# Pocket ARM

## The Companion Guide

ARMv7 assembly language on iPhone, iPad and Android

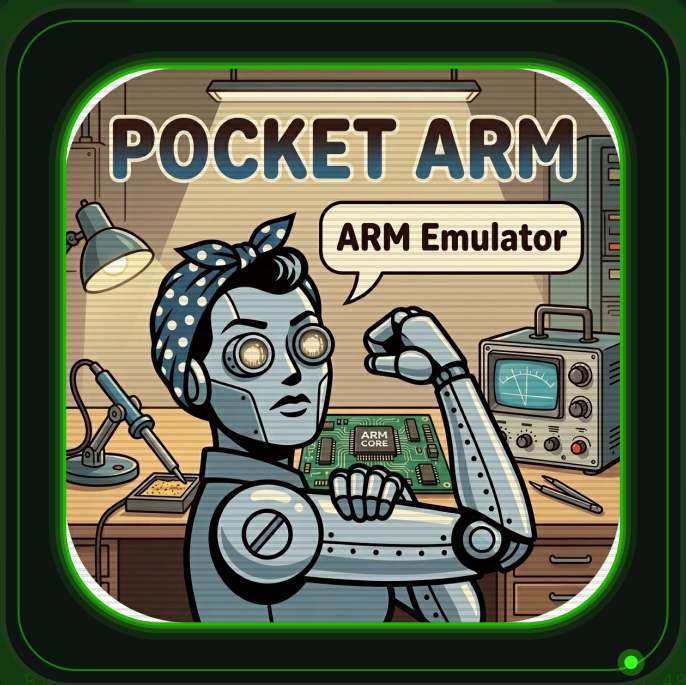
CPSR

N 0

Z 1

C 1

V 0




FREE EDITION

000C E0803001 ADD R3, R0, R1
0010 E1A00001 MOV R0, R1
0018 E2522001 SUBS R2, R2, #1
001C 1AFFFFFA BNE LOOP

2008 6F 63 6B 65 74 41 52 4D
2010 00 00 00 00 00 00 00 00
7FF8 2C 00 00 00 00 00 00 00

| R0 0x00000015 | R1 0x00000022 | SP 0x00008000 | PC 0x00000020 |
|---|---|---|---|
| REGISTERS | SOURCE | MEMORY | CONSOLE |




# Pocket ARM

## The Companion Guide

Learning ARM assembly language on a phone or tablet

For Pocket ARM 1.0.3 on iPhone and iPad
and Pocket ARM 1.0.3 on Android

*Pocket ARM: The Companion Guide*
Copyright © 2026 Pymble Software Pty Ltd. All rights reserved.

**Free edition.** This guide is free. You are welcome to copy and share it, unchanged and in full, with anyone it might help.

First edition, typeset November 21, 2026.

Pocket ARM is published by Pymble Software. iPhone, iPad and App Store are trademarks of Apple Inc. Android and Google Play are trademarks of Google LLC. Arm and ARMv7 are trademarks of Arm Limited (or its subsidiaries); they are used here only to describe the instruction set the app simulates, and no endorsement is implied.

The screenshots were taken from the shipping apps running on the iOS Simulator and the Android Emulator. The example programs may be copied and used freely.

# Contents


**1 Welcome to Pocket ARM** **1**
- 1.1 What the app does . . . 1
- 1.2 What the app does not do . . . 2
- 1.3 Who this guide is for . . . 2
- 1.4 Versions covered . . . 2
- 1.5 How to read this guide . . . 3

**2 Getting Started** **4**
- 2.1 Installing the app . . . 4
- 2.2 The first screen . . . 4
- 2.3 Your first program, in one minute . . . 5
- 2.4 Phones, tablets and turning sideways . . . 6
- 2.5 The More tab . . . 8
- 2.6 Language . . . 9

**3 ARM in Ten Minutes** **10**
- 3.1 A processor, a memory, and a list of instructions . . . 10
- 3.2 Numbers: decimal, hex and binary . . . 10
- 3.3 Registers . . . 11
- 3.4 Instructions . . . 11
- 3.5 Flags and conditions . . . 12
- 3.6 The stack . . . 12
- 3.7 Where Pocket ARM differs from a real ARM . . . 12

**4 The Registers Panel** **14**
- 4.1 The CPSR strip . . . 14
- 4.2 The register grid . . . 15
- 4.3 Hex, decimal and binary . . . 15
- 4.4 Changing registers . . . 16

**5 The Source Editor and the Assembler** **17**
- 5.1 The editor . . . 17
- 5.2 The shape of a line . . . 18
- 5.3 Upper and lower case . . . 19
- 5.4 Registers . . . 19
- 5.5 Numbers and characters . . . 19
- 5.6 Operand forms . . . 20

5.7 Big numbers and addresses: `LDR =`, `ADR`, `MOVW` and `MOVT` . . . 21
5.8 Directives and data . . . 21
5.9 Assembling, and reading errors . . . 22

**6 The Disassembly and Breakpoints** **24**
6.1 Reading a row . . . 24
6.2 Reading the encoding . . . 25
6.3 Breakpoints . . . 25
6.4 Moving between the editor and the disassembly . . . 26

**7 The Memory Panel** **27**
7.1 The memory map . . . 27
7.2 Reading the dump . . . 27
7.3 Seeing your program's code . . . 29
7.4 Watching the stack . . . 29
7.5 A bug in 1.0.3: the segment buttons lag on iPhone and iPad . . . 29

**8 The Console, Output and Input** **31**
8.1 The status bar . . . 31
8.2 Messages . . . 32
8.3 Supervisor calls: how a program prints . . . 32
8.4 Input . . . 34
8.5 Runtime errors . . . 35

**9 Running and Debugging** **37**
9.1 The toolbar . . . 37
9.2 Running, stepping and +N . . . 38
9.3 When a program finishes . . . 38
9.4 A debugging routine . . . 39
9.5 Common surprises . . . 40

**10 Saving, Loading and Moving Programs** **41**
10.1 The library . . . 41
10.2 Saving: NAME . . . 41
10.3 Opening: LIBRARY . . . 42
10.4 iPhone and iPad: moving files with the Files app . . . 43
10.5 Android: copy and paste . . . 44
10.6 Moving programs between an iPhone and an Android phone . . . 44

**11 The Example Programs** **45**
11.1 hello . . . 46
11.2 input . . . 46
11.3 fibonacci . . . 46
11.4 gcd . . . 47
11.5 factorial . . . 47
11.6 bit_counting . . . 47

11.7 new_instructions_demo . . . 47
11.8 string_length . . . 47
11.9 caesar_cipher . . . 48
11.10bubble_sort . . . 48
11.11binary_search . . . 48

**12 Exercises and Recipes** **50**
12.1 Add up 1 to 10 . . . 50
12.2 Count down . . . 50
12.3 A function . . . 51
12.4 The largest number in a table . . . 51
12.5 Shout . . . 52
12.6 Guess the number . . . 52
12.7 Give the examples real data . . . 53
12.8 Recipes . . . 53

**13 Troubleshooting and FAQ** **55**
13.1 Assembling . . . 55
13.2 Running . . . 55
13.3 Memory panel . . . 56
13.4 Files . . . 56
13.5 Other questions . . . 57

**A iPhone, iPad and Android Compared** **58**

**B Instruction Set Reference** **60**
B.1 Condition codes . . . 60
B.2 Data processing . . . 61
B.3 Shifts . . . 61
B.4 Multiply and divide . . . 62
B.5 Moving constants and addresses . . . 63
B.6 Branches . . . 63
B.7 Loads and stores . . . 64
B.8 Multiple loads and stores . . . 64
B.9 Other instructions . . . 65
B.10 Directives . . . 65
B.11 Not supported . . . 66

**C Quick Reference and Glossary** **67**
C.1 Quick reference . . . 67
C.2 Glossary . . . 68

**Index** **70**

CHAPTER 1

# Welcome to Pocket ARM

Pocket ARM is a small computer that lives inside your phone. It has sixteen registers, a status register full of flags, 64 KB of memory and a console, and it understands a large, useful part of the 32-bit ARM instruction set (ARMv7, the "A32" instructions). You type a program in ARM assembly language, tap **ASSEMBLE**, and then watch it run: one instruction at a time if you like, with every register, flag and byte of memory on screen as it changes.

Assembly language is the closest you can get to what a processor actually does. Every phone, tablet and most laptops sold today run on ARM processors, so the instructions you learn here are, with very few changes, the ones your own phone is executing as you read this. Pocket ARM takes away everything that usually makes this hard to learn: there is no compiler to install, no linker script, no operating system to talk to, and nothing you can break.

This guide explains every screen of the app on iPhone, iPad and Android, teaches enough ARM assembly language to write real programs, and walks through the example programs that come with the app.

## 1.1 What the app does

Pocket ARM is an **assembler** and a **simulator** in one:

- The **assembler** reads the text of your program, checks it, and turns each line into the 32-bit machine code word a real ARM processor would run. If something is wrong it tells you which line, and why.
- The **simulator** then runs that machine code on a pretend ARM processor, and shows you its state.

The screen is divided into five areas, which on a phone are five tabs along the bottom:

| Tab | What it shows | Chapter |
|---|---|---|
| **REGISTERS** | R0 to R15 and the CPSR flags, in hex, decimal or binary | 4 |
| **SOURCE** | The program editor, the disassembly with breakpoints, and the file buttons | 5, 6, 10 |
| **MEMORY** | A hex dump of the text (code), data and stack areas | 7 |

| Tab | What it shows | Chapter |
|---|---|---|
| **CONSOLE** | Messages from the app and output from your program, and the input bar | 8 |
| **MORE** | Rate, share, support, privacy and feedback | 2 |

Above the tabs is a toolbar with the buttons that drive the simulator: **ASSEMBLE**, **RUN**, **STOP**, **STEP**, **+N**, **RESET** and **CLEAR** (Chapter 9).

On an iPad, and on Android tablets, the four working areas are shown together in a four-pane layout, so you can watch registers, code, memory and console at once.

## 1.2 What the app does not do

- **It is a teaching simulator, not a full ARM.** It runs the 32-bit ARM instructions most programs and courses use (Appendix B lists them all). It does not run Thumb code, floating point, NEON, or system instructions, and it has no interrupts or operating system.
- **It does not run real programs from elsewhere.** It reads assembly language text, not executable files. Programs written for Linux or a microcontroller need their system calls changed to Pocket ARM's (Chapter 8) before they will run.
- **Memory is small.** There are 64 KB, laid out as code, data and stack (Chapter 7). That is plenty for learning.
- **It never connects to anything.** The simulator works entirely on your device. Nothing you type is sent anywhere.

## 1.3 Who this guide is for

You do not need to have programmed before, although it helps. If you are taking a computer architecture or embedded systems course, preparing for one, or simply curious about what happens underneath the apps you use, this guide is for you. Chapter 3 explains the handful of ideas everything else rests on (registers, memory, instructions and flags) in plain language, and Appendix C has a glossary.

If you already know ARM assembly, skim Chapter 3 and read Chapter 5 carefully: it describes exactly what Pocket ARM's assembler accepts, which is a little different from the GNU and Keil assemblers, and §3.7 lists the places where the simulator behaves differently from real hardware.

## 1.4 Versions covered

This edition describes:

| Platform | Version | Runs on |
|---|---|---|
| iPhone and iPad | Pocket ARM 1.0.3 | iOS / iPadOS 26.2 or later |
| Android | Pocket ARM 1.0.3 | Android 8.0 (API level 26) or later |

The two apps share a design, the same assembler and the same simulator, but they are written separately (in Swift for Apple devices and Kotlin for Android), so some details differ. The biggest difference is how you get program files in and out of the app (Chapter 10). Where the platforms differ the guide says so, and Appendix A collects every difference in one table. Every screenshot comes in a pair, with the iPhone on the left and Android on the right.

## 1.5 How to read this guide

- **Chapter 2** gets the app installed and takes you on a guided tour with your first program.
- **Chapter 3** is a ten-minute primer on the ARM processor.
- **Chapters 4 to 8** describe the Registers, Source, Disassembly, Memory and Console areas in detail.
- **Chapter 9** explains running, stepping and breakpoints: the art of debugging.
- **Chapter 10** covers saving, loading and moving program files, on iPhone and on Android.
- **Chapter 11** walks through each of the eleven example programs that come with the app.
- **Chapter 12** has exercises, from easy to challenging, with hints.
- **Chapter 13** is troubleshooting and frequently asked questions.
- **Appendices** compare the platforms, list every instruction, and give a quick reference and glossary.

Names of buttons, labels and messages are printed in **bold** exactly as they appear in the app. Program text, register names and values you type are printed in `code type`.

> **Tip.** You can follow almost all of this guide with the programs that come with the app. Open the **LIBRARY** on the **SOURCE** tab, pick one, and tap **ASSEMBLE**.

CHAPTER 2

# Getting Started

## 2.1 Installing the app

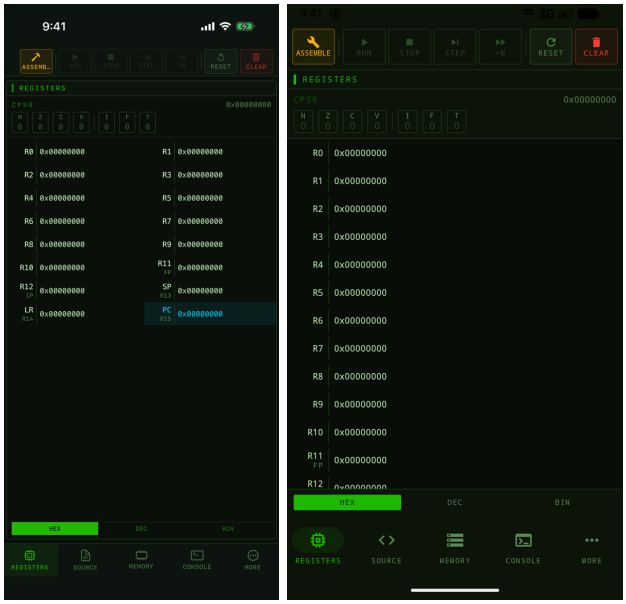
Pocket ARM is free.

- **iPhone and iPad:** search the App Store for *PocketARM* (one word, as it appears under the icon on your Home Screen). It needs iOS or iPadOS 26.2 or later.
- **Android:** search Google Play for *Pocket ARM* from Pymble Software. It needs Android 8.0 or later.

The app needs no account, no sign-in and no network connection. The first time it starts it copies eleven example programs into its file library (§10.1), ready to open.

## 2.2 The first screen

Pocket ARM opens on the **REGISTERS** tab with every register at zero, except the stack pointer **SP**, which starts at `0x00008000` (§7.1). The console holds one message, *PocketARM ready. Assemble and run ARMv7 code.*, and the editor already contains a short program that computes Fibonacci numbers.




*The first screen: iPhone (left) and Android (right).*

The layout is the same on both platforms:

- **The toolbar** across the top holds the seven simulator buttons. Buttons that can't be used at the moment are dimmed: **RUN**, **STEP** and **+N** stay dim until a program has been assembled, and **STOP** is lit only while a program is running.
- **The panel** in the middle shows the current tab. Every panel has a title bar with a short green bar at its left.
- **The tab bar** along the bottom switches between **REGISTERS**, **SOURCE**, **MEMORY**, **CONSOLE** and **MORE**. A red dot on **SOURCE** means the last assembly found errors; a green dot on **CONSOLE** means a program is running.

On iPhone the toolbar buttons have small labels, and the longest, **ASSEMBLE**, is shortened to **ASSEMB...** on narrower phones. Android shrinks the label to fit instead.

## 2.3 Your first program, in one minute

1. Tap the **SOURCE** tab. You'll see the editor with the Fibonacci program, line numbers down the left.
2. Tap **ASSEMBLE** in the toolbar. The console says *Assembled 10 instructions successfully.* and **RUN**, **STEP** and **+N** light up.
3. Tap **RUN**. The program runs in a fraction of a second.
4. Tap the **CONSOLE** tab. Near the bottom you'll see `SVC 0: R0 = 21 (0x15)` (on Android, `0x00000015`), which is the program printing its answer, then *SVC 1: Program exited.* The status bar reads **HALT: SVC CALL** and **STEPS: 45**: the program ran 45 instructions.




*The built-in Fibonacci program in the editor: iPhone (left) and Android (right). The two apps start with the same program; only some of the comments differ.*

That is the whole cycle: edit, assemble, run, look. Now try it slowly:

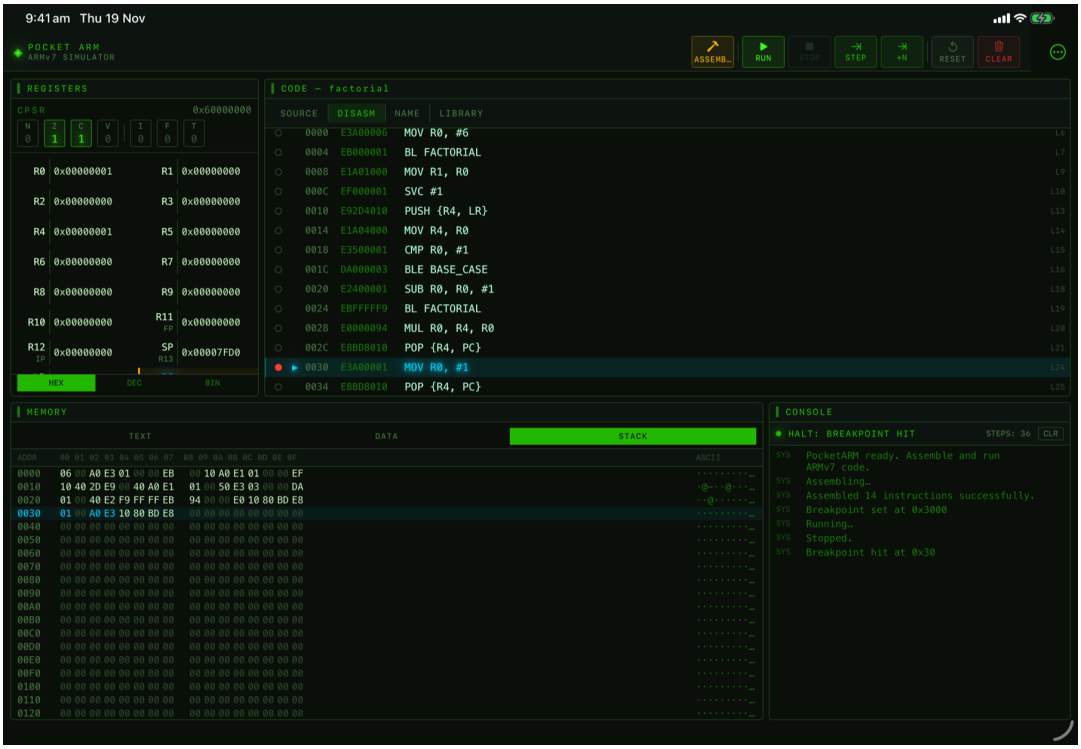
5. Tap **RESET**, then tap **STEP** a few times while watching the **REGISTERS** tab. Each tap runs one instruction. The register it changed glows amber, and **PC**, the program counter, moves on by 4 each time.

Chapter 9 explains every way of running a program.

> **Note.** The comment in the program says that R1 ends up holding *F(8) = 34*, but the program prints R0, which holds the previous number in the sequence, 21. Both are right: after eight trips round the loop R0 holds 21 and R1 holds 34. Step through it and watch.

## 2.4 Phones, tablets and turning sideways

On a phone held upright you see one area at a time and switch with the tabs. On an iPad, or an Android tablet, the app shows all four working areas at once:



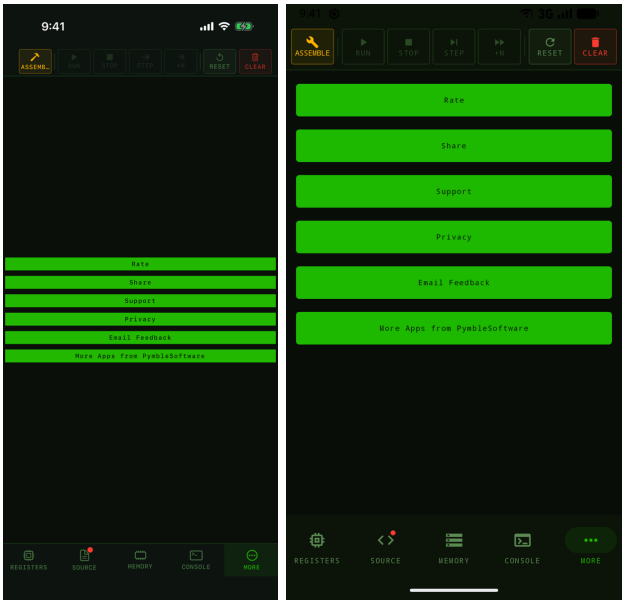

*The four-pane layout on iPad: Registers and Code above, Memory and Console below, with the toolbar across the top. Here the factorial example (§11.5) has stopped at a breakpoint deep in its recursion.*

The four-pane layout is used whenever the window is wide enough:

- **iPad:** always, in either orientation, and also in Split View when Pocket ARM has a wide enough share of the screen. The toolbar also shows the app's name, **POCKET ARM — ARMv7 SIMULATOR**.
- **iPhone:** iPhones always use the tab layout, except that the largest models switch to the four-pane layout when turned sideways.
- **Android:** any window at least 600 dp wide, which includes most phones when turned sideways, as well as tablets and foldables when open.

On iPad the **MORE** tab becomes a ··· button at the right-hand end of the toolbar, which opens the same buttons in a popover. On Android tablets, and on Android phones turned sideways, the four-pane layout has no **MORE** area at all; turn the phone upright to reach it.

## 2.5 The More tab




*The More tab: iPhone (left) and Android (right).*

| Button | What it does |
|---|---|
| **Rate** | Opens the app's page in the App Store or Google Play so you can leave a rating |
| **Share** | Opens your device's share sheet with a link to the app |
| **Support** | Opens the Pocket ARM page on pymblesoftware.com in your browser |
| **Privacy** | Opens Pymble Software's privacy policy in your browser |
| **Email Feedback** | Starts an email to the developer, with the app and your device model already in the subject |
| **More Apps from PymbleSoftware** | Lists Pymble Software's other apps in the store |

On iPhone and iPad, **Email Feedback** uses Apple's Mail app. If no Mail account is set up on the device, the button does nothing; write to `info@pymblesoftware.com` from your usual email app instead. On Android it offers a choice of email apps, or shows *No email app found* if there are none.

## 2.6 Language

Both apps are in English and Spanish, and follow the language of your device. To use Pocket ARM in Spanish without changing your whole device, use the per-app language setting: on iPhone, **Settings › Apps › PocketARM › Language**; on Android 13 or later, **Settings › System › Languages › App languages › Pocket ARM**.

Only the app's own words are translated. The instruction names, register names and the assembler's error messages are in English in every language, as they are in every real assembler.

CHAPTER 3

# ARM in Ten Minutes

This chapter explains the few ideas that everything in Pocket ARM is built on. If you have programmed in assembly language before, skim it, but do read §3.7.

## 3.1 A processor, a memory, and a list of instructions

A computer, at the bottom, is a processor connected to memory. Memory is a long row of numbered boxes called **bytes**, each holding a number from 0 to 255. The number on the box is its **address**. Pocket ARM's memory has 65,536 bytes, so the addresses run from `0x0000` to `0xFFFF`.

The processor reads **instructions** from memory and carries them out, one after another. Each ARM instruction is one 32-bit number (four bytes) that encodes something very simple: *add these two numbers*, *copy this value*, *load four bytes from that address*, *if the last result was zero, jump over there*. Everything a computer does is built out of millions of these tiny steps.

Nobody wants to write programs as 32-bit numbers, so we write them in **assembly language**, where each instruction is a short word with its operands:

```
        ADD  R3, R0, R1     @ R3 = R0 + R1
```

An **assembler** translates each line into its number. Pocket ARM shows you both: the line you wrote, and the number it became (§6.2).

## 3.2 Numbers: decimal, hex and binary

Inside the machine everything is binary: ones and zeros. Thirty-two binary digits (bits) are tiresome to read, so programmers write them in **hexadecimal** (base 16), where each digit stands for four bits and the digits run 0–9 then A–F. Hex numbers are written with `0x` in front.

| Decimal | Hex | Binary (8 bits) |
|---|---|---|
| 0 | `0x00` | `00000000` |
| 10 | `0x0A` | `00001010` |
| 42 | `0x2A` | `00101010` |
| 165 | `0xA5` | `10100101` |
| 255 | `0xFF` | `11111111` |

A 32-bit register can hold 0 to 4,294,967,295 (`0xFFFFFFFF`). The same bits can also stand for a **signed** number from −2,147,483,648 to 2,147,483,647, using **two's complement**: if the top bit is set the number is negative, and `0xFFFFFFFF` means −1. Pocket ARM's register panel can show values in hex, as signed decimal, or in binary (§4.3), so you can look at the same bits all three ways.

## 3.3 Registers

Memory is big but, for a processor, slow. So the processor has a handful of tiny, fast storage slots of its own called **registers**. ARM has sixteen, each 32 bits wide, named **R0** to **R15**. Almost every instruction works on registers: you load values into registers, do arithmetic on them, and store the results back to memory.

Three registers have special jobs, and special names:

| Register | Name | Job |
|---|---|---|
| R13 | **SP**, the stack pointer | The address of the top of the stack (§3.6) |
| R14 | **LR**, the link register | The return address when you call a function with `BL` |
| R15 | **PC**, the program counter | The address of the next instruction to run |

By convention R11 is also called **FP** (frame pointer) and R12 **IP** (intra-procedure scratch register). Pocket ARM accepts all these names, and shows the aliases under the register names on the Registers panel.

## 3.4 Instructions

ARM instructions fall into a few families. You will meet them all in the example programs:

- **Data processing:** `MOV`, `ADD`, `SUB`, `AND`, `ORR`, `EOR`, `CMP` and friends do arithmetic and logic on registers. Most take a destination register and two operands: `SUB R2, R2, #1` means *R2 = R2 − 1*. A `#` marks an **immediate**, a number written into the instruction itself.
- **Load and store:** `LDR` copies four bytes from memory into a register and `STR` copies a register to memory. Square brackets mean *the memory at this address*: `LDR R6, [R0, #8]` loads the word at address *R0 + 8*.
- **Branches:** `B label` jumps to a label; `BL label` jumps and remembers where it came from in LR, which is how you call a function; `BX LR` jumps back to the address in LR, which is how you return.
- **Supervisor call:** `SVC #n` asks the "operating system" for a service. In Pocket ARM, that is how a program prints and reads (Chapter 8).

## 3.5 Flags and conditions

Instructions such as `CMP R3, #0` don't store a result. Instead they set four **flags** in the **CPSR** (current program status register):

| Flag | Name | Set when the result… |
|---|---|---|
| **N** | Negative | …has its top bit set (is negative, read as signed) |
| **Z** | Zero | …is zero |
| **C** | Carry | …carried out of bit 31 (an addition overflowed as unsigned), or did *not* borrow (a subtraction) |
| **V** | Overflow | …overflowed as a signed number |

Other data-processing instructions set the flags only if you add **S** to their name: `SUBS R2, R2, #1` subtracts *and* sets the flags, so the next instruction can test whether R2 reached zero.

Almost every ARM instruction can then be made **conditional** by adding a two-letter condition: `BNE LOOP` branches only if Z is clear ("not equal"), `MOVEQ R0, #'E'` moves only if Z is set. The full list of conditions is in Appendix B. This is the heart of every loop and every `if` in assembly language:

```
LOOP:   ...
        SUBS R2, R2, #1     @ count down, setting Z when R2 reaches 0
        BNE  LOOP           @ go round again unless R2 is now 0
```

## 3.6 The stack

The **stack** is an area of memory used for saving registers, especially inside functions. It grows **downwards**: SP starts at `0x8000`, and `PUSH {R4, LR}` subtracts 8 from SP and stores R4 and LR in the eight bytes below the old top. `POP {R4, PC}` loads them back and adds 8 to SP; because the saved LR is loaded into PC, this also returns from the function. Chapter 7 shows how to watch the stack grow and shrink.

## 3.7 Where Pocket ARM differs from a real ARM

Pocket ARM runs ARM machine code faithfully enough for learning, but a few things are simplified. They matter only if you compare it with real hardware:

- **Reading PC gives the current instruction + 4, not + 8.** On a real ARM, an instruction that reads R15 sees its own address plus 8 (an effect of the original three-stage pipeline). In

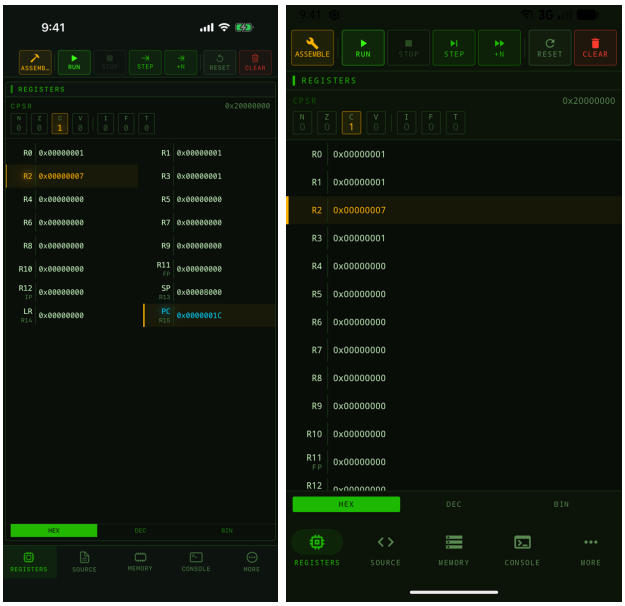
Pocket ARM it sees its address plus 4. Branches are assembled and executed consistently, so labels always work; only code that does arithmetic on PC directly sees the difference. Use labels, `ADR` or `LDR Rd, =label` instead.

- **Memory is 64 KB, little-endian, with no protection.** Any address from `0x0000` to `0xFFFF` can be read and written, including the code itself. Writes beyond the end are silently ignored; reads beyond the end stop the program with a runtime error (*Memory read fault* on iPhone and iPad, *Memory fault* on Android). Unaligned word accesses are allowed.
- **The program ends where the code ends.** When PC reaches an address that holds no assembled instruction, the simulator stops with *end of program* rather than executing whatever bytes are there.
- **Only user mode.** The **I**, **F** and **T** bits are shown in the CPSR but never change. There are no interrupts, exceptions, other processor modes, Thumb state or coprocessors.
- **Division by zero gives zero.** `SDIV` and `UDIV` by zero put 0 in the destination, as ARM cores do by default.
- **SVC is a service call to the app,** not an exception: the numbers 0 to 7 print and read (Chapter 8), and any other number stops the program.

CHAPTER 4

# The Registers Panel

The **REGISTERS** panel is the processor's dashboard. It shows the sixteen registers and the CPSR flags, and highlights whatever the last instruction changed.




*The Fibonacci program after seven steps: iPhone (left) and Android (right). The last instruction,* `SUBS R2, R2, #1`*, changed R2 (amber) and set the C flag. On iPhone, PC (blue) shows the next instruction's address,* `0x1C`*; on Android, PC is further down the list.*

## 4.1 The CPSR strip

Across the top is the **CPSR**, with its full 32-bit value in hex at the right and seven flag boxes below:

- **N**, **Z**, **C** and **V**, the condition flags (§3.5);
- **I**, **F** and **T**: the IRQ-disable, FIQ-disable and Thumb bits. They are part of a real CPSR and are shown for completeness, but in Pocket ARM they are always 0 (§3.7).

A flag that is set shows **1** in a lit box; a flag that is clear shows a dim **0**. A flag that the last

instruction set glows amber instead of green, so you can see at a glance what a `CMP` or an `S` instruction did. In the hex value, N, Z, C and V are the top four bits: `0x20000000` means only C is set, `0x60000000` means Z and C, `0x80000000` means only N.

## 4.2 The register grid

Below the flags come the sixteen registers. On iPhone and iPad they are laid out in two columns, R0 and R1 on the first row, and so on down to LR and PC. On Android they are in a single column, R0 at the top; scroll down to see SP, LR and PC, which on a small phone are below the bottom of the panel. SP, LR and PC are labelled by name, with their numbers (R13, R14, R15) beside them; R11 and R12 show their aliases **FP** and **IP**. On the iPad the Registers pane is short, so scroll it to reach LR and PC.

Two colours carry meaning:

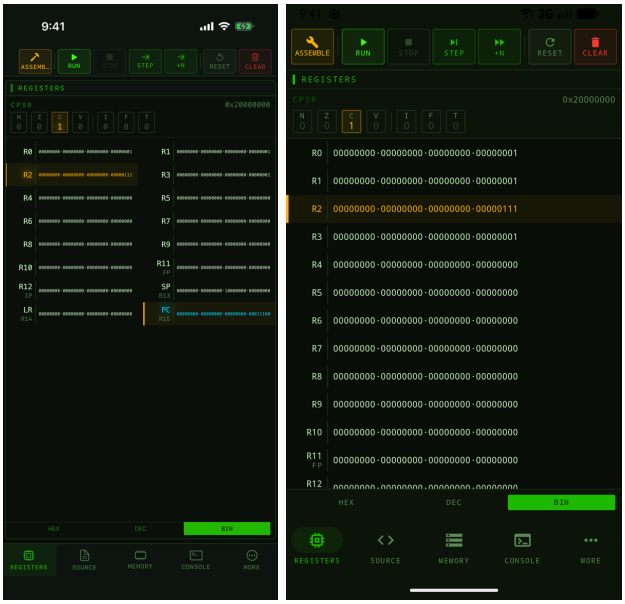
- **Amber**, with a bar at the left edge: the register was changed by the last instruction.
- **Blue**: the program counter, **PC**, always.

The highlights compare each register with its value before the last step. When a program runs at full speed the highlights show only what the final instruction changed.

## 4.3 Hex, decimal and binary

The three buttons at the bottom of the panel, **HEX**, **DEC** and **BIN**, choose how the values are written:

| Mode | How `0xFFFFFFFE` is shown | Notes |
|---|---|---|
| **HEX** | `0xFFFFFFFE` | Always eight digits |
| **DEC** | `-2` | Signed (two's complement), so values above `0x7FFFFFFF` appear negative |
| **BIN** | `11111111·11111111·` `11111111·11111110` | All 32 bits, in four groups of eight |




*Binary mode: iPhone (left) and Android (right). Binary is the best way to watch shifts, `AND`, `ORR` and `BIC` at work.*

On iPhone and iPad you can also tap any register to cycle through the three modes (HEX → DEC → BIN → HEX). On Android, use the buttons; tapping a register does nothing.

The mode affects only the Registers panel. Memory is always shown in hex, and console messages print numbers their own way (Chapter 8).

## 4.4 Changing registers

Registers are changed by your program, and by the toolbar:

- **ASSEMBLE**, **RESET** and **CLEAR** set every register to zero, except SP, which is set to `0x00008000`, and clear all the flags.
- **RUN**, **STEP** and **+N** after a program has finished also start again from these values (§9.3).

You can't type a value into a register in this version. To start a program with particular values, put `MOV` or `LDR Rd, =value` instructions at the top of it.

> **Tip.** If you are studying a function, start your program with the instructions that set up its arguments in R0–R3, then `BL` to it, then `SVC #1`. Several of the example programs (Chapter 11) are written exactly that way.

CHAPTER 5

# The Source Editor and the Assembler

The **SOURCE** tab is where you write programs. Its title bar reads **CODE**, or **CODE — name** once the program has a file name (Chapter 10). Under it is a row of four buttons:

| Button | What it does |
|---|---|
| **SOURCE** | Shows the editor (this chapter) |
| **DISASM** | Shows the assembled program, with breakpoints (Chapter 6) |
| **NAME** | Saves the program under a name (§10.2) |
| **LIBRARY** | Opens a saved or example program (§10.3) |

When the last assembly failed, a red warning triangle and the number of errors appear at the right of this row.

## 5.1 The editor

The editor is a plain text area with line numbers down the left. Tap to place the cursor and type; the usual select, copy, cut and paste gestures of your device all work, which is also the easiest way to move a program between apps (§10.5). Autocorrection and automatic capital letters are switched off.




*The `hello` example in the editor: iPhone (left) and Android (right).*

Long lines wrap onto the next row of the screen. The line numbers count lines of your text, not rows on the screen, so after a wrapped line the numbers and the text drift apart a little. Keep comments short, or turn the phone sideways, if you want them to line up exactly. When you're stuck on an error, the line number in the error message (§5.9) is always the line of your text.

The editor is not saved automatically. What you type stays while the app is open, but if the app is closed (or the system closes it in the background) the editor starts again with the built-in Fibonacci program. Save anything you want to keep with **NAME** (§10.2).

> **Tip.** Editing the source doesn't change the program the simulator is running. After any edit, tap **ASSEMBLE** again.

## 5.2 The shape of a line

Each line of a program has up to three parts, all optional:

```
label:   MNEMONIC  operand, operand, ...     @ comment
```

- A **label** is a name followed by a colon. It names the address of whatever comes next: the next instruction, or the next piece of data. A label can be on a line of its own.
- The **mnemonic** is the instruction (`ADD`, `LDR`, `BNE`) or a directive (`.asciz`, `.word`), followed by its **operands**, separated by commas.
- A **comment** starts with `@` or `;` and runs to the end of the line. Both characters work anywhere on a line.

Spaces and tabs are free: indent however you like. A line may hold only one instruction.

## 5.3 Upper and lower case

Pocket ARM ignores case in mnemonics, register names, labels and shift names. `add r0, r1, #1`, `ADD R0, R1, #1` and `Add r0, R1, #1` are the same instruction, and `loop:`, `LOOP:` and `Loop:` are the same label (so you can't have two labels that differ only in case). In the disassembly everything is shown in capitals.

Case does matter inside strings and character constants: `'a'` is 97 and `'A'` is 65.

## 5.4 Registers

| You can write | Means |
|---|---|
| `R0` … `R15` | The sixteen registers |
| `SP` | R13 |
| `LR` | R14 |
| `PC` | R15 |
| `FP` | R11 |
| `IP` | R12 |

## 5.5 Numbers and characters

Immediate operands start with `#`. In data directives the `#` is left out.

| Form | Example | Value |
|---|---|---|
| Decimal | `#42` | 42 |
| Negative decimal | `#-42` | −42 |
| Hexadecimal | `#0x2A`, `#0X2a` | 42 |
| Character | `#'A'` | 65 |
| Escaped character | `#'\n'` | 10 |

The escapes `\n` (newline, 10), `\r` (return, 13), `\t` (tab, 9) and `\0` (zero) work in both character constants and strings; a backslash before any other character stands for that character, so `\\` is a backslash and `\"` a double quote.

Binary (`0b…`) and octal numbers are not supported. Write them in hex.

### 5.5.1 Which immediates fit

An ARM instruction is only 32 bits, so a data-processing immediate has to fit in 12 of them: an 8-bit value (0–255) that can be rotated right by an even number of places. So `#255`, `#256` (1 rotated), `#0xFF000000` and `#0x3FC` all fit, but `#257` and `#0x101` do not. When an immediate doesn't fit the assembler says so:

*#257 cannot be encoded (8-bit value rotated by an even amount)*

`MOV` (and `MVN`) are more forgiving. If the value doesn't fit, Pocket ARM tries the opposite instruction with the inverted value (so `MOV R0, #-1` becomes `MVN R0, #0`), and if that fails too, uses `MOVW` for any value up to 65,535. For anything else, see §5.7.

## 5.6 Operand forms

Most data-processing instructions accept a flexible second operand, called **Operand2**:

| Form | Example | Meaning |
|---|---|---|
| Immediate | `ADD R0, R1, #4` | R0 = R1 + 4 |
| Register | `ADD R0, R1, R2` | R0 = R1 + R2 |
| Register shifted by a constant | `ADD R0, R1, R2, LSL #2` | R0 = R1 + (R2 × 4) |
| Register shifted by a register | `MOV R0, R1, LSL R3` | R0 = R1 shifted left by R3 |
| Rotate right through carry | `MOV R0, R1, RRX` | R0 = R1 rotated right one bit through C |

The shifts are `LSL` (logical shift left, 0–31), `LSR` (logical shift right, 1–32), `ASR` (arithmetic shift right, 1–32, which keeps the sign), `ROR` (rotate right, 1–31) and `RRX`.

You may also leave out the first source register when it is the same as the destination: `ADD R0, #1` means `ADD R0, R0, #1`, and `LSL R0, #2` means `LSL R0, R0, #2`.

Load and store addresses are written in square brackets. Chapter 7 and Appendix B list every form; the common ones are:

| Form | Example | Address used | Afterwards |
|---|---|---|---|
| Register | `LDR R0, [R1]` | R1 | |
| Offset | `LDR R0, [R1, #8]` | R1 + 8 | |
| Register offset | `LDR R0, [R1, R2, LSL #2]` | R1 + R2 × 4 | |
| Pre-indexed | `LDR R0, [R1, #4]!` | R1 + 4 | R1 = R1 + 4 |
| Post-indexed | `LDR R0, [R1], #4` | R1 | R1 = R1 + 4 |

## 5.7 Big numbers and addresses: LDR =, ADR, MOVW and MOVT

To load a label's address or a constant into a register, write:

```
LDR  R0, =msg        @ R0 = address of msg
LDR  R1, =1000       @ R1 = 1000
ADR  R2, table       @ R2 = address of table
```

In Pocket ARM, `LDR Rd, =value` and `ADR` both become a single `MOVW` instruction, so the value must fit in 16 bits (0–65,535). Every address in Pocket ARM's 64 KB memory does. For larger constants build the number in two halves:

```
MOVW R0, #0x5678     @ R0 = 0x00005678
MOVT R0, #0x1234     @ R0 = 0x12345678 (MOVT sets the top half, keeps the bottom)
```

## 5.8 Directives and data

Directives describe data, or are accepted and ignored so that programs written for other assemblers still assemble. The leading dot is optional: `.word` and `WORD` are the same.

| Directive | Example | Puts in memory |
|---|---|---|
| `.ascii` | `.ascii "Hi"` | The characters, with no terminator |
| `.asciz`, `.string` | `.asciz "Hi"` | The characters followed by a zero byte |
| `.byte`, `.dcb` | `.byte 1, 2, 'A', "xy"` | Bytes (−128 to 255) and strings |
| `.word`, `.dcd` | `.word 10, 20, 0x30` | 32-bit words, little-endian, aligned to 4 bytes |
| `.space`, `.skip`, `.zero` | `.space 32` | That many zero bytes (up to 16,384) |
| `.align` | `.align` | Zero bytes up to the next multiple of 4 |

**All data goes in the data area**, which starts at address `0x2000`, no matter where the directive is written in the program. The instructions go in the text area from `0x0000`. So you can put your strings and tables at the end of the program, after the last instruction, or in between, and the code still runs straight through. A label in front of a data directive gets a data address (`0x2000` and up); a label in front of an instruction gets a code address. Data can be up to 20,480 bytes.

These are accepted and ignored: `.text`, `.data`, `.section`, `.global`, `.globl`, `.arm`, `.syntax`, and the Keil/armasm words `AREA`, `ENTRY`, `END`, `EXPORT`, `IMPORT`, `PROC` and `ENDP`.

> **Note.** `.word` takes numbers only, not labels. To keep a table of addresses, load each one with `LDR Rd, =label`.

## 5.9 Assembling, and reading errors

Tap **ASSEMBLE**. The assembler reads the whole program twice: once to find where every label is, and again to encode each instruction. If all is well the console says *Assembled N instructions successfully.*, the state shows **PAUSED**, and the program is loaded into memory ready to run.

If anything is wrong, nothing is loaded. Instead:

- each bad line's number turns red, with a red dot, in the editor;
- the red triangle and the error count appear next to **LIBRARY**, and the **SOURCE** tab gets a red dot;
- the console lists each error as *Line n: message*, and the Console panel shows the same list in a red box at the bottom;
- the state on the Console panel reads **ERR: Assembly failed with** (it shows only the first 20 characters of a message).




*Three mistakes typed at the top of* `hello`*: iPhone (left) and Android (right).*

The assembler reports every error it finds, not just the first. The messages you are most likely to see:

| Message | Usual cause |
|---|---|
| *Unknown or unsupported mnemonic: X* | A typing mistake (`MOVE`), or an instruction Pocket ARM doesn't have (Appendix B) |
| *Unknown branch target* | The label after `B`/`BL` is misspelt or missing its definition |
| *Unknown label: X* | The label in `LDR Rd, =X` doesn't exist |
| *#n cannot be encoded (8-bit value rotated by an even amount)* | The immediate doesn't fit (§5.5) |
| *MOV: #n cannot be encoded - use LDR Rd, =value or MOVW/MOVT* | Even `MOVW` can't hold it: it's over 65,535 or negative |
| *LDR Rd, =value must fit in 16 bits - use MOVW/MOVT for larger values* | §5.7 |
| *X: expected a destination register* | The first operand is missing or isn't a register |
| *Expected a register or #immediate* | Often a missing `#` in front of a number |
| *Expected an address such as [Rn] or [Rn, #off]* | A load or store without square brackets |
| *Missing ]* | |
| *Offset must be within -4095..4095* | `LDR`/`STR` offset too big |
| *Halfword/signed-byte offset must be within -255..255* | `LDRH`, `STRH`, `LDRSB`, `LDRSH` have a smaller range |
| *LSL amount must be 0-31* (and similar) | Shift amount out of range |
| *PUSH: expected a register list such as {R4-R7, LR}* | Missing braces |
| *X: wrong number of operands* | `MUL`, `MLA`, `UMULL` and so on |
| *X: all operands must be registers* | Multiply and divide take no immediates |
| *X needs at least one value / X takes numbers* | A data directive with nothing or the wrong thing after it |

Appendix B gives the operand forms for every instruction, which is usually the quickest way to see what was expected.

CHAPTER 6

# The Disassembly and Breakpoints

Tap **DISASM** on the **SOURCE** tab to see the program as the simulator sees it: one row per machine instruction, with its address and its 32-bit code. This is also where you set breakpoints.

Until a program has been assembled the view reads **NO INSTRUCTIONS** — *Assemble source code first.*

## 6.1 Reading a row




*The Fibonacci program stopped at a breakpoint on* `SUBS`*: iPhone (left) and Android (right).*

From left to right, each row shows:

1. **The breakpoint circle.** Hollow when there is no breakpoint; solid red when there is.
2. **The PC arrow, ▶.** On the row of the next instruction to run. That row is also tinted blue and its text is blue.
3. **The address**, in hex. Instructions start at `0000` and go up in fours.
4. **The encoding**: the 32-bit machine code word, in hex.

5. **The instruction**, rewritten from your source: capitals throughout, registers as `R0`–`R12`, `SP`, `LR` and `PC` (so `fp` becomes `R11`), labels in capitals, numbers in decimal, and comments removed.
6. **The source line**, such as **L13**: the line of your program it came from.

As the program runs, the list scrolls itself to keep the PC row in view.

Only instructions appear here. Labels, comments, blank lines and directives don't produce instructions, which is why the line numbers skip. Data (strings, `.word` tables) lives in the data area and can be seen on the Memory panel (Chapter 7).

> **Note.** A few instructions are stored as a different instruction from the one you wrote, but the row still shows what you wrote: `LDR R0, =msg` and `ADR` become `MOVW`, `MOV R0, #-1` becomes `MVN R0, #0`, `PUSH` and `POP` are `STMDB` and `LDMIA` on SP, and `LSL R0, R1, #2` is really `MOV R0, R1, LSL #2`. The encoding column tells the truth.

## 6.2 Reading the encoding

You don't need to decode machine code to use Pocket ARM, but the encoding column rewards a little curiosity. Every ARM instruction begins with a four-bit **condition** field, so the first hex digit tells you when the instruction runs: **E** means *always* (`AL`), **1** means *NE*, **0** means *EQ*, **B** means *LT*, and so on (Appendix B). In the Fibonacci program, `BNE LOOP` is `1AFFFFFA`: condition 1 (NE), `A` for a branch, and `FFFFFA`, which is −6 in 24-bit two's complement: jump back six instructions from the one after next.

`SVC #1` is `EF000001`: always (`E`), `F` for supervisor call, and the number 1. `MOV R0, #0` is `E3A00000`.

## 6.3 Breakpoints

A **breakpoint** tells the simulator to stop just *before* running a particular instruction. Tap the circle at the left of a row to set one, and tap it again to clear it. The console confirms each change, with a message such as *Breakpoint set at 0x0018* (but see the note below for iPhone and iPad).

When a running program reaches an instruction with a breakpoint, it stops and:

- the console says *Breakpoint hit at 0x18* (on Android, *0x0018*);
- the state on the Console panel reads **HALT: BREAKPOINT HIT**;
- PC points at the breakpoint row, which has not yet run.

From there you can look at the registers and memory, **STEP** through the next few instructions, or tap **RUN** to carry on to the next breakpoint (or the end). In a loop, a breakpoint stops the program once on every trip round.

Breakpoints are checked when the program arrives at an instruction, so a breakpoint on the very first instruction (address `0000`) never fires unless the program later branches back to it: the program starts there rather than arriving there. Put it on the second instruction instead.

Breakpoints belong to the assembled program. **ASSEMBLE** and **CLEAR** remove them all; **RESET** keeps them.

> **Bug in 1.0.3 (iPhone and iPad).** The *Breakpoint set* and *Breakpoint cleared* messages pad the address on the wrong side, so a breakpoint at `0x18` is reported as *Breakpoint set at 0x1800*, and one at `0x30` as `0x3000`. The breakpoint itself is in the right place, and the *Breakpoint hit* message shows the correct address (*0x18*). Android reports *0x0018* in both messages.

## 6.4 Moving between the editor and the disassembly

On a phone, the **SOURCE** tab always opens on the editor. If you switch to another tab while looking at the disassembly, tap **DISASM** again when you come back. On a tablet the Code pane stays where you left it.

CHAPTER 7

# The Memory Panel

## 7.1 The memory map

Pocket ARM's memory is 64 KB, addresses `0x0000` to `0xFFFF`, laid out like this:

| Address | Area | What is there |
|---|---|---|
| `0x0000`– | **Text** | Your instructions, four bytes each, from `0x0000` upwards |
| `0x2000`– | **Data** | Everything from `.ascii`, `.asciz`, `.byte`, `.word`, `.space` and the other data directives, in the order they appear in the program (up to 20,480 bytes, so up to `0x6FFF`) |
| …`-0x7FFF` | **Stack** | SP starts at `0x8000` and the stack grows downwards from there |
| `0x8000`–`0xFFFF` | Free | Unused by the app; your program may use it |

Nothing stops a program from reading or writing anywhere in the 64 KB. If your program is very long (more than 2,048 instructions) its code runs on past `0x2000` into the data, and if a program pushes a great deal onto the stack it can grow down into the data. Neither happens in programs of a sensible size.

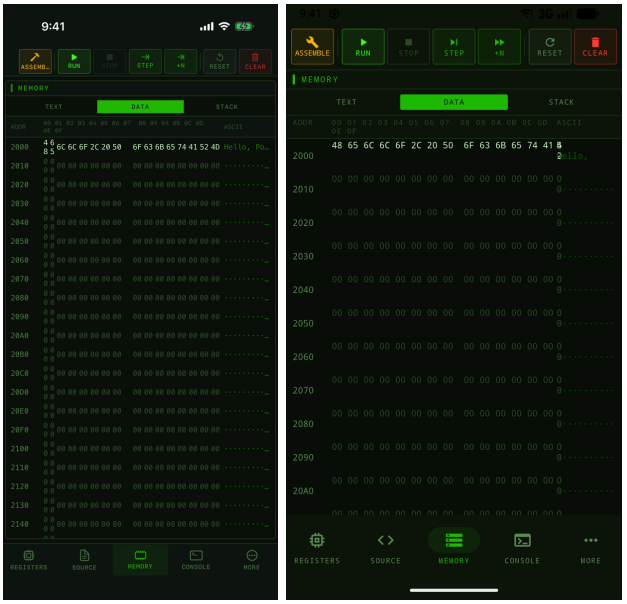
Words are stored **little-endian**: the least significant byte at the lowest address. So the word `0x12345678` stored at `0x2000` appears in memory as the bytes `78 56 34 12`.

## 7.2 Reading the dump

The three buttons at the top, **TEXT**, **DATA** and **STACK**, choose which area to show:

| Button | Shows |
|---|---|
| **TEXT** | `0x0000` to `0x1FFF` |
| **DATA** | `0x2000` to `0x3FFF` |
| **STACK** | `0x7F00` to `0x80FF`, the top 256 bytes of the stack and the 256 bytes above it |

Each row shows sixteen bytes: the address of the first, then the bytes in hex in two groups of eight, then the same bytes as text in the **ASCII** column, where anything that isn't a printable character is shown as a dot (·).




*The data area after running `hello`: iPhone (left) and Android (right). The string "Hello, PocketARM" starts at `0x2000` (`48 65 6C 6C 6F` is "Hello") and ends with the zero byte that `.asciz` adds.*

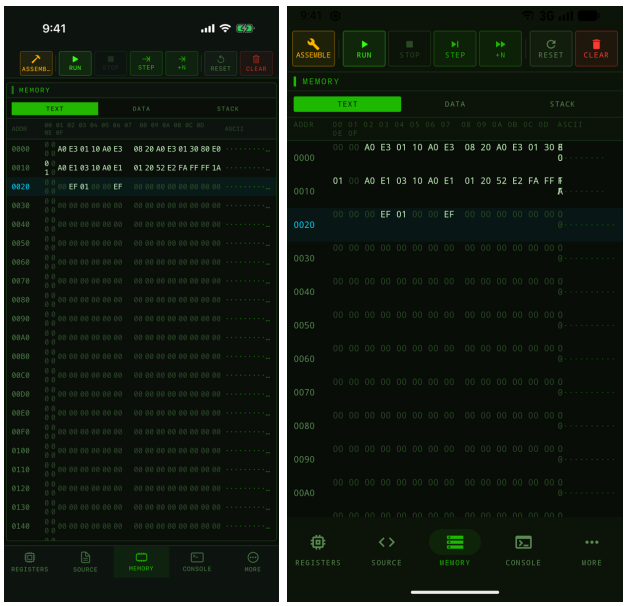
Zero bytes are dim, so the parts of memory your program has used stand out. The row holding the instruction PC points to has a blue address, and the four bytes of that instruction are blue too: on the **TEXT** view you can watch PC move through the code as you step.

The dump is updated after every instruction, so with the panel open (or on a tablet, where it is always open) you can watch a store instruction change memory as it happens.

> **Phone screens are narrow.** Sixteen bytes and their text don't quite fit across an upright phone, so on most phones the first byte of each row wraps onto two lines and the ASCII column is cut short. Turn the phone sideways, or use a tablet, to see whole rows. The data is all there; it's only the layout that is cramped.

## 7.3 Seeing your program's code

The **TEXT** view shows the machine code itself. Compare it with the encoding column on the **DISASM** view (§6.1): the instruction `E3A00000` at address `0000` appears in memory as `00 00 A0 E3`, because memory is little-endian.



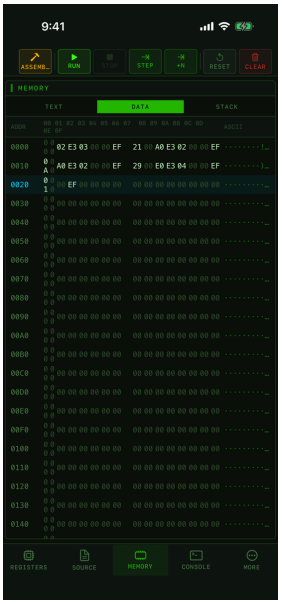

*The text area holding the Fibonacci program: iPhone (left) and Android (right).*

## 7.4 Watching the stack

Select **STACK** and step through a program that uses `PUSH` and `POP`, or `BL` to a function that saves registers, such as the factorial example (§11.5). Each `PUSH {R4, LR}` stores eight bytes just below SP and moves SP down by 8; you'll see the saved values appear at the bottom of the view (around `0x7FF8`–`0x7FFF`) and, as recursion goes deeper, further up. The iPad screenshot in §2.4 shows the factorial example six calls deep.

## 7.5 A bug in 1.0.3: the segment buttons lag on iPhone and iPad

On iPhone and iPad, when you tap **TEXT**, **DATA** or **STACK**, the button lights up straight away but the rows underneath still show the *previous* area until something else updates the panel. You can see it in the iPad screenshot in §2.4: **STACK** is selected, but the rows are the text area from `0000`.




*iPhone, one tap on **DATA** after running `hello`: the button is lit, but the rows are still the text area from 0000. Compare the picture in §7.2, taken after a second tap.*

The fix is simple: **tap the button a second time.** The rows then match the button. Stepping or running the program also brings them into line. Android doesn't have this problem.

CHAPTER 8

# The Console, Output and Input

The **CONSOLE** panel is where the app talks to you and where your program prints. It is also where your program asks for input.

## 8.1 The status bar

At the top of the console a coloured dot and a word show the simulator's state:

| State | Colour | Meaning |
|---|---|---|
| **IDLE** | dim green | No program assembled |
| **PAUSED** | amber | Assembled and ready, or stopped between steps, or waiting for input |
| **RUNNING** | bright green | Running at full speed |
| **HALT: END OF PROGRAM** | green | PC ran past the last instruction |
| **HALT: SVC CALL** | green | The program called `SVC #1` (or an unknown SVC number) |
| **HALT: BREAKPOINT HIT** | green | Stopped at a breakpoint (§6.3) |
| **HALT: USER STOPPED** | green | You tapped **STOP** |
| **ERR: ...** | red | Assembly failed, or a runtime error; only the first 20 characters of the message are shown here |

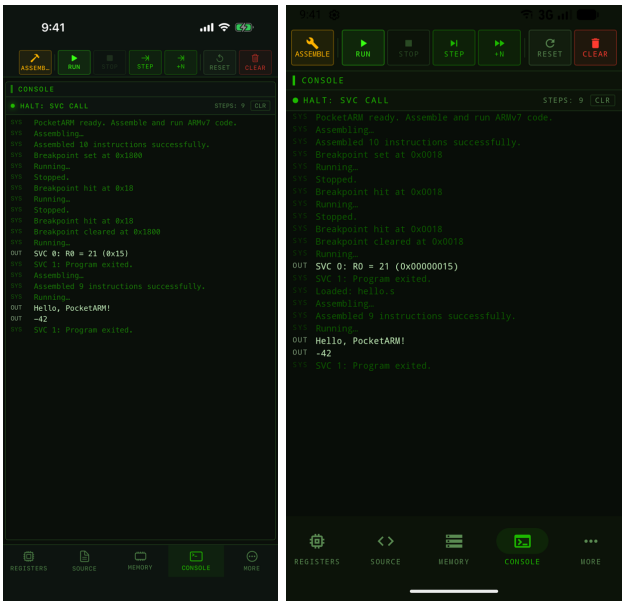
At the right, **STEPS:** counts the instructions run since the program was assembled, reset or restarted, and **CLR** clears the messages.

> **Bug in 1.0.3 (Android).** **CLR** does nothing on Android. The console keeps only its last 200 messages anyway, and **CLEAR** in the toolbar empties it (along with the assembled program; §9.4).

## 8.2 Messages

Each line has a three-letter tag and a colour:

| Tag | Colour | Comes from |
|---|---|---|
| **SYS** | dim green | The app: *Assembling…*, *Running…*, *Stopped.*, *Halted: … after N steps.* and the like |
| **OUT** | pale green | Your program's output |
| **ERR** | red | Assembly errors and runtime errors |
| **INF** | amber | Notices, and an echo of what you typed in answer to an input request |




*The `hello` example's output: iPhone (left) and Android (right).*

On Android, opening a file from the library or saving one also adds *Loaded: name.s* or *Saved: name.s*. The iPhone and iPad version doesn't log these.

## 8.3 Supervisor calls: how a program prints

Programs talk to the outside world through the **SVC** (supervisor call) instruction. On a real system SVC hands control to the operating system; in Pocket ARM it hands it to the app, which looks at

the number and, for some calls, at R0 and R1:

| Call | Does | Arguments | Result |
|---|---|---|---|
| `SVC #0` | Print R0 as a line: `SVC 0: R0 = 21 (0x15)` | R0 | |
| `SVC #1` | Exit: stop the program | | |
| `SVC #2` | Print one character | R0 = character code | |
| `SVC #3` | Print a zero-terminated string | R0 = address of the string | |
| `SVC #4` | Print R0 as a signed decimal number | R0 | |
| `SVC #5` | Read one character | | R0 = character code |
| `SVC #6` | Read a line of text into memory | R0 = buffer address, R1 = buffer size | R0 = length read |
| `SVC #7` | Read a number | | R0 = the number |
| any other | Prints *SVC n called (R0=…)* and stops, like `SVC #1` | | |

`SVC #0` is a debugging aid: it always prints a whole line, in decimal and hex, labelled with R0. The decimal is unsigned on iPhone and iPad (`4294967295` for −1) and signed on Android (`-1`), and Android always writes the hex with eight digits (`0x00000015` rather than `0x15`). Calls 2, 3 and 4 are for real output. They print nothing until a line is finished: characters build up on the current line, and a newline character (10) ends it and sends it to the console. So to print a number on a line of its own:

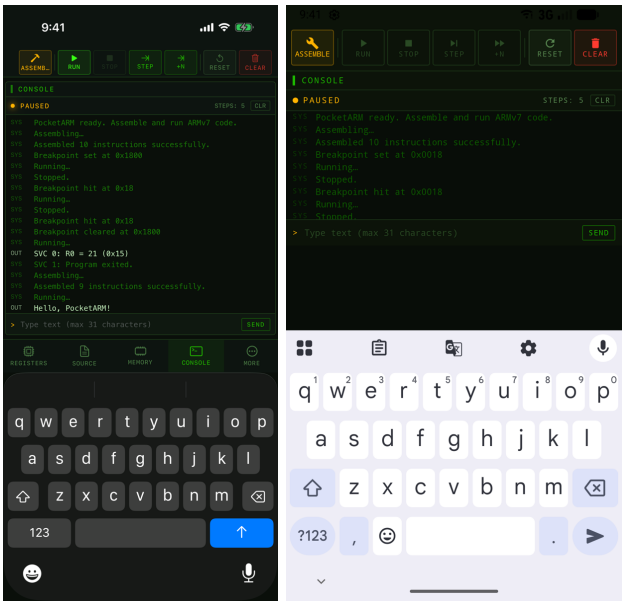
```
MOV  R0, #42
SVC  #4          @ "42" goes on the current line
MOV  R0, #10
SVC  #2          @ newline: the line appears
```

A line that hasn't been ended with a newline is shown anyway when the program stops, asks for input, or the app prints a message of its own. Carriage returns (13) and zero bytes are ignored. `SVC #3` prints at most 4,096 characters, and stops the program with an error if the string runs off the end of memory.

The calls take their arguments in R0 and R1 and only change R0 (and only for the read calls), so you don't need to save any other registers around them.

## 8.4 Input

When a program calls `SVC #5`, `#6` or `#7`, it pauses, and an input bar appears at the bottom of the console with a `>` prompt, a hint and a **SEND** button. The keyboard comes up by itself.




*The `input` example waiting for a name: iPhone (left) and Android (right).*

Type your answer and tap **SEND** (or the send/return key). What you typed is echoed in amber (`> Ada`) and the program carries on; if it was running at full speed it goes on running, and if you were stepping it stays paused.

| Call | Hint shown | What you can type |
|---|---|---|
| `SVC #5` | *Type a character, then Send* | Anything; the first character's code goes in R0. Sending nothing gives 10 (newline). |
| `SVC #6` | *Type text (max n characters)* | A line of text. It is copied into memory at R0 with a zero byte after it, and its length goes in R0. If it is longer than R1 − 1 characters, the rest is dropped, with a notice. |

| Call | Hint shown | What you can type |
|---|---|---|
| `SVC #7` | *Type a number (decimal or 0x hex)* | A decimal number, optionally with `-` or `+`, or hex with `0x`. If it isn't a number the app asks again: *Not a valid integer - try again (decimal, or hex like 0x1F).* |

Text is stored as UTF-8, so accented letters take two bytes each. The buffer for `SVC #6` must lie inside memory and be at least one byte long; otherwise the program stops with *SVC 6: invalid buffer*.




*The* ***input*** *example finished: iPhone (left) and Android (right).*

While the program is waiting, **RUN** and **STEP** can't continue it, because there is nothing to continue with until you answer. On Android those buttons are dimmed; on iPhone and iPad they stay lit, and tapping them prints *Waiting for input - type a value and press Send.* **RESET** and **CLEAR** abandon the input and the run.

## 8.5 Runtime errors

When the simulator can't carry on, it stops, prints *Runtime error: …* in red, and shows **ERR:** in the status bar. The runtime errors are:

- *Memory read fault at 0x…* (*Memory fault at 0x…* on Android): a load from beyond `0xFFFF`;
- *SVC 3: string runs past the end of memory at 0x…*: a string with no zero byte before the end of memory;
- *SVC 6: invalid buffer (R0=…, size R1=…).*

Tapping **RUN** or **STEP** after an error starts the program again from the beginning (§9.3).

CHAPTER 9

# Running and Debugging

The toolbar at the top of the screen drives the simulator. This chapter explains each button, what happens when a program finishes, and how to use them together to find out why a program doesn't do what you meant.

## 9.1 The toolbar

| Button | What it does | Available |
|---|---|---|
| **ASSEMBLE** | Assembles the editor's program and loads it, ready to run from address 0 | Always |
| **RUN** | Runs at full speed until the program ends, hits a breakpoint, asks for input, or you tap **STOP** | After assembling, when not running |
| **STOP** | Stops a running program where it is | While running |
| **STEP** | Runs exactly one instruction | After assembling, when not running |
| **+N** | Runs a number of instructions that you choose, then stops | After assembling, when not running |
| **RESET** | Puts the processor back to the start: registers cleared, SP = `0x8000`, memory reloaded from the program, step count 0 | Always |
| **CLEAR** | Throws away the assembled program, its breakpoints and the console messages | Always |

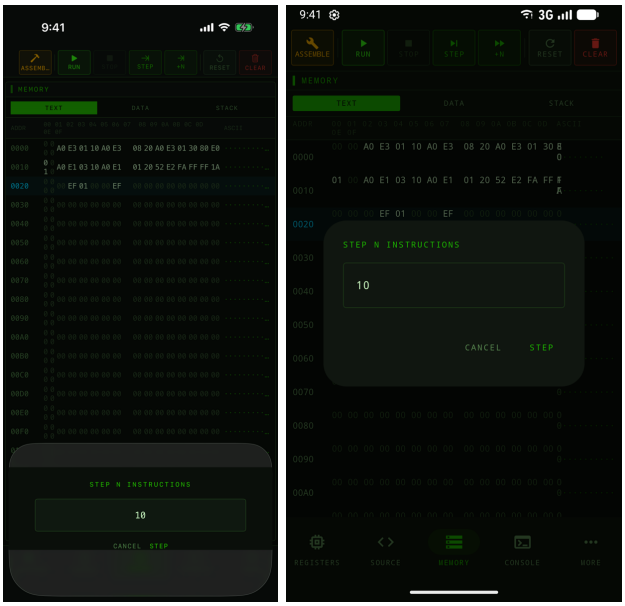
**RESET** keeps the assembled program and its breakpoints, so it is the button for "run that again". **CLEAR** doesn't touch the editor; your source stays where it is.

## 9.2 Running, stepping and +N

**RUN** runs about two hundred instructions a second on iPhone and iPad, and somewhat faster on Android. That is quick enough for every example in this guide to finish in well under a second, and slow enough that a program stuck in a loop can be stopped. While a program runs, the **CONSOLE** tab shows a green dot.

**STEP** is the most useful button in the app. Each tap runs one instruction and stops, so you can see exactly what that instruction did: the register it changed glows amber (§4.2), a flag that changed glows amber (§4.1), and PC moves to the next instruction.

**+N** asks how many instructions to run (10, unless you change it), then runs that many, stopping early if the program ends, hits a breakpoint or asks for input. It's handy for getting through a loop quickly.




*The +N prompt: a sheet on iPhone (left) and a dialog on Android (right).*

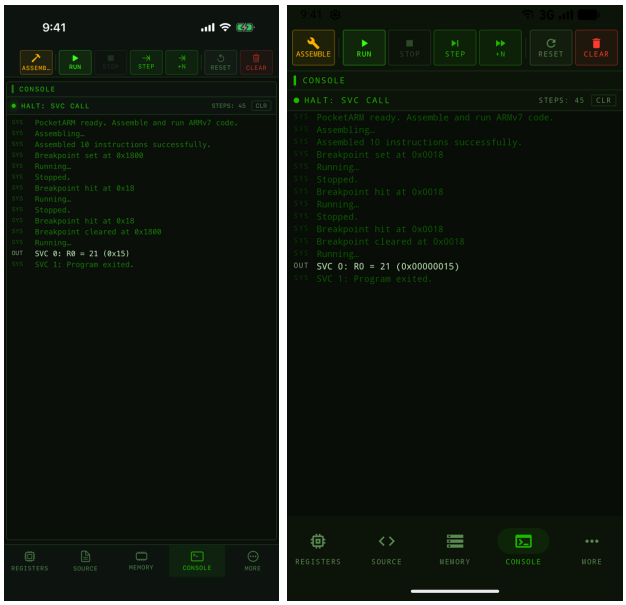
## 9.3 When a program finishes

A run comes to an end in one of four ways, shown on the Console panel's status bar (§8.1):

- **HALT: SVC CALL**: the program called `SVC #1`.
- **HALT: END OF PROGRAM**: PC moved past the last instruction. Many of the examples end like this.
- **ERR: ...**: a runtime error (§8.5).
- **HALT: BREAKPOINT HIT** or **HALT: USER STOPPED**: the program was interrupted part-way.

After the first three, the program is finished: tapping **RUN**, **STEP** or **+N** starts it again from the top, exactly as if you had tapped **RESET** first. After a breakpoint or **STOP**, the same buttons *continue* from where the program stopped.

**STEPS** on the Console panel shows how many instructions ran. When a program runs off its end, the console also sums up with a line such as *Halted: end of program after 55 steps.*; a program that exits with `SVC #1` says *SVC 1: Program exited.* instead.




*The Fibonacci program after stopping twice at a breakpoint, which was then cleared, and running to the end: iPhone (left) and Android (right). The program ran 45 instructions in all.*

## 9.4 A debugging routine

When a program gives the wrong answer, resist the urge to stare at the source. Let the simulator show you instead:

1. **Assemble** and check the console for errors first.
2. **Set a breakpoint** on the first instruction of the part you suspect (§6.3), and **RUN** to it. Everything before it ran at full speed.
3. **Predict, then STEP.** Before each tap, say what you expect the instruction to do. When the amber highlight shows something else, you have found the problem.
4. **Check the flags at every conditional branch.** Most loop bugs are an off-by-one in a `CMP` or the wrong condition (`BLT` instead of `BLE`, signed instead of unsigned: see Appendix B).
5. **Watch memory** for loads and stores: switch the Memory panel to **DATA** or **STACK**, or use a tablet, where it is always visible.

6. **RESET** and run again with what you learned. Breakpoints survive a reset, but not a re-assembly.

> **Tip.** If a program seems to hang, tap **STOP** and look at PC on the Registers panel and the ▶ in the disassembly: you'll be inside the loop that never ends. Then look at the register the loop counts with.

## 9.5 Common surprises

- **The program ran straight into the next function.** Execution doesn't stop at a label. Put `SVC #1` after your main code so it doesn't fall through into the functions that follow it, as the examples do.
- **`CMP` didn't seem to work.** Check you used a conditional instruction straight after it, and that nothing in between (an `ADDS`, `SUBS`, or another `CMP`) changed the flags.
- **`SUBS` vs `SUB`.** Only the `S` form sets the flags. `SUB R2, R2, #1` followed by `BNE` tests whatever flags were set earlier.
- **`BX LR` went to address 0.** LR was overwritten by a nested `BL`. Save it with `PUSH {LR}` (or `PUSH {R4, LR}`) at the start of any function that calls another, and return with `POP {PC}`.
- **My edit made no difference.** Tap **ASSEMBLE** after editing.

CHAPTER 10

# Saving, Loading and Moving Programs

Pocket ARM keeps your programs as plain text files with the extension `.s`, the usual extension for assembly language. Saving and opening them inside the app works the same way on every device. Getting files *into* and *out of* the app is where the iPhone and Android versions differ most, so this chapter takes each in turn.

## 10.1 The library

The app keeps one folder of programs, its **library**. The first time the app starts, it copies the eleven example programs into it (Chapter 11):

`binary_search`, `bit_counting`, `bubble_sort`, `caesar_cipher`, `factorial`, `fibonacci`, `gcd`, `hello`, `input`, `new_instructions_demo` and `string_length`.

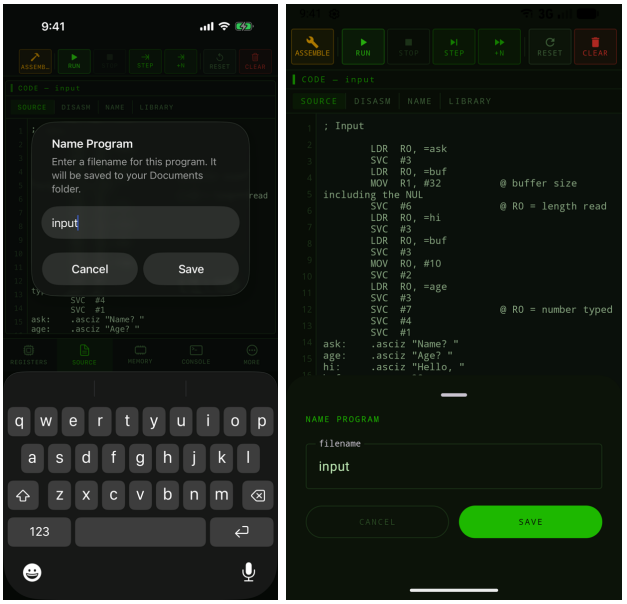
They are ordinary files: you can change them, and save over them. The app copies them only once, so a changed or deleted example stays changed or deleted. To get the originals back, delete and reinstall the app (on Android, **Settings › Apps › Pocket ARM › Storage › Clear storage** also works, but it deletes your own programs too).

| | iPhone and iPad | Android |
|---|---|---|
| Where the library lives | The app's **Documents** folder, which the Files app shows as **On My iPhone › PocketARM** (or **On My iPad**) | The app's private storage, which only the app can see |
| Seen in the Files app | Yes | No |
| Included in device backups | Yes, in iCloud or computer backups of the device | Yes, when Google backup is turned on for the device |

## 10.2 Saving: NAME

To save the program in the editor, tap **NAME** on the **SOURCE** tab, type a name, and tap **Save** (iPhone) or **SAVE** (Android). Leave off the extension: the app adds `.s`. The title bar then reads

**CODE — name.**



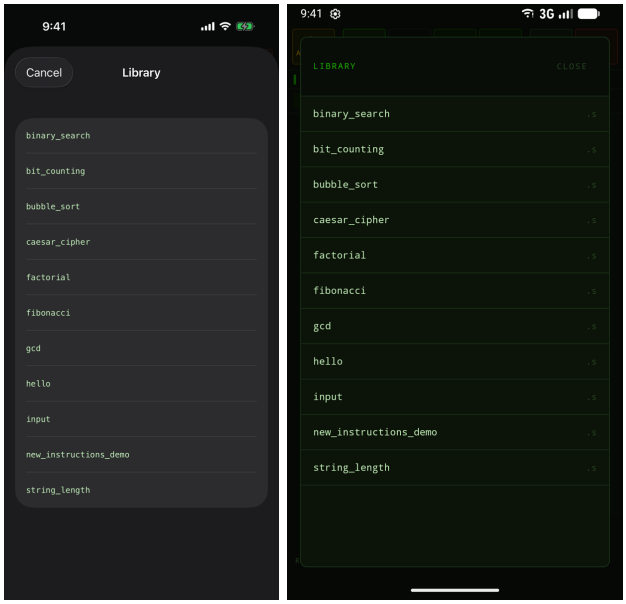

*Saving: an alert on iPhone (left), a sheet sliding up from the bottom on Android (right).*

- The name box starts with the name of the file you opened or last saved, so to save your changes, tap **NAME** and then **Save**.
- Saving over an existing name **replaces that file without asking**. To keep the original, type a new name.
- Use letters, digits, `_` and `-`. Avoid `/`, which can't be part of a file name.
- Android confirms with *Saved: name.s* in the console; iPhone and iPad save silently.

There is no automatic save, and no undo for a save. Each time you reach something that works, save it, perhaps with a number on the end (`sort1`, `sort2`) so you can go back.

## 10.3 Opening: LIBRARY

Tap **LIBRARY** to see every `.s` file in the library, in alphabetical order. Tap one to load it into the editor. It replaces whatever was in the editor, without asking, so save first if you need to.




*The library: a sheet with **Cancel** on iPhone (left); a full-screen panel with **CLOSE** on Android (right).*

Loading a file doesn't assemble it: tap **ASSEMBLE** next. The library has no delete or rename button in either version.

## 10.4 iPhone and iPad: moving files with the Files app

Because the library is an ordinary folder in **On My iPhone** (or **On My iPad**), the Files app is the way to manage it:

**To bring a program in** (from iCloud Drive, a USB stick, an email attachment you saved, a Mac):

1. Open the **Files** app and find the `.s` file.
2. Touch and hold it, choose **Copy** (or **Move**).
3. Go to **Browse** › **On My iPhone** › **PocketARM**, touch and hold an empty space, and choose **Paste**.
4. In Pocket ARM, tap **LIBRARY**. The file is in the list.

The file must end in `.s` to appear in the library. A file ending in `.asm` or `.txt` can be renamed in the Files app (touch and hold, **Rename**).

**To send a program out:** in the Files app, touch and hold the file in **On My iPhone** › **PocketARM** and choose **Share** (to send it by Mail, Messages, AirDrop and so on), or **Copy** it to another folder such as iCloud Drive.

**To delete or rename a program:** do it in the same Files app folder.

**From a Mac or PC:** connect the device with a cable, select it in the Finder sidebar (on Windows, in Apple Devices or iTunes), choose **Files**, and drag `.s` files into or out of **PocketARM**.

> **Opening a file directly in 1.0.3.** Pocket ARM tells iOS that it can open `.s` and `.asm` files, so they may offer **Open in PocketARM** (or show the Pocket ARM icon) in the share sheet. In version 1.0.3 this doesn't load the program into the editor: at most, the panel title changes to the file's name while the editor keeps the old program (and **NAME** would then save that old program under the new name). Use the copy-and-paste-in-Files method above instead.

## 10.5 Android: copy and paste

The Android version keeps its library in private app storage. Other apps, the Files app and a computer connected by USB can't see it, and the app has no import, export or share button for programs. The way to move a program in or out is the clipboard:

**To bring a program in:**

1. In the app where the program is (a browser, an email, a notes app, Google Drive's text viewer), select all its text and **Copy** it.
2. In Pocket ARM, open the **SOURCE** tab, select everything in the editor (touch and hold, then **Select all**), and **Paste** over it.
3. Tap **NAME** and save it, so it's in the library from now on.

**To send a program out:** open it from the **LIBRARY**, touch and hold in the editor, **Select all**, **Copy**, and paste it into an email, a message, or a document.

The same method works on iPhone and iPad too, and it's often the quickest way to try a program from a web page or course notes on any device.

## 10.6 Moving programs between an iPhone and an Android phone

The files are identical plain text on both, so any route that carries text works: email a program to yourself from one and copy it from the email on the other, or keep your programs in a shared notes app or cloud folder.

CHAPTER 11

# The Example Programs

Pocket ARM comes with eleven example programs in its library (§10.1). This chapter takes them roughly in order of difficulty: what each one does, what to look for as you step through it, and what result to expect. The step counts are those shown by **STEPS** on the Console panel when the program has finished.

Open any of them with **LIBRARY** on the **SOURCE** tab, then tap **ASSEMBLE**.

| Program | Shows | Prints? | Ends with |
|---|---|---|---|
| `hello` | Strings and characters, SVC 2, 3, 4 | Yes | `SVC #1`, 9 steps |
| `input` | Reading text and numbers, SVC 6, 7 | Yes | `SVC #1` |
| `fibonacci` | A counted loop with `SUBS`/`BNE` | No: R0 = 55 | End of program, 55 steps |
| `gcd` | A function, `CMP` with `BEQ`/`BGT` | No: R0 = 6 | `SVC #1`, 27 steps |
| `bit_counting` | `AND`, `LSR`, counting bits | No: R1 = 4 | `SVC #1`, 56 steps |
| `factorial` | Recursion, `PUSH`/`POP`, `MUL` | No: R0 = R1 = 720 | `SVC #1`, 50 steps |
| `new_instructions_demo` | `SDIV`, `MLS`, scaled offsets, conditional `MOV` | Yes | `SVC #1`, 47 steps |
| `string_length` | `LDRB` and a pointer | No: R1 = length | `SVC #1` |
| `caesar_cipher` | Character arithmetic, nested `BL` | No | End of program |
| `bubble_sort` | Nested loops, `LDR`/`STR` with offsets | No | End of program |
| `binary_search` | Divide and conquer, `ASR`, scaled addressing | No: R0 = index | `SVC #1` |

**Before you start.** Four of the examples, `string_length`, `caesar_cipher`, `bubble_sort` and `binary_search`, work on a string or an array at a fixed address (`0x100` or `0x200`) but don't put anything there, so they run on memory full of zeros: the string is empty, the array is all zeros, and the search finds nothing. They are still worth stepping through to see how they work, and §11.9–§11.11 show how to give them real data. That makes good practice in its own right.

## 11.1 hello

```
        LDR  R0, =msg
        SVC  #3               @ print string
        MOV  R0, #'!'
        SVC  #2               @ print character
        MOV  R0, #10
        SVC  #2               @ newline ends the console line
        MOV  R0, #-42
        SVC  #4               @ print signed integer
        SVC  #1               @ exit
msg:    .asciz "Hello, PocketARM"
```

The console shows `Hello, PocketARM!` and then `-42`. Points to notice:

- `msg` is defined after the code, but because it is data it lives at `0x2000`, so `LDR R0, =msg` loads `0x2000`. Look on the Memory panel's **DATA** view to see the string (§7.2).
- The `!` appears on the same line as the string: nothing is printed until the newline (10).
- `-42` is printed without a newline after it; the console shows it when the program exits.
- `MOV R0, #-42` doesn't fit as an immediate, so the assembler quietly uses `MVN R0, #41`. Check the encoding on **DISASM**.

## 11.2 input

Asks *Name?* with `SVC #6` into a 32-byte buffer, prints *Hello,* and the name, then asks *Age?* with `SVC #7` and prints the number back. Chapter 8 has screenshots. Try typing more than 31 characters for the name, a negative age, a hex age such as `0x24`, or a word instead of a number.

## 11.3 fibonacci

Computes the first ten Fibonacci numbers in a loop, leaving each in R1 in turn; at the end `MOV r0, r1` copies F(10) = 55 into R0, and the program runs off its end (**HALT: END OF PROGRAM**). Step it with the Registers panel open and watch R1 go 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 while R0 counts down from 10, and watch the Z flag on `SUBS` as R0 reaches zero.

The program the app starts with (§2.3) is a variation of this one that runs eight times and prints R0 with `SVC #0`.

## 11.4 gcd

The greatest common divisor of 48 and 18, by Euclid's subtraction method: while the numbers differ, subtract the smaller from the larger. `_start` puts the arguments in R0 and R1, calls `gcd` with `BL`, and exits with `SVC #1` when it returns. Watch LR: `BL gcd` sets it to `0x0C`, the address of the `SVC #1`, and `BX lr` returns there. Result: R0 = 6.

## 11.5 factorial

The classic recursive function: *fact(n) = n × fact(n − 1)*, with *fact(1) = 1*. Each call saves R4 (which holds its own *n*) and LR on the stack with `PUSH {r4, lr}`, and returns with `POP {r4, pc}`, which restores R4 and loads the saved return address straight into PC.

This is the program to watch the stack with. Set a breakpoint on `MOV r0, #1` under `base_case:` (address `0x30`), run, and switch the Memory panel to **STACK** (tap it twice on iPhone and iPad, §7.5). SP is `0x7FD0`: six calls deep, 48 bytes of saved registers. Then **STEP** and watch the `POP`s unwind and `MUL` build 720.

## 11.6 bit_counting

Counts the 1 bits in `0xA5` (binary `10100101`) by repeatedly taking the lowest bit with `AND r3, r2, #1` and shifting right with `LSR`. Put the Registers panel in **BIN** mode to watch R2 lose a bit each time round. Result: R1 = 4.

## 11.7 new_instructions_demo

A tour of instructions that were added in later ARM versions or are easy to miss:

- `SDIV R2, R0, R1` divides 100 by 7 (14), and `MLS R3, R2, R1, R0` computes the remainder, 100 − 14 × 7 = 2.
- `LDR R7, [R4, R6, LSL #2]` loads `nums[i]` from a `.word` table, scaling the index by 4: the sum is 100.
- `TST R0, #1` tests the lowest bit without changing R0, and `MOVEQ`/`MOVNE` pick `'E'` (even) or `'O'` (odd) without a branch.

It prints `14 2`, `100` and `E` on three lines.

## 11.8 string_length

The C function `strlen`: walk a pointer along a string with `LDRB`, counting bytes until a zero. As supplied it looks at address `0x200`, where there is nothing, so R1 = 0 after 9 steps. To measure a real string, change the first instruction and add the string at the end:

```
_start:
    LDR  r0, =text     ; was: MOV r0, #0x200
    BL   strlen
    MOV  r0, r1
    SVC  #4            ; print the length
    SVC  #1
    ...                ; strlen as before
text:   .asciz "Pocket ARM"
```

It prints 10.

## 11.9 caesar_cipher

Encrypts a string in place by shifting each letter three places (A→D, x→a), leaving other characters alone. It uses a second function, `mod26`, to wrap round the alphabet. Again the string is expected at `0x200` and isn't there, so the program finishes after 8 steps having changed nothing. Give it a string as in §11.8 (`LDR r0, =text` and `text: .asciz "Hello, World"`), and add `LDR r0, =text`, `SVC #3` and `SVC #1` after the last instruction to print the result: *Khoor, Zruog.*

> **Watch out.** `caesar_cipher` calls `mod26` with `BL` from inside its loop, which overwrites LR. That's fine here because the outer code never returns with `BX lr`, but it's exactly the bug described in §9.5 if you turn the loop into a function.

## 11.10 bubble_sort

Sorts an array of words into ascending order: compare each neighbouring pair, swap if they're the wrong way round, and repeat with one fewer element each pass. As supplied it sorts six zeros at `0x100`. To sort real numbers, replace `MOV r0, #0x100` with `LDR r0, =arr` and add a table at the end:

```
arr:    .word 42, 7, 19, 3, 88, 1
```

Then view **DATA** on the Memory panel before and after running: `2A 00 00 00 07 00 00 00 …` becomes `01 00 00 00 03 00 00 00 …`. (`BLE no_swap` compares signed numbers, so negative numbers sort correctly too.)

## 11.11 binary_search

Searches a sorted array for 42 by repeatedly halving the range. As supplied the array at `0x100` is all zeros, so 42 isn't found and R0 ends as `0xFFFFFFFF` (−1, shown as `-1` in **DEC** mode). Give it real data the same way, with `LDR r0, =arr` and:

```
arr:    .word 3, 8, 15, 23, 42, 57, 64, 91
```

and it finds 42 at index 4 after 40 steps (add `SVC #4` after the `BL` to print it). Change the target in R2 to a number that isn't there, such as 40, and it returns −1.

The line `ADD r6, r0, r5, LSL #2` computes the address of `arr[mid]` in one instruction: base plus index times four.

CHAPTER 12

# Exercises and Recipes

The best way to learn assembly language is to write it. Each exercise below has a goal, a hint, and a worked answer. Try it yourself first; then type in the answer, assemble it, and step through it. Every answer here has been run on Pocket ARM's own assembler and simulator and prints exactly what is shown.

Start each exercise with a fresh editor: open any program from the **LIBRARY**, select all the text, and delete it. Save your answers with **NAME** as you go.

## 12.1 Add up 1 to 10

**Goal:** print the sum 1 + 2 + … + 10.

**Hint:** keep a running total in one register and a counter in another; loop while the counter is ≤ 10.

```
@ Add the numbers 1 to 10
        MOV   R0, #0          @ total
        MOV   R1, #1          @ n
loop:   ADD   R0, R0, R1
        ADD   R1, R1, #1
        CMP   R1, #10
        BLE   loop
        SVC   #4              @ prints 55
        MOV   R0, #10
        SVC   #2
        SVC   #1
```

**Then try:** change it to add only the odd numbers (hint: add 2 to the counter). Change `BLE` to `BLT` and predict the answer before you run it.

## 12.2 Count down

**Goal:** print 5, 4, 3, 2, 1 on separate lines, then *Lift-off!*

**Hint:** `SVC #4` changes nothing but R0's meaning, so keep the counter in another register (R4) and copy it to R0 to print.

```
@ Count down from 5, one number per line
        MOV   R4, #5
again:  MOV   R0, R4
        SVC   #4
```

```
        MOV   R0, #10
        SVC   #2
        SUBS  R4, R4, #1
        BNE   again
        LDR   R0, =done
        SVC   #3
        SVC   #1
done:   .asciz "Lift-off!\n"
```

Note the `\n` inside the string: it ends the line, just as `MOV R0, #10` / `SVC #2` does.

## 12.3 A function

**Goal:** print the squares of 1 to 5 using a function `square` that takes its argument in R0 and returns the result in R0.

**Hint:** call it with `BL`, return with `BX LR`. Keep the loop counter in R4 so the function and the SVC calls don't disturb it.

```
@ Squares with a function
        MOV   R4, #1
loop:   MOV   R0, R4
        BL    square
        SVC   #4
        MOV   R0, #' '
        SVC   #2
        ADD   R4, R4, #1
        CMP   R4, #6
        BNE   loop
        MOV   R0, #10
        SVC   #2
        SVC   #1
square: MUL   R0, R0, R0      @ R0 = R0 * R0
        BX    LR
```

It prints `1 4 9 16 25`. Step it and watch LR change at each `BL`.

**Then try:** write `cube` as a function that calls `square`. You'll need `PUSH {R4, LR}` at its start and `POP {R4, PC}` at its end, or it won't find its way back (§9.5).

## 12.4 The largest number in a table

**Goal:** find the largest of six numbers stored with `.word`, one of them negative.

**Hint:** post-indexed loads, `LDR R1, [R4], #4`, fetch a word and move the pointer on in one instruction. A conditional `MOV` avoids a branch.

```
@ Find the largest number in a table
        LDR   R4, =table
        MOV   R5, #6          @ how many
        LDR   R0, [R4], #4    @ best so far = first
        SUB   R5, R5, #1
scan:   LDR   R1, [R4], #4
        CMP   R1, R0
```

```
        MOVGT R0, R1          @ signed compare: use MOVHI for unsigned
        SUBS  R5, R5, #1
        BNE   scan
        SVC   #4              @ prints 99
        SVC   #1
table:  .word 17, -4, 99, 23, 0, 42
```

**Then try:** change `MOVGT` to `MOVHI` and run it again. It prints `-4`: compared as an unsigned number, $-4$ is `0xFFFFFFFC`, the largest value in the table. This is the difference between signed and unsigned conditions (Appendix B).

## 12.5 Shout

**Goal:** read a line of text and print it back in capitals.

**Hint:** `SVC #6` reads into a buffer made with `.space`. Lower-case letters are 32 more than their capitals, and a conditional `SUBLE` makes the change only when the character is between `'a'` and `'z'`.

```
@ Read a line and print it in capitals
        LDR   R0, =prompt
        SVC   #3
        LDR   R0, =buf
        MOV   R1, #64
        SVC   #6
        LDR   R4, =buf
next:   LDRB  R0, [R4], #1     @ load a byte, then move R4 on
        CMP   R0, #0
        BEQ   finish
        CMP   R0, #'a'
        BLT   show
        CMP   R0, #'z'
        SUBLE R0, R0, #32      @ 'a'..'z' -> 'A'..'Z'
show:   SVC   #2
        B     next
finish: MOV   R0, #10
        SVC   #2
        SVC   #1
prompt: .asciz "Say something: "
buf:    .space 64
```

Type `hello world` and it prints `HELLO WORLD`.

## 12.6 Guess the number

**Goal:** a game. The program thinks of 37; you guess, and it says *Too low*, *Too high* or *Got it!*

**Hint:** `SVC #7` reads a number into R0. After one `CMP`, three conditional instructions can choose the message, and the flags survive to the final `BNE` because neither `LDR` nor `SVC` changes them.

```
@ Guess the number (it's 37)
        MOV   R4, #37
ask:    LDR   R0, =q
```

```
        SVC   #3
        SVC   #7                @ R0 = guess
        CMP   R0, R4
        LDRLT R0, =low
        LDRGT R0, =high
        LDREQ R0, =yes
        SVC   #3
        BNE   ask
        SVC   #1
q:      .asciz "Guess: "
low:    .asciz "Too low\n"
high:   .asciz "Too high\n"
yes:    .asciz "Got it!\n"
```

**Then try:** count the guesses in R5 and print the count at the end.

## 12.7 Give the examples real data

The four examples that work on empty memory (§11.8–§11.11) make good exercises: change each one to use a string or table of your own, defined with `.asciz` or `.word` and loaded with `LDR r0, =label`, and check the result on the Memory panel's **DATA** view.

## 12.8 Recipes

| To… | Write |
|---|---|
| Print a number and a newline | `SVC #4` then `MOV R0, #10` / `SVC #2` |
| Print a message | `LDR R0, =msg` / `SVC #3`, with `msg: .asciz "text\n"` |
| Print R0 quickly while debugging | `SVC #0` |
| Stop the program | `SVC #1` |
| Load a constant over 255 | `LDR R0, =1000` (up to 65,535), or `MOVW`/`MOVT` |
| Multiply by 8 | `LSL R0, R0, #3` |
| Divide by 4, signed | `ASR R0, R0, #2` (or `SDIV`) |
| Remainder of R0 ÷ R1 | `SDIV R2, R0, R1` then `MLS R3, R2, R1, R0` |
| Test whether R0 is odd | `TST R0, #1` then `BNE odd` |
| Swap R0 and R1 | `EOR R0, R0, R1` / `EOR R1, R0, R1` / `EOR R0, R0, R1` |
| Absolute value | `CMP R0, #0` / `RSBLT R0, R0, #0` |
| Loop N times | `MOV R4, #N` … `SUBS R4, R4, #1` / `BNE loop` |
| Call a function that calls others | `PUSH {R4, LR}` at its start, `POP {R4, PC}` to return |
| Make room for a buffer | `buf: .space 64` |

| To… | Write |
|---|---|
| Walk through a byte string | `LDRB R0, [R4], #1` |
| Walk through a word table | `LDR R0, [R4], #4`, or `LDR R0, [R4, R5, LSL #2]` with an index in R5 |

CHAPTER 13

# Troubleshooting and FAQ

## 13.1 Assembling

**RUN and STEP stay dim.** Nothing has been assembled, or the last assembly failed. Tap **ASSEMBLE** and read the console. On Android they are also dim while the program waits for input.

**"Unknown or unsupported mnemonic".** Check the spelling. If it's spelt right, the instruction isn't one Pocket ARM supports (Appendix B lists what is). Common ones that aren't: `CLZ`, `UBFX`, `BFI`, `LDRD`/`STRD`, `MRS`/`MSR`, `SWP`, and anything Thumb or floating point. `.equ`, `.set` and `.include` aren't supported either: use the number itself, or `LDR Rd, =value`.

**"Unknown branch target" when the label is right there.** A label needs a colon where it's defined (`loop:`), but not where it's used (`B loop`). Labels can't start with a digit.

**"cannot be encoded".** The immediate doesn't fit in an instruction (§5.5). Use `LDR Rd, =value` for anything up to 65,535, or `MOVW` and `MOVT`.

**My program assembled but does nothing.** Did you tap **ASSEMBLE** after your last edit? The simulator runs the last *assembled* program, not what's in the editor.

**The line numbers in the editor don't line up with my code.** A long line wrapped (§5.1). The numbers in error messages always refer to lines of your text.

## 13.2 Running

**The program never stops.** Tap **STOP**, then look at where PC is (§9.4). Usual causes: a loop whose counter never reaches the value the branch tests; a `B` back to the start instead of `SVC #1`; or `BX LR` after LR was overwritten.

**It stops with "end of program" before doing everything.** PC ran off the end of the code, or into a gap. This also happens if a branch goes to a data label: data isn't code.

**After the program finishes, STEP starts it again from the beginning.** That's deliberate (§9.3). To run it again from where a breakpoint stopped it, don't tap **RESET**; just tap **RUN** or **STEP**.

**My breakpoint doesn't stop the program.** Breakpoints on the very first instruction don't fire (§6.3), and **ASSEMBLE** clears all breakpoints. Set them again after assembling.

**The console says "Breakpoint set at 0x1800" but I set it at 0x18.** A display bug in the iPhone and iPad version 1.0.3 (§6.3). The breakpoint is in the right place.

**Nothing appears in the console when I print.** Output appears a line at a time: print a newline (`MOV R0, #10` / `SVC #2`) or put `\n` at the end of your strings (§8.3). Also check you are looking at the right tab: the output is on **CONSOLE**.

**SVC #0 printed a huge number.** On iPhone and iPad, `SVC #0` shows R0 as an unsigned number, so −1 is shown as `4294967295`. Use `SVC #4` to print signed numbers.

**CLR doesn't clear the console on Android.** A bug in Android version 1.0.3 (§8.1). Use **CLEAR** in the toolbar (which also unloads the program), or just carry on: the console keeps only the last 200 lines.

## 13.3 Memory panel

**I tapped DATA but I'm still looking at the code.** On iPhone and iPad 1.0.3, tap the segment button a second time (§7.5).

**The rows look jumbled.** Upright phones are a little too narrow for a full row (§7.2). Turn the phone sideways.

## 13.4 Files

**Where are my saved programs?** In the **LIBRARY** on the **SOURCE** tab. On iPhone and iPad they are also in the Files app under **On My iPhone › PocketARM**; on Android they are in the app's private storage (§10.1).

**I lost my program when the app closed.** The editor isn't saved automatically. Save with **NAME** (§10.2).

**I saved over an example by mistake.** Saving replaces a file of the same name without asking. Reinstall the app to get the original examples back (§10.1), or copy the program from Chapter 11 of this guide.

**How do I delete a program?** On iPhone and iPad, in the Files app. On Android there's no way to delete a single program in version 1.0.3; save over it with something else, or clear the app's storage (which removes all your programs).

**How do I get a program from my computer into the app?** On iPhone and iPad, through the Files app or the Finder (§10.4). On Android, by copy and paste (§10.5).

**I opened a .s file from the Files app and the editor didn't change.** Opening files from outside the app doesn't load them in version 1.0.3 (§10.4). Copy the file into **On My iPhone › PocketARM** and open it from the **LIBRARY**.

## 13.5 Other questions

**Is this the same as the ARM in my phone?** It's the same 32-bit ARM ("A32") instruction set, as used by ARMv7 processors and by 32-bit programs on many ARMv8 ones. Today's phones run 64-bit code (AArch64), which has different registers and instructions, but the ideas in this guide carry straight across.

**Can I run code written for the Raspberry Pi or a course that uses GNU `as`?** Usually, after small changes: replace Linux system calls (`SVC #0` with a number in R7) with Pocket ARM's calls (§8.3), remove `.equ` and `.include`, and change `.word label` to `LDR Rd, =label`. Directives such as `.text`, `.data` and `.global` can stay.

**Does the app need the internet, or collect data?** No. The simulator runs entirely on your device. Only the buttons on the **MORE** tab open web pages or the store.

**How do I report a bug or suggest a feature?** Use **Email Feedback** on the **MORE** tab, or write to `info@pymblesoftware.com`. Say which device and version you have, and include the program if you can.

APPENDIX A

# iPhone, iPad and Android Compared

The iPhone and iPad version (1.0.3) and the Android version (1.0.3) share their assembler, simulator, example programs, layout, colours and wording: a program assembles to the same machine code and runs to the same result on both. This table lists every difference a user is likely to notice. Each was checked in the apps' source code and, where it shows on screen, in the screenshots in this guide.

| Feature | iPhone and iPad | Android |
|---|---|---|
| Minimum system | iOS / iPadOS 26.2 | Android 8.0 (API 26) |
| Name under the icon | **PocketARM** | **Pocket ARM** |
| Four-pane layout | iPad, always; the largest iPhones when sideways | Any window 600 dp or wider: tablets, and most phones when sideways |
| **MORE** in the four-pane layout | ··· button at the right of the toolbar opens a popover | Not available; turn the phone upright |
| App name in the toolbar | **POCKET ARM — ARMv7 SIMULATOR** on iPad; no room for it on iPhone | Never shown |
| Toolbar labels that don't fit | Cut short (**ASSEMB...**) | Shrunk to fit |
| Register layout | Two columns; PC always on screen on iPhone | One column; scroll down for SP, LR and PC |
| Tap a register | Cycles HEX → DEC → BIN | Nothing; use the buttons |
| **RUN**, **STEP**, **+N** while waiting for input | Lit; tapping prints *Waiting for input...* | Dimmed |
| **+N** prompt | Sheet with a number pad, **CANCEL** / **STEP** | Dialog (digits only, up to six), **CANCEL** / **STEP** |
| Run speed | About 200 instructions a second | Faster (up to about three times) |
| *Breakpoint set/cleared* message | Address padded wrongly: *0x1800* for `0x18` (a bug, §6.3) | *0x0018* |
| *Breakpoint hit* message | *0x18* | *0x0018* |
| Breakpoint rows in DISASM | Red dot, light red tint | Red dot, light red tint, red bar at the left edge |

| Feature | iPhone and iPad | Android |
|---|---|---|
| Memory **TEXT**/**DATA**/**STACK** buttons | Rows lag one tap behind; tap twice (a bug, §7.5) | Work first time |
| Console **CLR** | Clears the console | Does nothing (a bug, §8.1) |
| `SVC #0` output | Unsigned decimal, short hex: `R0 = 21 (0x15)` | Signed decimal, 8-digit hex: `R0 = 21 (0x00000015)` |
| Load fault message | *Memory read fault at 0x…* | *Memory fault at 0x…* |
| Console notes for files | None | *Loaded: name.s*, *Saved: name.s* |
| **NAME** | Alert titled **Name Program**, **Cancel** / **Save** | Sheet titled **NAME PROGRAM**, **CANCEL** / **SAVE** |
| **LIBRARY** | Sheet titled **Library**, **Cancel** | Full-screen panel, **CLOSE**, `.s` shown beside each name |
| Where programs are kept | Documents folder, visible in Files as **On My iPhone › PocketARM** | Private app storage |
| Bringing programs in | Files app, Finder/iTunes file sharing, or copy and paste (§10.4) | Copy and paste only (§10.5) |
| Sending programs out | Share or copy from the Files app; Finder; copy and paste | Copy and paste only |
| Deleting or renaming programs | In the Files app | Not possible (save over, or clear the app's storage) |
| Opening `.s`/`.asm` files from other apps | Offered, but doesn't load the program in 1.0.3 (§10.4) | Not offered |
| **Email Feedback** | Apple Mail sheet; nothing happens if Mail isn't set up | Choice of email apps; *No email app found* if none |
| **Rate**, **Share**, **More Apps** | App Store | Google Play |
| Languages | English and Spanish | English and Spanish |
| Orientation | iPhone: upright and sideways (not upside down); iPad: any | Any |

APPENDIX B

# Instruction Set Reference

This appendix lists every instruction and directive Pocket ARM's assembler accepts, with the operand forms it understands. In the tables, `Rd` is the destination register, `Rn` and `Rm` are source registers, `Rs` holds a shift amount, `Op2` is a flexible second operand (§5.6), and `{S}` and `{cond}` mark where the optional suffixes go.

## B.1 Condition codes

Any instruction can take a condition suffix, which makes it run only when the flags match. The first hex digit of the encoding (§6.2) is the condition's number.

| Suffix | Meaning | Flags tested | Code |
|---|---|---|---|
| `EQ` | Equal | Z = 1 | 0 |
| `NE` | Not equal | Z = 0 | 1 |
| `CS` or `HS` | Carry set / unsigned higher or same | C = 1 | 2 |
| `CC` or `LO` | Carry clear / unsigned lower | C = 0 | 3 |
| `MI` | Minus, negative | N = 1 | 4 |
| `PL` | Plus, positive or zero | N = 0 | 5 |
| `VS` | Overflow | V = 1 | 6 |
| `VC` | No overflow | V = 0 | 7 |
| `HI` | Unsigned higher | C = 1 and Z = 0 | 8 |
| `LS` | Unsigned lower or same | C = 0 or Z = 1 | 9 |
| `GE` | Signed greater than or equal | N = V | A |
| `LT` | Signed less than | N ≠ V | B |
| `GT` | Signed greater than | Z = 0 and N = V | C |
| `LE` | Signed less than or equal | Z = 1 or N ≠ V | D |
| `AL` | Always (the default) | none | E |

After `CMP Ra, Rb`: use `GT`, `GE`, `LT`, `LE` when the numbers are signed, and `HI`, `HS`, `LO`, `LS` when they are unsigned (addresses, characters, sizes).

**Suffix order.** The `S` suffix and the condition can be written either way round: `ADDSEQ` and `ADDEQS` are the same. Mnemonics are matched longest first, so `BLS` is `B` + `LS` and `BLE` is `B` + `LE`, while `BL` alone is branch with link.

## B.2 Data processing

| Instruction | Forms | Operation | `S` allowed |
|---|---|---|---|
| `MOV` | `MOV Rd, Op2` | Rd = Op2 | yes |
| `MVN` | `MVN Rd, Op2` | Rd = NOT Op2 | yes |
| `ADD` | `ADD Rd, Rn, Op2` · `ADD Rd, Op2` | Rd = Rn + Op2 | yes |
| `ADC` | as `ADD` | Rd = Rn + Op2 + C | yes |
| `SUB` | as `ADD` | Rd = Rn − Op2 | yes |
| `SBC` | as `ADD` | Rd = Rn − Op2 − NOT C | yes |
| `RSB` | as `ADD` | Rd = Op2 − Rn | yes |
| `RSC` | as `ADD` | Rd = Op2 − Rn − NOT C | yes |
| `AND` | as `ADD` | Rd = Rn AND Op2 | yes |
| `ORR` | as `ADD` | Rd = Rn OR Op2 | yes |
| `EOR` | as `ADD` | Rd = Rn XOR Op2 | yes |
| `BIC` | as `ADD` | Rd = Rn AND NOT Op2 | yes |
| `CMP` | `CMP Rn, Op2` | Flags from Rn − Op2 | always sets flags |
| `CMN` | `CMN Rn, Op2` | Flags from Rn + Op2 | always sets flags |
| `TST` | `TST Rn, Op2` | Flags from Rn AND Op2 | always sets flags |
| `TEQ` | `TEQ Rn, Op2` | Flags from Rn XOR Op2 | always sets flags |

With `S`, arithmetic instructions set N, Z, C and V; logical ones (`MOV`, `MVN`, `AND`, `ORR`, `EOR`, `BIC`, `TST`, `TEQ`) set N and Z, and set C from the shifter (the last bit shifted out), leaving V alone.

## B.3 Shifts

| Instruction | Forms | Operation |
|---|---|---|
| `LSL` | `LSL Rd, Rm, #n` (0–31) · `LSL Rd, Rm, Rs` · `LSL Rd, #n` | Logical shift left |
| `LSR` | as `LSL`, `#n` 1–32 | Logical shift right (zeros in at the top) |
| `ASR` | as `LSL`, `#n` 1–32 | Arithmetic shift right (copies of the sign bit in at the top) |
| `ROR` | as `LSL`, `#n` 1–31 | Rotate right |
| `RRX` | `RRX Rd, Rm` | Rotate right one bit through the carry flag |

All take `S`. They are assembled as `MOV` with a shifted operand, so they also appear inside Op2: `ADD R0, R1, R2, LSL #2`.

## B.4 Multiply and divide

All operands must be registers.

| Instruction | Forms | Operation | `S` allowed |
|---|---|---|---|
| `MUL` | `MUL Rd, Rn, Rm` · `MUL Rd, Rn` | Rd = Rn × Rm (low 32 bits); the short form is Rd = Rn × Rd | yes |
| `MLA` | `MLA Rd, Rn, Rm, Ra` | Rd = Rn × Rm + Ra | yes |
| `MLS` | `MLS Rd, Rn, Rm, Ra` | Rd = Ra − Rn × Rm | no |
| `UMULL` | `UMULL RdLo, RdHi, Rn, Rm` | 64-bit unsigned product | yes |
| `SMULL` | `SMULL RdLo, RdHi, Rn, Rm` | 64-bit signed product | yes |
| `UMLAL` | `UMLAL RdLo, RdHi, Rn, Rm` | Add 64-bit unsigned product to RdHi:RdLo | yes |
| `SMLAL` | `SMLAL RdLo, RdHi, Rn, Rm` | Add 64-bit signed product to RdHi:RdLo | yes |

| Instruction | Forms | Operation | S allowed |
|---|---|---|---|
| SDIV | SDIV Rd, Rn, Rm · SDIV Rd, Rm | Rd = Rn ÷ Rm, signed, rounded towards zero; the short form is Rd = Rd ÷ Rm | no |
| UDIV | as SDIV | Unsigned division | no |

Division by zero gives 0.

## B.5 Moving constants and addresses

| Instruction | Forms | Operation |
|---|---|---|
| MOVW | MOVW Rd, #imm16 | Rd = imm16 (0–65,535) |
| MOVT | MOVT Rd, #imm16 | Top 16 bits of Rd = imm16; bottom 16 bits kept |
| LDR = | LDR Rd, =label · LDR Rd, = number | Rd = the address or number (assembled as MOVW; must fit in 16 bits) |
| ADR | ADR Rd, label | Rd = the label's address (assembled as MOVW) |

## B.6 Branches

| Instruction | Forms | Operation |
|---|---|---|
| B | B label | Jump |
| BL | BL label | LR = address of the next instruction, then jump |
| BX | BX Rm | Jump to the address in Rm (bit 0 ignored) |
| BLX | BLX Rm | LR = address of the next instruction, then jump to the address in Rm |

All take conditions: BEQ, BLNE, BXGT and so on.

## B.7 Loads and stores

| Instruction | Operation | Offset range |
|---|---|---|
| `LDR Rt, addr` | Load a 32-bit word | ±4095, or a register, optionally shifted |
| `STR Rt, addr` | Store a 32-bit word | as `LDR` |
| `LDRB Rt, addr` | Load a byte, zero-extended | as `LDR` |
| `STRB Rt, addr` | Store the low byte | as `LDR` |
| `LDRH Rt, addr` | Load a halfword (16 bits), zero-extended | ±255, or a register (no shift) |
| `STRH Rt, addr` | Store the low halfword | as `LDRH` |
| `LDRSB Rt, addr` | Load a byte, sign-extended | as `LDRH` |
| `LDRSH Rt, addr` | Load a halfword, sign-extended | as `LDRH` |

Address forms (`addr`):

| Form | Address | Base register afterwards |
|---|---|---|
| `[Rn]` | Rn | unchanged |
| `[Rn, #off]` | Rn + off (off may be negative) | unchanged |
| `[Rn, Rm]` · `[Rn, -Rm]` | Rn ± Rm | unchanged |
| `[Rn, Rm, LSL #n]` | Rn + (Rm shifted); also `LSR`, `ASR`, `ROR`, `RRX` (word and byte only) | unchanged |
| `[Rn, #off]!` (and the register forms with `!`) | Rn + off | Rn = Rn + off |
| `[Rn], #off` · `[Rn], Rm` · `[Rn], Rm, LSL #n` | Rn | Rn = Rn + off (or Rm) |

## B.8 Multiple loads and stores

| Instruction | Forms | Operation |
|---|---|---|
| `PUSH` | `PUSH {reglist}` | Store the registers below SP, lowest register at the lowest address; SP decreases |

| Instruction | Forms | Operation |
|---|---|---|
| `POP` | `POP {reglist}` | Load the registers from SP upwards; SP increases. `POP {…, PC}` returns |
| `LDM` / `STM` | `LDMIA Rn{!}, {reglist}` and so on | Load or store several registers at Rn |

A register list is a comma-separated list in braces, with ranges: `{R4-R7, LR}`. `!` after the base register writes the final address back. The addressing modes are:

| Suffix | Meaning | Load alias | Store alias |
|---|---|---|---|
| `IA` (or none) | Increment after | `LDMFD` | `STMEA` |
| `IB` | Increment before | `LDMED` | `STMFA` |
| `DA` | Decrement after | `LDMFA` | `STMED` |
| `DB` | Decrement before | `LDMEA` | `STMFD` |

`PUSH {list}` is `STMDB SP!, {list}` and `POP {list}` is `LDMIA SP!, {list}`.

## B.9 Other instructions

| Instruction | Forms | Operation |
|---|---|---|
| `SVC` (or `SWI`) | `SVC #n` | Supervisor call: print, read or exit (§8.3) |
| `NOP` | `NOP` | Do nothing |

## B.10 Directives

| Directive | Operands | Effect |
|---|---|---|
| `.ascii` | `"text"`, … | String bytes, no terminator |
| `.asciz`, `.string` | `"text"`, … | String bytes, each followed by a zero byte |
| `.byte`, `.dcb` | numbers (−128 to 255), characters, strings | Bytes |

| Directive | Operands | Effect |
|---|---|---|
| `.word`, `.dcd` | numbers | 32-bit little-endian words, aligned to 4 |
| `.space`, `.skip`, `.zero` | size (0–16,384) | Zero bytes |
| `.align` | | Pad the data area to a multiple of 4 |
| `.text`, `.data`, `.section`, `.global`, `.globl`, `.arm`, `.syntax`, `AREA`, `ENTRY`, `END`, `EXPORT`, `IMPORT`, `PROC`, `ENDP` | anything | Ignored |

All data directives add to the data area at `0x2000` (§5.8); the dot is optional.

## B.11 Not supported

Thumb and Thumb-2 instructions, `IT` blocks, `CLZ`, `REV`, bit-field instructions (`BFI`, `BFC`, `UBFX`, `SBFX`), saturating and SIMD arithmetic, `LDRD`/`STRD`, exclusive loads and stores, `SWP`, `MRS`/`MSR`, coprocessor and floating-point instructions, and the directives `.equ`, `.set`, `.include`, `.macro`, `.ltorg` and `.word label`.

APPENDIX C

# Quick Reference and Glossary

## C.1 Quick reference

**Toolbar**

| Button | Does |
|---|---|
| **ASSEMBLE** | Assemble the editor and load the program; clears breakpoints |
| **RUN / STOP** | Run at full speed / stop |
| **STEP / +N** | Run one / N instructions |
| **RESET** | Back to the start, keeping the program and breakpoints |
| **CLEAR** | Unload the program, breakpoints and console messages |

**Memory map**

| Address | Contents |
|---|---|
| `0x0000` | Code, 4 bytes per instruction |
| `0x2000` | Data from `.ascii`, `.asciz`, `.byte`, `.word`, `.space` |
| `0x8000` | Initial SP; the stack grows down from here |
| `0xFFFF` | Last byte of memory |

**Supervisor calls**

| Call | Does |
|---|---|
| `SVC #0` | Print `SVC 0: R0 = n (0xh)` |
| `SVC #1` | Exit |
| `SVC #2` | Print character R0 |

| Call | Does |
|---|---|
| `SVC #3` | Print zero-terminated string at R0 |
| `SVC #4` | Print R0 as signed decimal |
| `SVC #5` | Read a character into R0 |
| `SVC #6` | Read a line into the buffer at R0 (size R1); length in R0 |
| `SVC #7` | Read a number into R0 |

### Program skeleton

```
@ name.s - what it does
        LDR   R0, =msg
        SVC   #3              @ print a message
        BL    work            @ call a function
        SVC   #1              @ exit

work:   PUSH  {R4, LR}
        @ ... the work ...
        POP   {R4, PC}        @ return

msg:    .asciz "Hello\n"
buf:    .space 64
```

### Colours

| Colour | On | Means |
|---|---|---|
| Amber | Registers, flags | Changed by the last instruction |
| Blue | Registers, disassembly, memory | PC and the instruction it points to |
| Red | Disassembly | Breakpoint |
| Red | Editor line numbers, console | Assembly or runtime error |

## C.2 Glossary

**Address** The number of a byte in memory, from `0x0000` to `0xFFFF` in Pocket ARM.

**Assembler** The program that turns assembly language text into machine code. Pocket ARM's runs when you tap **ASSEMBLE**.

**Assembly language** A human-readable form of machine code: one line per instruction, with names instead of numbers.

**Breakpoint** A mark on an instruction that makes a running program stop just before it (§6.3).

**Byte** Eight bits; a number from 0 to 255. Memory is a row of bytes.

**Carry flag (C)** Set when an addition carries out of the top bit, or a subtraction does *not* need to borrow; also receives the last bit shifted out.

**Condition code** A suffix such as `EQ` or `GT` that makes an instruction run only when the flags match (Appendix B).

**CPSR** Current Program Status Register: the flags N, Z, C, V (and I, F, T).

**Data area** The part of memory from `0x2000` where the assembler puts strings and tables.

**Directive** A line that tells the assembler to do something other than produce an instruction, such as `.asciz` or `.word`.

**Disassembly** A listing of the machine code alongside the instruction each word represents (Chapter 6).

**Encoding** The 32-bit number an instruction is assembled into.

**Flags** Single bits in the CPSR, set by comparisons and `S` instructions, tested by conditions.

**Hexadecimal (hex)** Base 16, written with `0x`; each hex digit is four bits.

**Immediate** A constant written into the instruction itself, with `#`.

**Label** A name for an address, written `name:` at the start of a line.

**Link register (LR, R14)** Holds the return address after `BL`.

**Little-endian** Storing the least significant byte of a word at the lowest address.

**Machine code** The numbers the processor actually executes.

**Mnemonic** The name of an instruction, such as `ADD` or `LDR`.

**Negative flag (N)** A copy of the top bit of the result: set if the result is negative when read as a signed number.

**Operand2** The flexible second operand of a data-processing instruction: an immediate, a register, or a shifted register.

**Overflow flag (V)** Set when a signed addition or subtraction gives a result too big or too small to fit in 32 bits.

**Program counter (PC, R15)** The address of the next instruction to run.

**Register** One of sixteen 32-bit storage slots inside the processor, R0 to R15.

**Simulator** A program that imitates a processor. Pocket ARM's runs the assembled machine code one instruction at a time.

**Stack** Memory used last-in first-out for saving registers, addressed by SP; it grows downwards from `0x8000`.

**Stack pointer (SP, R13)** The address of the last item pushed onto the stack.

**SVC** Supervisor call: the instruction a program uses to ask for a service; in Pocket ARM, to print, read or exit (§8.3).

**Two's complement** The usual way of storing signed numbers: $-1$ is `0xFFFFFFFF`, $-2$ is `0xFFFFFFFE`, and so on.

**Word** 32 bits, four bytes: the size of a register and of an instruction.

**Zero flag (Z)** Set when a result is zero, which after `CMP` means the two values were equal.

# Index


## A

ADD, 11, 18, 19, 20, 49, 61, 62, 69
ADR, 13, 21, 25, 63
.align, 21, 66
AND, 11, 45, 47, 61
AND, 16
.ascii, 21, 27, 65, 67
.asciz, 18, 21, 27, 48, 53, 65, 67, 69
.asciz, 28
Assembler, 1, 2, 3, 9, 10, 20, 22, 46, 50, 58, 60, 69

## B

B, 11, 63
BIC, 61
BIC, 16
BL, 11, 16, 23, 29, 40, 45, 47, 48, 51, 61, 63, 69
BNE, 12, 18, 25, 40, 45, 52, 53
Breakpoint, 1, 3, 7, 17, 24, 25, 31, 37, 38, 39, 47, 55, 67
Bugs in 1.0.3
    breakpoint message, 26, 56, 58
    CLR on Android, 31, 56, 59
    memory segment buttons, 30
    opening files, 44
BX, 11, 47, 48, 51, 55, 63
BX, 40
.byte, 21, 27, 65, 67

## C

CMP, 11, 12, 15, 39, 45, 52, 53, 61, 69
CMP, 40
Comment, 6, 18, 25
Condition codes, 12, 14, 25, 39, 52, 60, 63, 69
Console, 1, 2, 4, 5, 16, 22, 25, 31, 33, 34, 37, 38, 39, 42, 45, 46, 55, 56, 59, 67
Copy and paste, 44, 56, 59
CPSR, 1, 12, 13, 14, 69

## D

Data area, 21, 25, 28, 66
Directive, 18, 19, 21, 23, 25, 27, 60, 66
Disassembly, 1, 17, 19, 24, 26, 29, 40, 46, 58, 68

## E

Email Feedback, 8, 57, 59
Encoding, 24, 25, 29, 46, 60
EOR, 11, 53, 61
Example programs
    binary_search, 41, 45
    bit_counting, 41, 45
    bubble_sort, 41, 45
    caesar_cipher, 41, 45, 48
    factorial, 41, 45
    fibonacci, 4, 5, 14, 18, 24, 25, 29, 39, 41, 45, 46
    gcd, 41, 45, 47
    hello, 18, 22, 28, 30, 32, 41, 45
    input, 34, 41, 45
    new_instructions_demo, 41, 45
    string_length, 41, 45

## F

Files app, 41, 43, 44, 56, 59
Flags, 1, 2, 12, 14, 15, 16, 39, 40, 52, 60, 61, 68, 69
    C (carry), 12, 60, 69
    N (negative), 12, 19, 69
    V (overflow), 12, 60, 69
    Z (zero), 12, 21, 28, 66, 69
Four-pane layout, 2, 7, 58
Function, 11, 12, 16, 29, 40, 45, 47, 48, 51, 53

## H

Hexadecimal, 1, 10, 14, 16, 19, 24, 25, 28, 33, 35, 46, 59, 60, 69

## I

Immediate, 11, 20, 23, 46, 55, 69
Input bar, 2, 34

## L

Label, 3, 5, 11, 13, 18, 19, 21, 22, 25, 40, 53, 55, 57, 58, 63, 66
LDM, 25, 65
LDR =, 16, 21, 25, 46, 48, 53, 55

LDR, 11, 13, 16, 18, 20, 21, 22, 23, 25, 45, 46, 47, 48, 51, 52, 53, 55, 57, 63, 64, 69
LDR =, 23
LDRB, 45, 47, 54, 64
LDRH, 23, 64
Library, 3, 4, 17, 22, 32, 41, 42, 43, 44, 45, 50, 56, 59
Link register, 11, 12, 15, 19, 23, 25, 29, 40, 47, 48, 51, 53, 55, 58, 63, 65, 69
Little-endian, 13, 21, 27, 29, 66

M

Machine code, 1, 12, 24, 25, 29, 58, 68
Memory panel, 1, 5, 25, 39, 46, 47, 48
MLA, 23, 62
MLS, 45, 47, 53, 62
More tab, 2, 5, 7, 8, 57, 58
MOV, 11, 16, 20, 25, 45, 46, 47, 48, 51, 53, 56, 61, 62
MOV, 23
MOVT, 53, 55, 63
MOVW, 20, 21, 23, 25, 53, 55, 63
MUL, 23, 45, 47, 62
MVN, 20, 25, 46, 61

N

NAME button, 17, 18, 41, 44, 50, 56, 59
NOP, 65

O

Operand2, 20, 69
ORR, 11, 61
ORR, 16

P

POP, 12, 25, 29, 40, 45, 47, 51, 53, 65
Program counter, 6, 11, 12, 14, 15, 19, 24, 25, 28, 31, 38, 40, 44, 47, 51, 53, 55, 58, 65, 68, 69
PUSH, 12, 25, 29, 40, 45, 47, 51, 53, 64
PUSH, 23

R

Recursion, 7, 29, 47
Register, 1, 2, 3, 4, 6, 9, 11, 12, 14, 15, 16, 19, 20, 21, 23, 25, 29, 33, 37, 38, 40, 47, 50, 57, 58, 60, 62, 64, 69
Registers panel, 1, 4, 6, 11, 14, 16, 40, 46, 47
Runtime error, 13, 31, 35, 38, 68

S

SDIV, 13, 45, 47, 53, 63
Shifts, 20, 23, 25, 45, 47, 49, 53, 62, 64
Simulator, 1, 2, 3, 5, 13, 18, 24, 25, 31, 35, 37, 39, 50, 55, 57, 58
.space, 21, 27, 52, 53, 66, 67
Spanish, 9, 59
Stack, 1, 2, 4, 11, 12, 27, 28, 47, 67, 69
Stack pointer, 4, 11, 12, 15, 16, 19, 25, 27, 29, 37, 47, 58, 64, 67, 69
Status bar, 5, 35, 38
Step count, 5, 31, 39, 45
STM, 25, 65
STR, 11, 23, 45, 64
STRB, 64
SUB, 11, 61
SUB, 40
SUBS, 12, 40, 45, 46, 53
SUBS, 14, 24
Supervisor call, 5, 11, 13, 16, 25, 31, 32, 34, 36, 38, 40, 45, 46, 47, 48, 50, 51, 52, 53, 55, 57, 59, 65, 67, 69

T

Text area, 17, 21, 29
Toolbar, 2, 5, 7, 16, 31, 37, 56, 58
  ASSEMBLE, 1, 2, 3, 5, 16, 18, 22, 26, 37, 40, 43, 45, 55, 67, 68
  CLEAR, 2, 16, 26, 31, 35, 37, 56, 67
  +N, 2, 5, 16, 37, 38, 39, 58, 67
  RESET, 2, 6, 16, 26, 35, 37, 39, 40, 55, 67
  RUN, 2, 5, 16, 25, 35, 36, 37, 38, 39, 55, 58, 67
  STEP, 2, 5, 16, 25, 35, 36, 37, 38, 39, 47, 55, 58, 67
  STOP, 2, 5, 31, 37, 39, 40, 55, 67
TST, 47, 53, 61
Two's complement, 11, 15

U

UDIV, 13, 63
UMULL, 23, 62

W

.word, 18, 21, 25, 27, 47, 51, 53, 57, 66, 67, 69




## ABOUT THE PUBLISHER

# Pymble Software

Pymble Software is a Sydney software company, in business since 2006, led by a developer with 35 years in the industry. We consult on software projects (apps, payment gateways, banking and security), make games and tools for iPhone and Android, and write about what we've learned.

Our apps include Pocket CPM, Pocket ARM, Web3 Inspector, iRelayChat, Shieldkeep, Orbital Fury, Noir Descent, Rescue Rapunzel, Plate Panic and Ticker Tape Tycoon.

Find them all, and our books, at pymblesoftware.com.

---

# See what the processor sees.
# One instruction at a time.

*Pocket ARM* is an ARMv7 assembler and simulator that fits in your pocket. Type a program, assemble it, and step through it with every register, flag and byte of memory in view. Set breakpoints, read and print through the console, and watch the stack grow and shrink.

This free companion guide explains every screen of the app on iPhone, iPad and Android, and teaches the ARM assembly language you need to use it. You'll learn how to:

- read registers and the **N**, **Z**, **C** and **V** flags, in hex, decimal and binary;
- write loops, functions and recursion, and print and read with **SVC**;
- debug with **STEP**, **+N** and breakpoints;
- save your programs and move them between devices on both platforms;
- follow all eleven example programs, and solve exercises checked on the app's own simulator.

For students, hobbyists and anyone curious about what happens underneath the code.