT makes a new sample of sound every 16 cycles, and a write takes effect on the cycle it is made.

13. 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.

Below it is a second ROM, 4K, at `$D000–$DFFF`. Its first half is the character set (chapter 8). Its second half, from `$D800`, is more of the system: HELP's screen and the monitor (chapter 16), and room for what a later ROM adds. Nothing in it has a promised address. The way into the ROM is the jump table.

What it does at start-up

The reset vector points at `$FFE0`, the top of the ROM. It sets the stack pointer to `$FF` and looks at WARMMK, `$0318–$0319`. If it holds `$55 $AA`, the machine has been running and someone pressed RESET: the ROM does what STOP and RESTORE does (chapter 10), and BASIC says `READY.` with your program and its variables as they were, whatever the processor was doing, jammed or not. Switched on, memory is all zeros, so it's a cold start:

1. Sets the stack pointer to `$FF`.
2. Points the interrupt hooks, IRQVEC at `$0314` and NMIVEC at `$0316`, to its own handlers.
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, seeds RND, and sets WARMMK to `$55 $AA`: from now on a reset is warm.
6. Clears the screen and prints the start-up message, with the memory and the bytes free for BASIC. (It doesn't look for the NIECE: a program that needs it does, chapter 15.)
7. Clears I, and prints `READY.`.
8. Beeps: 880 Hz for 300 ms on the speaker's timer (divisor 1,136; chapter 12), counted out in 15 frames on FRAME. Then the timer's gate and the speaker's data go back to 0.
9. Hands over to UNC BASIC, which waits for a line (chapter 2). Keys pressed during the beep wait in KBD.

Up to `READY.` it takes about 23,000 cycles, a little over a frame; the beep takes 15 frames more.

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

Address	Routine	In	Out	Changes
$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: machine code from the tape starts, a BASIC program goes to BASIC's READY., or BASIC prints the error (chapter 14)	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
$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
$FF24	MON		Doesn't return: the monitor (chapter 16)	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 screen editor ("The screen 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 12), say. Calling RND from machine language moves BASIC's sequence on too.

MON is the monitor, from machine language: JMP $FF24 where a program would otherwise stop, and its memory is there to look at (chapter 16).

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 4), 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, 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.

CHROUT also keeps the screen editor's note of which rows belong together (LINK, below): a row the cursor wrapped onto carries on the line of the row above, up to three rows, and a row that code 13 moved to starts a line of its own. That's how RETURN on a listed line knows where the line ends. Storing straight to screen memory leaves the notes alone.

The screen editor's cursor

The cursor is part of BASIC's screen editor, not of printing: it's there only while the machine is waiting for you to type. At READY. and at an INPUT, the editor shows a cursor (code $A0, a reversed space, which the video chip blinks: chapter 8) 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.

The same editor reads every line BASIC waits for (chapter 2, "The screen editor"). It keeps three things in memory. LINK ($03DA–$03F2) has a byte for each row of the screen: 1 if the row carries on the line of the row above, 0 if a line starts on it. Clearing the screen sets them all to 0, scrolling moves them up with their rows, and row 0's is always 0. RLCOL and RLROW ($03D8–$03D9) are where the cursor was when the editor began to wait: RETURN on that line reads from that column, which is how INPUT leaves out its prompt. RLMOV ($03F3) is 0 until a

cursor key or INSERT is pressed: while it's 0, RETURN takes what was typed, from RLCOL to the cursor. Whenever BASIC prints `READY.` it reads row 8 of the keyboard matrix (chapter 11) and waits while a cursor key is down.

A program started from the Machine Room starts on BASIC's screen just as BASIC left it, `READY.` and all (chapter 6), 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 6,700
CHROUT	about 125, on any row
CHROUT, when it scrolls	about 18,400: nearly a whole frame
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.

These figures are for ROM Rev A, and a later ROM may be faster. The jump-table addresses, and what each call does, won't change.

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 screen editor's cursor.
`$0314-$0315`	IRQVEC	The interrupt handler's address.
`$0316-$0317`	NMIVEC	The NMI handler's address: the RESTORE key (chapter 10).

Address	Name	Holds
$0318–$0319	WARMMK	$55 $AA once switched on: RESET keeps the program. Change it, and the next RESET is a cold start.
$03D0–$03D7	NUMBUF	NUMSTR's eight digits.
$03D8–$03D9	RLCOL, RLROW	The column and row where the screen editor began to wait.
$03DA–$03F2	LINK	A byte a row: 1 if the row carries on the line of the row above.
$03F3	RLMOV	Not 0 once the screen editor's cursor has been moved.

UNC BASIC also uses $D0–$F6 in the fast page and most of $0204–$03F3; 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	JMP ($0316): the RESTORE key (chapter 10).
$FFFC	Reset	The start-up code, at $FFE0: a cold start, or a warm one after RESET.
$FFFE	IRQ and BRK	JMP ($0314).

14. 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: machine language starts, and a BASIC program waits for you to RUN it. SAVE records your BASIC program on a tape, to load again tomorrow or to give to a friend. 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, or, for a BASIC program, says READY..

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 6: 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 **BASIC program** (type 2, below) always replaces the resident program: LOAD forgets yours, as NEW does, puts the one from the tape at $0800, clears the variables and the FOR and GOSUB stacks, and prints READY.. It doesn't run it: type LIST to see it, RUN to run it.

```
LOAD "HELLO"
PRESS PLAY ON TAPE
SEARCHING FOR HELLO
FOUND HELLO
LOADING
READY.
```

A BASIC program needs room below the top of memory (below the bitmap while `GR` is on), or it gives `?OUT OF MEMORY` and leaves your program as it was. It doesn't care which machine saved it: a program saved on a UNC 64 loads on a UNC 16 if it fits.

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 13): 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. A BASIC program that stops part way leaves no program at all (as `NEW` does): its header said it would replace yours. One that's damaged inside, its lines not adding up to its length, gives `?LOAD ERROR` the same way. `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 13): 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 a machine-language program, or goes to BASIC's READY. with a BASIC program loaded or after printing its message.

Saving a BASIC program

Put a tape in the deck, a blank one or one with room after what's on it, and type SAVE with a name:

```
SAVE "HELLO"
PRESS RECORD AND PLAY ON TAPE
SAVING HELLO
READY.
```

1. If RECORD and PLAY aren't down, SAVE asks for them with PRESS RECORD AND PLAY ON TAPE, and waits.
2. Press RECORD (it takes PLAY down with it). SAVE prints SAVING and the name, switches the motor on and records the program: a header with its name, the program from $0800 to its end marker, and a checksum (the format, below). The border shows the bytes going out, as it does when they come in.
3. At the end it switches the motor off, puts the border back, and BASIC says READY.. Your program is still there, and so are its variables.

The name works as LOAD's does: up to 16 characters in quotes; more are left off. SAVE alone saves the program with no name: LOAD with no name finds it, as it finds any first program, but LOAD "NAME" never will, so give programs names.

The program goes on the tape where the tape is: at its start for a tape just put in, after the last program for a tape you've just saved or loaded to the end. Programs saved one after another follow each other, and LOAD "NAME" finds each of them later. Recording goes over whatever was on the tape there, as on any cassette.

Saving takes as long as loading: about 600 bytes a second, and 29 more for the header and the checksum. A program of 1,000 bytes takes under two seconds. SAVE works in a program too: it records the program it's in, and the program carries on.

Stops SAVE	Message
The STOP key, while it waits or while it records	BREAK
STOP on the deck: RECORD and PLAY come up	BREAK

Either way the motor goes off and the border comes back, and your program is untouched. What was recorded so far is only part of a program, which LOAD can't use: save it again.

The deck

The deck has three keys. **PLAY** stays down while the tape plays. **RECORD** goes down with PLAY, as on any cassette recorder, and records instead of playing. **STOP/EJECT** lets PLAY and RECORD up; pressed again, it ejects the tape. A cassette goes in at its start. A **blank tape** has nothing on it yet.

The tape moves only while PLAY is down **and** the computer runs the motor. That's how LOAD and SAVE can stop the tape and start it again without your help. At the end of the tape, the deck lets PLAY up by itself, unless it's recording: then the tape goes on as long as there's something to record.

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 3: RECORD is down. Bit 4: room for a byte (recording). Bit 6: overrun (playing). Bit 7: a byte is ready (playing)	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	With RECORD down, the next byte to record: TAPE bit 4 clears until the deck has it. Without, 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 13): 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.

Recording works the other way round. With RECORD down, nothing comes off the tape (bits 6 and 7 stay 0), and bit 4 of TAPE says there's **room for a byte**. Write the byte to TAPEDATA: bit 4 clears, and at the deck's next byte time, every 1,664 cycles while the motor runs, the byte goes on the tape and bit 4 sets again. A byte written while bit 4 is clear takes the place of the one waiting, which is never recorded. A byte time with nothing waiting records nothing, so the tape has no gaps for a slow program to leave; it only takes longer. Wait for bit 4 once more after the last byte, so that it's on the tape before you switch the motor off.

```
; Record 256 bytes from $3000 on the tape yourself
TAPE    = $C040
TDATA   = $C041
BORDER  = $C020
ROOM    = $10           ; TAPE bit 4: room for a byte
        .org $0800
start:  LDA #4
        STA TAPE        ; motor on
        LDX #0
next:   LDA TAPE
        AND #ROOM       ; wait for room: RECORD down, the last byte recorded
        BEQ next
        LDA $3000,X
        STA TDATA       ; the deck records it at the next byte time
        STA BORDER      ; stripes
        INX
        BNE next
last:   LDA TAPE        ; and the last one too, before the motor stops
        AND #ROOM
        BEQ last
        LDA #0
        STA TAPE        ; motor off
        HLT
```

Run it with a blank tape in the deck and RECORD down. Without RECORD, bit 4 stays 0 and it waits: press RECORD, and the bytes go on the tape, 256 of them in under half a second.

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. 2 = a BASIC program
6	16	Name: capitals, padded with spaces
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 EORs 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 or a type it doesn't know rather than guess.

Type 2, a BASIC program, is the program exactly as BASIC keeps it in memory (chapter 2): from `$0800` up to and including its end marker.

Field	Type 2 holds
Load address	`$0800`. `LOAD` gives `?LOAD ERROR` for anything else
Length	The program's bytes, its end marker included: PRGEND − `$0800` + 1. At least 1 (an empty program is just the end marker)
Entry address	0: not used
The program	Line after line: a length byte (the whole line, in bytes, at least 4), the line number (low byte first), the text with its tokens, and a 0. Then a 0 length byte, the end marker, as the very last byte

There are no addresses in it, only lengths, so the same bytes work wherever the program sits, on a UNC 16 or a UNC 64. The variables aren't on the tape: `LOAD` clears them, and `RUN` would anyway. Before it hands the program to BASIC, `LOAD` walks its lines: each at least 4 bytes and ending in 0, and the end marker exactly at the end. Anything else is `?LOAD ERROR`.

The tape carries bytes, not sound: the deck is a byte device, and a program sees bytes, never a bit stream. `SAVE` records BASIC programs only; machine language goes on tape on the host: `unc-asm prog.s --tape prog.tap [--name NAME]` (`docs/machine-room.md`), or `unc_asm::tape::make` in Rust. A BASIC listing goes on tape on the host with `unc-asm --basic prog.bas --tape prog.tap`, which types it into a machine and saves what BASIC made of it, or `unc_asm::tape::basic` in Rust. Hosts get a recorded tape back with `Machine::eject_tape` (`docs/hosts.md`). `LOAD` and `SAVE` use `$02E4-$030F` and `$D0-$D3` (Appendix D).

15. The NIECE

The NIECE is a graphics chip, one of the two parts that make a UNC 16 a UNC 64 (the other is the 64K memory expansion: they're separate, and each works without the other). Its name is the **Northern Image Engine, Colour Extension**; the chip itself is marked NV-1, for Northern Video.

The video chip shows what's in the computer's memory. The NIECE works the other way round. It has 64K of memory of its own, which the processor can't see, and it does its own drawing: you tell it *what* to draw, in a few registers, and it draws. It fills boxes, draws lines, and copies a picture from one place to another at any size, bigger or smaller, in sixteen colours. The processor never touches a pixel, so a 1 MHz processor is enough to decide what a whole screen of them should be.

It also has an **arithmetic unit**, which multiplies and divides in no time at all. The two belong together: the sums that say where things go are the other half of drawing them.

Is it there?

Address	Register	Read
`$C14F`	NIECE	`$17`: the NIECE. Writing does nothing.

NIECE reads `$17` with the chip in. Without it, NIECE reads `$FF`, like every unused address: every address from `$C100` to `$C14F` does, and ignores what you write.

The start-up message doesn't mention it: it counts the memory, and that's all. From BASIC, `PRINT PEEK(49487)` says 23 with the NIECE in and 255 without.

BASIC has no words for the NIECE, though `PEEK` and `POKE` reach its registers like any others. STOP and RESTORE take its page out of the picture (chapter 10), so the way back to `READY.` works from a program that was showing one. A program that needs the NIECE looks for itself, and says so if it isn't there:

```
        LDA $C14F
        CMP #$17
        BNE noniece
```

The sheet and the two pages

The NIECE's memory is one big picture, the **sheet**: 256 pixels wide and 512 lines high. Every pixel is four bits, a colour from 0 to 15, the same sixteen as BORDER's (Appendix A). Two pixels share a byte, the left one in the high four bits, so a line of the sheet is 128 bytes and the whole sheet is 65,536.

```
byte = y * 128 + x / 2
left pixel (x even): bits 4-7      right pixel (x odd): bits 0-3
```

The top left of the sheet is two **pages**, each 160 × 100 pixels:

	Sheet's lines	Sheet's columns
Page 0	0-99	0-159
Page 1	100-199	0-159
Yours	0-199	160-255
Yours	200-511	0-255

One page is **shown** and the other is **drawn on**. When a picture is finished, the two change places (FLIP, below), so nobody ever sees a picture half drawn. Page 0 is shown at switch-on, when the whole sheet is colour 0.

The rest of the sheet is yours: 99,072 pixels to keep pictures in, for the NIECE to copy onto the page.

In the picture, the page shown is behind the computer's own. Each of its pixels is two of the screen's wide and two high, so 160 × 100 fills the screen's 320 × 200. Wherever the computer's own picture is paper, you see the page; text and the bitmap's ink are in front, in white. So the text screen is still there for a score, and a space is a window. BACKGROUND doesn't show while the page does.

Address	Register	Read and write
`$C100`	NCTRL	Bit 0: 1 shows the page. 0 at reset, and after STOP and RESTORE.

Commands and the queue

To draw, set the registers that say what and where, then write a command's number to NGO.

Address	Register	
`$C103`	NGO	Write only. 1 FILL, 2 BLIT, 3 LINE, 4 FLIP. Any other number does nothing.
`$C101`	NSTATUS	Read only. Bit 7: busy, the queue isn't empty. Bit 6: the queue is full. Bit 5: a command was lost; it stays set until anything is written to NSTATUS. Bit 0: the page shown, 0 or 1.
`$C102`	NQUEUE	Read only. How many commands are in the queue, 0-64.

The command doesn't happen at once. It joins a **queue** of up to 64 commands, and it takes the registers with it *as they stand when NGO is written*: change them straight afterwards for the next command, and the one in the queue keeps what it had. The NIECE works through the queue in order while your program carries on.

A command written when the queue already holds 64 is **lost**: it never runs, and bit 5 of NSTATUS says so, for as long as it takes you to look: reading NSTATUS doesn't clear it, writing it does. A program that might fill the queue looks at bit 6 first:

```
@full:  BIT NSTATUS         ; bit 6 goes to V
        BVS @full           ; full: wait
        STA NGO
```

And to wait until everything asked for has been drawn:

```
@busy:  BIT NSTATUS         ; bit 7 goes to N
        BMI @busy
```

What a command costs

The NIECE draws **four pixels a cycle**, and takes 4 cycles to start each command:

```
cycles = 4 + pixels / 4, rounded up
```

A command starts when it's written, or when the one before it finishes if that's later. All of its pixels appear together, on the cycle it finishes: until then the sheet is as it was. Nothing about this is left to chance. The cost never depends on what the pixels are.

For FILL and BLIT, the pixels counted are the ones of the destination that are on the page. What hangs over the edge is cut off and costs nothing, and a command wholly off the page costs its 4 cycles. See-through pixels and columns the depth line turns away (below) still count: the chip has to look at them.

	Pixels	Cycles
Fill the whole page	16,000	4,004: a fifth of a frame
A 16 × 16 picture	256	68
The same, drawn 64 × 64	4,096	1,028
A column 2 wide and 100 high	200	54
A frame at 50 a second has	79,872	19,968

So the page can be drawn five times over in a frame, and storing the dozen registers of a command takes the processor about as long as the NIECE takes to draw a small picture. The queue is rarely more than a few commands long unless you ask for something big.

Where: the destination

Address	Register	Read and write
`$C108-$C109`	NDX	The left edge, low byte first: -32,768 to 32,767.
`$C10A-$C10B`	NDY	The top edge, the same way.
`$C10C-$C10D`	NDW	The width, 0 to 65,535.
`$C10E-$C10F`	NDH	The height, 0 to 65,535.
`$C115`	NCOLOUR	Bits 0-3: the colour FILL and LINE draw in, 0-15. Bits 4-7: FILL's second colour, for a pattern.
`$C11D`	NPATTERN	FILL's pattern. 0 at reset: all one colour.
`$C116`	NFLAGS	See below.

The destination is a rectangle on the page that's being drawn on: (0, 0) is the page's top left pixel and (159, 99) its bottom right, whichever page it is. The rectangle can start off the page (NDX and NDY can be negative) and be far bigger than it; only the part on the page is drawn. All registers are 0 at reset, and they keep their values from one command to the next, so a program only stores the ones that change.

FILL (1) draws the rectangle in NCOLOUR. A width or height of 0 draws nothing.

With NPATTERN not 0, FILL mixes two colours. The pattern is four pixels wide and two lines high: bits 7 to 4 are the four pixels of the even lines, left to right, and bits 3 to 0 those of the odd lines. A 1 is drawn in the second colour, NCOLOUR's high four bits, and a 0 in the first. It repeats across and down from the top left corner of the page (or of the sheet, with SHEET), wherever the rectangle is, so two fills side by side join up without a seam. `%01011010` is a chequerboard, which from a chair's distance is a seventeenth colour; `%10000000` is one pixel in eight. A patterned FILL costs what a plain one does.

```
LDA #14 << 4 | 6    ; light blue on blue
STA NCOLOUR
LDA #%01011010      ; every other pixel, the lines staggered
STA NPATTERN
LDA #FILL
STA NGO
```

The short way in. Most things on a 160 × 100 page are within a byte of its corner, and storing two bytes for each of four numbers is half of what a small picture costs to ask for. Four more registers set both bytes with one store:

Address	Register	Write
`$C119`	NDXS	NDX, from a signed byte: -128 to 127. The high byte becomes 0 or `$FF` to match.
`$C11A`	NDYS	NDY, the same way.
`$C11B`	NDWS	NDW, 0 to 255: the high byte becomes 0.
`$C11C`	NDHS	NDH, the same way.

They're the same four numbers, not new ones: NDX reads what NDXS set, and each short register reads its number's low byte. So a picture half off the left edge is `LDA #-6`, `STA NDXS`, and nothing is left in a high byte for the next command to trip over.

LINE (3) draws a line in NCOLOUR from (NDX, NDY) to (NDX + NDW, NDY + NDH), both ends included. Here NDW and NDH are signed, so a line can go up and to the left: -30 is `$E2`, `$FF`. The line has one pixel for each step along its longer side, each at the nearest place on the other (a half rounds to the right, or down). Its cost is its whole length, longer side plus one, in pixels, on the page or not.

FLIP (4) makes the two pages change places: the one drawn on is shown, and the next commands draw on the other. It **waits for the beam to leave the picture** (the start of line 251, where FRAME goes up: chapter 9), and the queue waits behind it. So a FLIP never tears the picture, and a program that draws, FLIPs and waits for the queue to empty is in step with the screen without counting a line. It draws nothing and costs nothing but the wait.

```
; The NIECE's first picture: a sky, a field, a sun and a path.
NCTRL   = $C100
NSTATUS = $C101
NGO     = $C103
NDX     = $C108
NDY     = $C10A
NDW     = $C10C
NDH     = $C10E
NCOLOUR = $C115
NIECE   = $C14F
CLS     = $FF00
FILL    = 1
LINE    = 3
FLIP    = 4

        .org $0800
start:  LDA NIECE
        CMP #$17
        BNE none            ; no NIECE: nothing to do
        JSR CLS             ; spaces: the whole page shows through
        LDX #0
@box:   LDA boxes,X         ; five bytes a box: x, y, width, height, colour
        STA NDX             ; (the high bytes are 0 since reset)
        LDA boxes+1,X
        STA NDY
        LDA boxes+2,X
        STA NDW
        LDA boxes+3,X
        STA NDH
        LDA boxes+4,X
        STA NCOLOUR
        LDA #FILL
        STA NGO             ; into the queue, with the registers as they are
        TXA
        CLC
        ADC #5
        TAX
        CPX #15
        BNE @box
        LDA #0              ; the path: from (0, 99) to (159, 60), in white
        STA NDX
        LDA #99
        STA NDY
        LDA #159
        STA NDW
        LDA #<[-39]
```

```
        STA NDH
        LDA #>[-39]
        STA NDH+1
        LDA #1
        STA NCOLOUR
        LDA #LINE
        STA NGO
        LDA #FLIP           ; when the beam has left the picture
        STA NGO
@busy:  BIT NSTATUS
        BMI @busy
        LDA #1
        STA NCTRL           ; show the page
none:   HLT

boxes:  .byte 0, 0, 160, 60, 14         ; the sky, light blue
        .byte 0, 60, 160, 40, 5         ; the field, green
        .byte 120, 10, 16, 16, 7        ; the sun, yellow
```

The three boxes are 16,000 and 256 pixels: 4,076 cycles of drawing, which the NIECE does while the program is still storing the line's registers and then waiting.

From where: the source

BLIT (2) copies a rectangle of the sheet, the **source**, onto the destination rectangle, and makes it fit.

Address	Register	Read and write
`$C110`	NSX	The source's left edge on the sheet, 0-255.
`$C111-$C112`	NSY	Its top edge, 0-511 (nine bits).
`$C113`	NSW	Its width, 1-255.
`$C114`	NSH	Its height, 1-255.
`$C107`	NKEY	The see-through colour, 0-15. 0 at reset.

If the destination is the same size as the source, it's a plain copy. If it's wider, narrower, taller or shorter, the picture is stretched or squeezed to fit, each way on its own. Pixel (*i*, *j*) of the destination, counting from its top left corner, takes the source's pixel

```
( NSX + i * NSW / NDW ,  NSY + j * NSH / NDH )      divisions rounded down
```

so every pixel of a picture drawn at twice its size becomes 2 × 2, and one drawn at half its size keeps every other pixel. This holds for the pixels that are on the page even when most of the destination isn't: a 64 × 64 picture drawn 600 high shows the right slice of its middle. A source that runs off the sheet's edge carries on from the opposite edge. A width or height of 0, source or destination, draws nothing.

The source can be anywhere on the sheet, the pages included. The destination doesn't carry on round an edge the way a source does: what's off the page, or with SHEET off the sheet, is cut off.

NFLAGS says how:

Bit	Name	
0	KEY	Source pixels of the colour in NKEY aren't drawn: what was there shows through. BLIT only.
1	FLIPX	The source is mirrored left to right. BLIT only.
2	FLIPY	The source is upside down. BLIT only.
3	ZTEST	The depth line decides which columns are drawn.
4	ZWRITE	The columns drawn are written into the depth line.
5	SHEET	Draw on the sheet itself instead of the page: the destination's (0, 0) is the sheet's, and all 256 × 512 of it can be drawn on.
6-7		Which **ink table** the colours go through, 0-3. BLIT only.

Loading the sheet

The processor reaches the sheet's bytes through a window, one at a time:

Address	Register	Read and write
`$C104-$C105`	NVADDR	A byte of the sheet, 0-65,535: line × 128 + column / 2.
`$C106`	NVDATA	The byte at NVADDR. Reading it or writing it moves NVADDR on by one (after `$FFFF` comes 0).

NVDATA doesn't go through the queue: a byte written is in the sheet at once. So it can get there before a command that was asked for first: FILL a place and then write bytes into it, and the FILL lands on top of them when its turn comes. Wait for the queue to empty before writing where a command is going to draw.

```
; A picture on the sheet, drawn at three sizes.
NCTRL   = $C100
NSTATUS = $C101
NGO     = $C103
NVADDR  = $C104
NVDATA  = $C106
NDX     = $C108
NDY     = $C10A
NDW     = $C10C
NDH     = $C10E
NSX     = $C110
NSY     = $C111
NSW     = $C113
NSH     = $C114
NCOLOUR = $C115
NFLAGS  = $C116
CLS     = $FF00
FILL    = 1
BLIT    = 2
FLIP    = 4
KEY     = 1
ADDR    = $10               ; where the next line of the picture goes
FACEY   = 200               ; the picture's place on the sheet: (0, 200)

        .org $0800
start:  JSR CLS
        LDA #<[FACEY * 128]
        STA ADDR
        LDA #>[FACEY * 128]
        STA ADDR+1
        LDX #0
@row:   LDA ADDR            ; eight lines of four bytes
        STA NVADDR
        LDA ADDR+1
        STA NVADDR+1
        LDY #4
@byte:  LDA face,X
        STA NVDATA          ; and NVADDR moves on
        INX
        DEY
        BNE @byte
        CLC
        LDA ADDR
        ADC #128            ; the sheet's next line
        STA ADDR
        BCC @same
```

```
        INC ADDR+1
@same:  CPX #32
        BNE @row

        LDA #160            ; the page blue all over
        STA NDW
        LDA #100
        STA NDH
        LDA #6
        STA NCOLOUR
        LDA #FILL
        STA NGO

        LDA #FACEY          ; the source: 8 x 8 at (0, 200)
        STA NSY
        LDA #8
        STA NSW
        STA NSH
        LDA #KEY            ; colour 0 is see-through
        STA NFLAGS
        LDX #0
@face:  LDA where,X         ; three bytes a face: x, y, size
        STA NDX
        LDA where+1,X
        STA NDY
        LDA where+2,X
        STA NDW
        STA NDH
        LDA #BLIT
        STA NGO
        INX
        INX
        INX
        CPX #9
        BNE @face
        LDA #FLIP
        STA NGO
@busy:  BIT NSTATUS
        BMI @busy
        LDA #1
        STA NCTRL
        HLT

where:  .byte 4, 46, 8          ; as it is
        .byte 20, 38, 24        ; three times the size
        .byte 60, 10, 80        ; ten times
```

```
face:   .byte $00, $77, $77, $00
        .byte $07, $77, $77, $70
        .byte $77, $97, $79, $77
        .byte $77, $77, $77, $77
        .byte $79, $77, $77, $97
        .byte $77, $99, $99, $77
        .byte $07, $77, $77, $70
        .byte $00, $77, $77, $00
```

A digit of each byte is a pixel, so the picture can be read in the listing: 7 is yellow, 9 brown, and 0 is the see-through colour, where the blue stays.

Moving things. Nothing drawn on a page has to be rubbed out. Each frame, draw the background again (one FILL, or one BLIT of a picture kept on the sheet), draw everything where it is now, and FLIP. A thing moves because it was drawn somewhere else this time.

Scrolling is a BLIT whose source moves: keep a picture wider than the page on the sheet, and copy the 160 columns that start at NSX.

Ink tables

The NIECE has four **ink tables** of sixteen colours each. BLIT looks every source pixel up in one of them, chosen by bits 6 and 7 of NFLAGS, and draws the colour it finds there. At reset every table holds 0, 1, 2 ... 15, so every colour comes out as itself.

Address	Register	Read and write
`$C118`	NTABLE	Which table shows at NINK, 0-3.
`$C120-$C12F`	NINK	The table: what each source colour, 0-15, is drawn as.

Fill table 1 with darker colours and the same picture can be drawn in daylight or in shadow. Fill one with white and a thing can flash when it's hit. Give two tables different colours for 2 and 10 and one picture of a ship is two fleets.

KEY looks at the source's colour, before the table: the see-through colour stays see-through whatever the table says, and a table can still draw that colour from another one, so a picture keyed on 0 can have black in it. And a command takes a copy of its table with it into the queue, like the registers: change the table straight after NGO and the next command can have different colours again, so four tables are as many palettes as you care to store.

The depth line

The NIECE keeps one more thing: a number from 0 to 255 for each column of the page, the **depth line**. It says how far away the thing drawn in that column is. Small is near.

Address	Register	Read and write
`$C117`	NDEPTH	This command's depth, 0-255.

With **ZWRITE** set, every column a command draws in takes NDEPTH as its depth. With **ZTEST** set, a command draws only in the columns where NDEPTH is *less* than the column's depth: where it's nearer than what's there. With both, it draws where it's nearer and those columns take its depth. Both work on whole columns, and on FILL, BLIT and LINE alike.

At reset every column is 255, and a FILL of the whole page with ZWRITE and NDEPTH 255 makes it so again.

This is what a picture of a room wants. Draw each wall as columns with ZWRITE, each with its distance. Then draw a thing in the room as *one* BLIT with ZTEST and its own distance, and the NIECE leaves out the columns where a wall is nearer. A barrel half behind a corner is half drawn, and the program never works out which half.

The arithmetic unit

Address	Register	
`$C130-$C131`	NMULA	A number, -32,768 to 32,767, low byte first. Reads as written.
`$C132-$C133`	NMULB	Another.
`$C134-$C137`	NPROD	Read only: NMULA × NMULB, four bytes, low byte first, signed.
`$C138-$C13B`	NDIVN	A number, 0 to 4,294,967,295: four bytes, low byte first. Reads as written.
`$C13C-$C13D`	NDIVD	What to divide it by, 0 to 65,535.
`$C140-$C143`	NQUOT	Read only: NDIVN / NDIVD, rounded down, four bytes.
`$C144-$C145`	NREM	Read only: the remainder.

There's nothing to wait for and no command to give: NPROD, NQUOT and NREM are always the answer for what the other registers hold now. Dividing by 0 gives every bit set: NQUOT `$FFFFFFFF` and NREM `$FFFF`.

The multiplier is signed and the divider isn't. Chapter 7's multiply works a bit at a time, round a loop; this takes the stores and the loads:

```
        LDA speed           ; 16 bits each
        STA NMULA
        LDA speed+1
        STA NMULA+1
        LDA cosine
        STA NMULB
        LDA cosine+1
        STA NMULB+1
        LDA NPROD+1         ; the product's middle bytes: speed * cosine / 256
        STA dx
        LDA NPROD+2
        STA dx+1
```

Numbers with a fraction work as they are. If 256 stands for 1.0 in both, the product's bytes 1 and 2 are the answer on the same scale.

Three cassettes, and what 1 MHz makes of them

Three programs for the NIECE are in the Machine Room with their source. All three want the 64K expansion as well, for their pictures and tables.

CLOUDBERRY is a platformer that scrolls: two levels, each nearly thirteen screens long, a pixel at a time. The stretch of level you can see is kept on the sheet as a ring 256 pixels wide, and the page's background is one BLIT a frame whose source starts wherever the camera is: a source that runs off the sheet's edge carries on from the other, so the ring is never copied. When the camera crosses into a new column of tiles, twelve small BLITs draw that column into the ring. Three more layers move behind it at an eighth, a quarter and a half of the speed, the sky is patterned FILLs, and everything that moves is a BLIT. A three-voice tune plays on the AUNT all the while. The processor's worst frame is about 8,100 cycles of 19,968 and the NIECE's about 12,500 of drawing: 50 frames a second, with most of both to spare.

FELLRIDER is a moped over the fells, and its sums are real ones: a body with a place, a speed, a tilt and a spin, two wheels each on its own spring and damper, tyres that grip as hard as they're pressed down, and an engine whose push on the back wheel pushes back on the body, so opening the throttle lifts the front. That's 46 multiplications a frame, all of them the arithmetic unit's. The NIECE can't turn a picture, so the program doesn't ask it to: the wheels are round, the fork and the swingarm are LINEs from the hubs, and only the body and its rider are kept at 32 tilts, made on the sheet when it starts, with FLIPX and FLIPY together for the other half of the circle. The ground is a ring on the sheet, like CLOUDBERRY's. The processor's worst frame is about 13,000 cycles of 19,968: 50 frames a second.

FROSTHALL is a walk round a hall in three dimensions. The NIECE knows nothing of that: the program sends a ray out through the columns of the picture, the arithmetic unit does the six or seven divisions and multiplications a ray needs, and each column of wall is one BLIT, a pixel's width of a picture drawn as tall as the wall is near. The things in the hall are one BLIT each, and the depth line hides what's behind a wall. A ray costs the processor 800 or 900 cycles, so 80 of them a picture is 12 to 18 pictures a second; with a ray for every other column, and the columns between filled in where both neighbours met the same flat wall, it's 16 to 26, and that's how it starts. The NIECE itself is idle most of the time: the processor is what there isn't enough of.

They read better than these paragraphs: `games/niece/cloudberry.s`, `games/niece/fellrider.s` and `games/niece/frosthall.s`.

What it doesn't do

It doesn't know what a wall is, or a ship, or a floor. It has no sprites that stay where they're put, no collisions and no interrupts of its own (and BLIT has no pattern: a picture is its own): FRAME and the VBLANK interrupt (chapters 9 and 10) are the computer's, as before. It copies rectangles and it does sums. What they add up to is the program's business.

16. The monitor

The monitor is a small program in the ROM for looking at memory as it is: bytes in hex, and machine language listed back as instructions. It's there on every machine. From BASIC:

```
MON
*_
```

The `*` is its prompt, and it reads a line the way BASIC does, with the same screen editor. Numbers are hexadecimal, always, with no `$`: one to four digits, capitals or small letters. `Q` goes back to `READY.`, and nothing in memory has changed unless you changed it: the BASIC program is as it was.

Looking at memory

An address shows the byte there. Two addresses with a dot between them show everything from the first to the second, eight bytes to a row, each row starting with its address:

```
*FF00
FF00- 06
*0800.0817
0800- 0A 0A 00 80 20 22 48 49
0808- 22 00 08 14 00 84 20 31
0810- 30 00 00 00 00 00 00 00
```

The first is the jump table's first byte: `$06`, a `JMP`. The rest is a BASIC program as BASIC keeps it (chapter 2): a line of ten bytes, number 10, where `$80` is `PRINT`; then line 20, where `$84` is `GOTO`.

RETURN on an empty line shows the next eight, and again, and again: walk through memory with one key. A row always ends at an address ending in 7 or F, so the rows line up whatever address you started from. Several addresses on a line, with spaces between, show one byte each. STOP ends a long one.

Changing memory

An address, a colon, and bytes: they're stored from that address on.

```
*3000: 28 02 33 20 C0 03
*3000.3005
3000- 28 02 33 20 C0 03
```

A colon at the start of a line carries on from where the last store ended, so a long routine goes in a row at a time:

```
*3000: 28 02
*: 33 20 C0
*: 03
```

Storing to ROM, or to memory the machine doesn't have, does nothing, as always. Storing to the I/O addresses does what a POKE there would: C020: 2 turns the border red.

Listing instructions

L after an address lists twenty instructions from there, each with its address, its bytes and what they mean:

```
*3000L
3000  28 02     LDA #$02
3002  33 20 C0  STA $C020
3005  03        RTS
3006  00        BRK
```

L alone lists the next twenty. A branch shows the address it goes to, not the distance. A byte that isn't an instruction this book describes (Appendix C) lists as ???, one byte long; machine language listed from the wrong starting byte is nonsense for a few lines and then, usually, falls into step.

This is how to read the ROM. E000L is where it starts, and FF00L is the jump table (chapter 13).

Running a routine

G after an address calls the routine there, as SYS would: when it ends with RTS, the prompt is back. G alone goes to the address last given.

```
*3000G
```

The border turns red, and the * comes back. A routine that never returns never gives the prompt back either: STOP and RESTORE (chapter 10) get BASIC back, and MON again finds memory as the routine left it.

A first program

Four instructions, eight bytes: every screen code there is, put straight into screen memory (chapter 8), round and round.

```
*3000: 12 34 00 04 16 06 00 30
*3000L
3000  12        TXA
3001  34 00 04  STA $0400,X
3004  16        INX
3005  06 00 30  JMP $3000
*3000G
```

The top of the screen fills with the character set: all 256 codes, the second half as the first reversed. This routine never returns, so the prompt doesn't come back: STOP and RESTORE end it. Then MON again, 3002: 90 05, and 3000G puts them ten rows down.

To see the second half as the pictures it holds in the ROM, switch the screen to 256 glyphs first (chapter 8): `C000: 1` before the `G`. STOP and RESTORE switch it back.

All of it

You type	The monitor
`1000`	Shows the byte at `$1000`.
`1000.101F`	Shows `$1000` to `$101F`.
RETURN alone	Shows the next eight bytes.
`1000: A9 01`	Stores bytes from `$1000`.
`: 60`	Stores more, after the last.
`1000L`	Lists twenty instructions from `$1000`.
`L`	Lists the next twenty.
`1000G`	Calls the routine at `$1000`. `G` alone: at the address last given.
`Q`	Goes back to BASIC.

Anything else gets a `?` and the prompt again. The address "last given" is the one `:` stores at and `G` goes to; showing and listing keep a place of their own, which moves on as they go.

From machine language, the monitor is `JMP $FF24` (chapter 13). It uses `$D0–$DF` in the fast page, which BASIC only needs while it's drawing or loading, and the line you type is in BASIC's line buffer. It doesn't touch the interrupt vectors: a `BRK` is still yours (chapter 10).

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.

The first half, `$00–$7F`: what the screen shows as it's switched on.

	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	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	◆

- `$00–$07`: bit 0 lights the top-left quarter of the cell, bit 1 the top-right and bit 2 the bottom-left. `$0F` is all four. The seven with the bottom-right quarter are these reversed (as the machine is switched on, pattern `n` from 8 to 14 is code `n` exclusive-or `$8F`), and are in the second half.
- `$08–$0E` fill the bottom 1 to 7 eighths of the cell.
- `$10–$1A`: the lines run through the middle of the cell, two pixels thick, and reach its edges, so neighbouring cells join into boxes.
- `$20–$7E` are ASCII: a character's screen code is its ASCII code. `$7F` is a diamond.
- `$80–$FF`, as the machine is switched on: 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 8).

The second half, `$80–$FF` **in 256-glyph mode** (VMODE bit 0, chapter 8): no reverse video and no blinking, and these instead.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	▗	▚	▐	▜	▃	▙	▟	|	▌	▐	▖	▗	▘	▝	‾	|
9	♠	♥	♦	♣	•	○	←	↑	→	↓	◢	◣	◤	◥	☺	♪
A	`$08–$0E` reversed							row 8, reversed								
B	row 9, reversed															
C	=	‖	╔	╗	╚	╝	╠	╣	╦	╩	╬	◜	◝	◟	◞	squares
D	stripes across	stripes down	dark shade	◜	◝	◟	◞	Å	Ä	Ö	å	ä	ö	Ø	ø	Æ
E	æ	É	é	Ü	ü	ß	ñ	£	¥	§	°	±	×	÷	π	µ
F	½	¼	¾	¿	¡	«	»	ship	alien	man	tree	house	key	star	bomb	skull

- `$80–$86` are the quarter blocks with the bottom-right quarter lit: pattern `n`, 8 to 14, is at `n + $78`. `$87–$8D` fill the left 1 to 7 eighths of the cell. `$8E` is the top eighth and `$8F` the right eighth. `$A0–$A6` are the bottom eighths of `$08–$0E` reversed, and `$A7–$AF` these eighths reversed: `$A0` is all but the bottom eighth.

- `$C0–$CA`: double lines, in the order of the single ones at `$10–$1A`, thin and on the same two pixels in every cell, so they join.
- `$D3–$D6`: the four quarters of a ball two cells across: top-left, top-right, bottom-left, bottom-right.

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`
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.

Text and the bitmap are drawn in white on BACKGROUND. The NIECE (chapter 15) draws in all sixteen at once: each pixel of its page is one of these numbers.

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 and `$0316–$0317` the NMI's. 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–$D7FF`	2K	**UNSCII character ROM**, 256 glyphs: the first half of the 4K ROM. Read-only.
`$D800–$DFFF`	2K	**System ROM**, the 4K ROM's second half: HELP's screen and the monitor. Read-only.
`$E000–$FFFF`	8K	**System ROM**, the 8K ROM: BASIC, the screen, tape, the jump table and the vectors. Read-only.

I/O registers

Address	Name	Read	Write	Chapter
`$C000`	VMODE	Video mode	Bit 0: 256-glyph character set. Bit 1: bitmap. Bit 2: mixed (bitmap above, the last TEXTROWS text rows below). Bit 3: steady cursor (`$A0` doesn't blink). All 0 at reset	8
`$C001`	RASTER	Current raster line, bits 0-7; latches bit 8 for RASTER_HI		9
`$C002`	RASTER_HI	Bit 0: raster line bit 8, as latched by the last read of RASTER		9
`$C003`	RASTER_CMP	Compare line	Compare line (0-255)	10
`$C004`	IRQ_ENABLE	Enabled sources	Bit 0 raster, bit 1 vertical blank	10
`$C005`	IRQ_STATUS	Sources that have fired	1 bits acknowledge	10
`$C006`	FRAME	Frame counter		9
`$C010`	KBD	Last key; bit 7 set while not taken		11

Address	Name	Read	Write	Chapter
$C011	KBD_STROBE	As KBD (bit 7 included), then clears bit 7	Clears bit 7	11
$C012	HELD	The key down now (the latest pressed), or 0		11
$C013	CASE	Bit 0	Bit 0: letters arrive as typed, small unless shifted. 0 at reset: capitals	11
$C020	BORDER	Border colour	Border colour, 0-15	8
$C021	BACKGROUND	Background colour	Background colour, 0-15	8
$C022	CHARSET	Character set page	Character set page; $D0 at reset	8
$C023	BITMAP	Bitmap page	Bitmap page; $20 at reset	8
$C024	TEXTROWS	Text rows under the bitmap in mixed mode	0-25; 4 at reset	8
$C030	BEEPER		Any value flips SPEAKER bit 1	12
$C031	SPEAKER	Bit 0 gate, bit 1 data, bit 5 the timer's output	Bit 0: timer gate. Bit 1: speaker data; BEEPER flips it	12
$C032	TIMER_LO	As written	Timer divisor, low byte	12
$C033	TIMER_HI	As written	Timer divisor, high byte: loads the divisor (0 is 65,536)	12
$C034	NOISE	Pseudo-random byte from the cycle count		12
$C040	TAPE	Bit 0: PLAY down. Bit 1: tape in. Bit 2: motor on. Bit 3: RECORD down. Bit 4: room for a byte (recording). Bit 6: overrun (playing). Bit 7: byte ready (playing)	Bit 2: motor on	14
$C041	TAPEDATA	The byte off the tape; reading takes it (clears TAPE bits 6 and 7)	With RECORD down: the next byte to record (clears TAPE bit 4 until the deck takes it)	14
$C050	SHIFT_IN		Pushes a byte into the shifter's window	8
$C051	SHIFT_AMOUNT	Shift, 0-7	Shift (bits 0-2)	8
$C052	SHIFT_OUT	The window shifted right by SHIFT_AMOUNT, low byte		8
$C053	SHIFT_FLIP		Pushes a byte with its bits reversed	8
$C060–$C068	KEYS	Keyboard matrix, rows 0-8: 1 while a key is down		11

Address	Name	Read	Write	Chapter
$C0C0-$C0C7		Voice 1 of the AUNT: FREQ_LO, FREQ_HI, WIDTH, WAVE, CONTROL, ATTACK_DECAY, SUSTAIN_RELEASE, TRIM. As written	As read	12
$C0C8-$C0CF		Voice 2, the same		12
$C0D0-$C0D7		Voice 3, the same		12
$C0D8	CUTOFF	The filter's corner	0-255	12
$C0D9	FILTER	As written	Bits 0-2: voice 1, 2, 3 through the filter. Bits 4-7: resonance	12
$C0DA	VOLUME	As written	Bits 0-3: volume. Bit 4 low-pass, bit 5 band-pass, bit 6 high-pass. Bit 7: voice 3 silent	12
$C0DC	VOICE3	Voice 3's waveform now, top 8 bits		12
$C0DD	ENV3	Voice 3's envelope now		12
$C0DF	SOUND	$16: the AUNT		12
$C0E0-$C0FF	WAVETABLE	As written	The wave table: 32 steps	12
$C100	NCTRL	Bit 0: the NIECE's page shown. The NIECE,	Bit 0	15
$C101	NSTATUS	Bit 7 busy, bit 6 queue full, bit 5 a command lost, bit 0 the page shown	Any value: clears bit 5	15
$C102	NQUEUE	Commands in the queue, 0-64		15
$C103	NGO		A command into the queue: 1 FILL, 2 BLIT, 3 LINE, 4 FLIP	15
$C104-$C105	NVADDR	As written	A byte of the sheet	15
$C106	NVDATA	The sheet's byte at NVADDR; NVADDR moves on	The same	15
$C107	NKEY	As written	The see-through colour, 0-15	15
$C108-$C109	NDX	As written	The destination's left edge, signed	15
$C10A-$C10B	NDY	As written	The destination's top edge, signed	15
$C10C-$C10D	NDW	As written	The destination's width (LINE: how far across, signed)	15
$C10E-$C10F	NDH	As written	The destination's height (LINE: how far down, signed)	15
$C110	NSX	As written	The source's left edge, 0-255	15

Address	Name	Read	Write	Chapter
$C111–$C112	NSY	As written	The source's top edge, 0-511	15
$C113	NSW	As written	The source's width	15
$C114	NSH	As written	The source's height	15
$C115	NCOLOUR	As written	Bits 0-3: FILL's and LINE's colour. Bits 4-7: FILL's second colour	15
$C116	NFLAGS	As written	Bit 0 KEY, 1 FLIPX, 2 FLIPY, 3 ZTEST, 4 ZWRITE, 5 SHEET, 6-7 the ink table	15
$C117	NDEPTH	As written	The command's depth, 0-255	15
$C118	NTABLE	As written	The ink table at NINK, 0-3	15
$C119	NDXS	NDX's low byte	NDX from a signed byte	15
$C11A	NDYS	NDY's low byte	NDY from a signed byte	15
$C11B	NDWS	NDW's low byte	NDW, 0-255	15
$C11C	NDHS	NDH's low byte	NDH, 0-255	15
$C11D	NPATTERN	As written	FILL's pattern: 4 pixels by 2 lines, a 1 the second colour	15
$C120–$C12F	NINK	As written	The ink table: sixteen colours	15
$C130–$C131	NMULA	As written	A signed 16-bit number	15
$C132–$C133	NMULB	As written	Another	15
$C134–$C137	NPROD	NMULA × NMULB, 32 bits		15
$C138–$C13B	NDIVN	As written	A 32-bit number	15
$C13C–$C13D	NDIVD	As written	A 16-bit number	15
$C140–$C143	NQUOT	NDIVN / NDIVD		15
$C144–$C145	NREM	The remainder		15
$C14F	NIECE	$17: the NIECE is in		15
$C3F0–$C3FF		Reserved	Reserved	

Without the NIECE (chapter 15), every address in $C100–$C14F reads $FF and ignores writes, NIECE included. With it, the ones not listed here do the same. 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 (FRAME goes up and the VBLANK interrupt comes at line 251, as the beam leaves the picture)
Text area	lines 51-250
Tape	a byte every 1,664 cycles (12 a frame), played or recorded, 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 not instructions this book describes: as a rule, 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, or an NMI	-
`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
`LDA $nnnn,Y`	$2D	3	5	Load A	N Z

Syntax	Opcode	Bytes	Cycles	Operation	Flags
`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`	$4E	2	6	A AND M	N Z
`ORA ##$nn`	$50	2	2	A OR M	N Z

Syntax	Opcode	Bytes	Cycles	Operation	Flags
`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`	$83	3	4	Store Y	-
`CPX #$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

Syntax	Opcode	Bytes	Cycles	Operation	Flags
`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 machine code, gives BASIC's READY. with a BASIC program loaded, 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
$FF24	MON		Doesn't return: the monitor's * prompt (chapter 16). Its Q goes to READY	everything; $D0–$DF

The next free slot is $FF27. The table can grow up to $FFDF; the reset code starts at $FFE0.

The ROM doesn't use the timer or the shifter, and leaves SPEAKER at 0. Of the keyboard matrix it reads only row 8, at READY., to wait for the cursor keys to be let go. NOISE ($C034, chapter 12) is the hardware's own random byte: faster than RND, and the way to seed it.

Vectors

Address	Vector	ROM Rev A value
$FFFA–$FFFB	NMI	JMP ($0316)
$FFFC–$FFFD	Reset	$FFE0: start-up, cold or (after RESET) warm

Address	Vector	ROM Rev A value
$FFFE-$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 and SAVE use $D0-$D3. The monitor keeps its place in $D0-$DF
$DC-$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 screen 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, SAVE: the name, padded with spaces
$02F4	TNAMED	LOAD, SAVE: not 0 if a name was given
$02F5-$030C	THDR	LOAD: the header as read; SAVE: as it's recorded. From the version byte to the entry address
$030D	TCHK	LOAD, SAVE: the bytes so far, EORed
$030E	TOVR	LOAD: $40 if a byte was missed
$030F	TBORD	LOAD, SAVE: the border colour before
$0310-$0311	DATAP	READ: where the next DATA item starts. The high byte is 0 before the first READ, and after RESTORE, RUN and NEW
$0312-$0313	DATAL	READ: the line that item is on
$0314-$0315	IRQVEC	Interrupt handler address
$0316-$0317	NMIVEC	NMI handler address: the RESTORE key

Address	Name	Holds
$0318–$0319	WARMMK	$55 $AA once switched on: RESET is a warm start
$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
$03D8	RLCOL	The screen editor: the column the cursor was in when it began to wait
$03D9	RLROW	The screen editor: the row it was on, kept right when the screen scrolls
$03DA–$03F2	LINK	The screen editor: a byte for each row, 1 if the row carries on the line of the row above, 0 if a line starts on it
$03F3	RLMOV	The screen editor: 0 until a cursor key or INSERT is pressed; then RETURN takes the line off the screen

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 6.
`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`.
`can't paste '1+2' into a name`	Only a value that's a plain word or number can be pasted with `\`.

Message	Meaning
`unknown function 'foo'`	A name followed by `(` that isn't one of the functions in chapter 6.
`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, or in the Machine Room among the project's files).
`includes nest too deep at 'file.s' (does it include itself?)`	More than 16 includes inside each other.
`includes aren't available here`	The program was assembled on its own, without files to include from (a host that takes one source).
`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, or an NMI
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 6)

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