PicoMite CSUBs — Writing them and building them with armcfgen.py

This manual is in three parts:

1. **Writing a CSUB** — what a CSUB is, exactly how BASIC passes arguments to it

(integers, floats, strings, arrays and string arrays), the rules you must

follow, and worked examples in both *merge* and *join* form.

2. **Building a CSUB** — turning your C into a pasteable `CSUB … END CSUB` block

with `armcfgen.py`.

3. **Appendix A — The CallTable** — every firmware routine a CSUB can call.

Part 1 — Writing a CSUB

1.1 What a CSUB is

A **CSUB** is a block of compiled machine code embedded directly in a BASIC

program as hexadecimal words:

```
CSUB myfunc INTEGER
  00000000
  B5704B08 0004681B 68006841 ...
END CSUB
```

When BASIC executes `myfunc x`, the interpreter jumps straight into that machine

code. A CSUB therefore lets you write performance-critical or hardware-level

work in C (or assembler) and call it from BASIC like any other subroutine.

Three consequences follow from "it's just embedded machine code", and they shape

everything else in this manual:

- **There is no linker.** A CSUB cannot call firmware functions or C library

functions by name. The *only* way it reaches anything in the firmware is

through the **CallTable** — a table of function pointers the firmware exposes

(see §1.5 and Appendix A).

- **It must be self-contained and position independent.** The blob is pasted into flash wherever your program happens to live, so it can't depend on a fixed load address — the firmware does no relocation. The build handles this for you, including read-only data (see §1.4).
- **It targets Cortex-M0+.** Code built for the M0+ instruction set runs unchanged on both the RP2040 (M0+) and the RP2350 (M33), so one blob works on every PicoMite board.

A CSUB is a *subroutine*: it returns nothing usable to BASIC (its C return value is ignored). It communicates results by **writing back through its arguments**.

(A `CFUNCTION` is the value-returning sibling; this manual concentrates on CSUBs, but the argument rules are identical.)

1.2 How arguments are passed — the golden rule

> ***Every argument is passed to the CSUB as a pointer.***

BASIC never passes a value in a register — it passes the *address* of the argument's storage. Your C function therefore always receives pointers, one per BASIC argument, in order:

```
long long myfunc(long long *a, double *b, unsigned char *c)
/*                ^ INTEGER     ^ FLOAT     ^ STRING           */
```

Because you get the address of the caller's storage, **writing through the pointer changes the BASIC variable in place** — that is how a CSUB returns results. The C return type is always `long long` and is ignored.

Two limits: a CSUB may take at most **10** arguments, and array **bounds are not passed** (see §1.3.4).

1.2.1 Variables vs expressions

The in-place behaviour only works when the caller passes a **variable**:

```
myfunc x          ' x is a variable -> CSUB gets &x, can change it
myfunc 5          ' 5 is an expression -> CSUB gets a pointer to a throwaway
myfunc a+b        '   temporary; any change is lost
```

Always pass a variable for anything the CSUB writes to.

1.3 The five kinds of argument

1.3.1 Integers

A BASIC integer is a **64-bit signed** value. You receive a pointer to it:

```
long long myfunc(long long *a)
{
    *a = *a * 2;        /* doubles the caller's integer */
    return 0;
}

x% = 21 : myfunc x% : Print x%      ' 42
```

> ***Note: 64-bit add/subtract compile inline, but 64-bit multiply/divide need a helper that isn't in the blob (see §1.4). For `*a * 2` GCC uses a shift, so it's fine; a general 64×64 multiply is not.***

1.3.2 Floats

A BASIC float is a **C `double`** (64-bit). You receive a pointer to it:

```
long long myfunc(double *a)
{
    *a = FMul(*a, *a);   /* squares it - note FMul, NOT *  (see §1.4) */
    return 0;
}
```

1.3.3 Strings

A BASIC string is **not** a C string. It is a **length-prefixed** buffer:

```
byte 0      : length (0–255)
bytes 1..N  : the characters (NOT null-terminated)
```

You receive a pointer to byte 0:

```
long long supper(unsigned char *s)   /* uppercase a string in place */
{
    int n = s[0];                    /* byte 0 is the length        */
    for (int i = 1; i <= n; i++)     /* characters are s[1]..s[n]   */
        if (s[i] >= 'a' && s[i] <= 'z') s[i] -= 32;
    return 0;
}

a$ = "hello" : supper a$ : Print a$     ' HELLO
```

To **return** a string, set the length byte and write the characters:

```
s[0] = 3; s[1] = 'a'; s[2] = 'b'; s[3] = 'c';   /* s$ becomes "abc" */
```

Maximum length is 255. If you want a normal null-terminated C string to work with, you must convert it yourself (read `s[0]`, copy `s[1..n]`, append `\0`);

the firmware's internal `CtoM`/`MtoC` helpers are not in the CallTable.

1.3.4 Numeric arrays

An array argument (written `a()` in the call) passes a pointer to the array's

contiguous data — `long long *` for an integer array, `double *` for a float

array:

```
long long isum(long long *a, long long *n, long long *result)
{
    long long s = 0;
    for (int i = 0; i < (int)*n; i++) s += a[i];
    *result = s;
    return 0;
}
```

```
Dim Integer v(4) = (10,20,30,40,50)
Dim Integer total
isum v(), 5, total
Print total            ' 150
```

> ***Array bounds are not passed. The CSUB has no idea how big `a()` is, so pass the element count as a separate argument (as `n` above) — or agree on a fixed size. Reading past the end corrupts memory.***

You can also pass a single element (`a(3)`), which gives a pointer to that one

element's storage.

1.3.5 String arrays

A string array passes a pointer to the **first element**. The elements are laid

out back-to-back, each one a length-prefixed string of fixed width:

```
stride = (declared LENGTH) + 1     bytes per element
       = 256 by default            (LENGTH defaults to 255)
element e starts at  base + e * stride
```

So the CSUB needs both the **element count** and the **stride** (i.e. the

`LENGTH`) to walk the array:

```
long long aupper(unsigned char *s, long long *nn, long long *len)
{
    int n = (int)*nn, stride = (int)*len + 1;   /* LENGTH + 1 */
    for (int e = 0; e < n; e++) {
        unsigned char *el = s + e * stride;     /* this element */
        int L = el[0];                          /* its length byte */
        for (int i = 1; i <= L; i++)
            if (el[i] >= 'a' && el[i] <= 'z') el[i] -= 32;
    }
    return 0;
}
```

```
Dim s$(2) Length 16 = ("one","two","three")
aupper s$(), 3, 16
Print s$(0), s$(1), s$(2)      ' ONE TWO THREE
```

Pass the same `LENGTH` you declared the array with; if you used the default,

the stride is 256.

1.3.6 Summary

BASIC argument	C parameter	Notes
integer `x%`	`long long *`	64-bit; write back to modify
float `x!`	`double *`	C `double`; do maths via CallTable (§1.4)
string `s$`	`unsigned char *`	length-prefixed: `s[0]`=len, `s[1..]`=chars, max 255
integer array `a()`	`long long *`	contiguous; bounds **not** passed
float array `a()`	`double *`	contiguous; bounds **not** passed
string array `s$()`	`unsigned char *`	stride = `LENGTH`+1 (default 256); each element length-prefixed
single element `a(i)`	pointer to that element	
expression `a+b`, literal	pointer to a temporary	read-only in effect

1.4 The rules you must follow

These are not style preferences — break them and the blob crashes or won't build.

1. **No floating-point or 64-bit-multiply C operators.** Cortex-M0+ has no FPU

and no 64-bit multiply, so `a * b`, `a + b`, etc. on `double` (and `*`/`/` on

`long long`) make the compiler emit a libgcc helper (`__aeabi_dadd`,

`__aeabi_dmul`, `__aeabi_lmul`, …) that is **not in the blob**. Do that maths

by calling the firmware instead (§1.5). 32-bit integer maths and 64-bit

add/subtract are fine.

Safety net: if you slip up, the build fails with `undefined reference to

__aeabi_…` rather than producing a broken blob.

2. **Constant data (`.rodata`) is fine.** `const` tables, strings and `double`

literals work: the tool compiles position-independent (text-relative) code and

appends the rodata to the blob, reached PC-relative — so it runs wherever it

lands in flash, with no firmware relocation. (Passing values in through an

argument array is still slightly leaner when you have the choice.)

3. **Pass a variable, not an expression**, for anything you write back (§1.2.1).

4. **At most 10 arguments.** Bundle extras (and any state that must persist between calls) into a parameter array.

5. **Target stays Cortex-M0+** so the blob runs on both chips.

1.5 Calling firmware routines (the CallTable)

Because there's no linker, a CSUB reaches the firmware through the **CallTable**.

The easy way is to include the header and call routines by name:

```
#include "PicoCFunctions.h"
...
*a = FMul(*a, *b);          /* double multiply  */
*a = FAdd(Sine(*a), *b);    /* sin(), add        */
DrawPixel(x, y, RGB_value); /* plot a pixel      */
```

The header locates the CallTable **automatically at runtime** (via the Cortex-M `VTOR` register), so there is **no CallTable argument to pass** and the **same blob runs on every variant and both chips**. Compile with `-I` pointing at the firmware tree so the header is found (Part 2).

The routines you will use most:

Call	Purpose
`FAdd(a,b)` `FSub(a,b)` `FMul(a,b)` `FDiv(a,b)`	`double` + − × ÷
`Sine(x)` `Cosine(x)` `Sqrt(x)` `Atan2(y,x)` `Power(b,e)`	`double` maths
`IntToFloat(i)` `FloatToInt(f)`	convert (FloatToInt rounds)
`DrawPixel(x,y,c)`	plot one pixel (c is a full RGB colour)
`Display_Refresh()`	push direct drawing to a buffered panel
`HRes` `VRes`	current screen size

`DrawPixel` is **format- and version-independent**: you pass a full RGB colour and the firmware packs it for whatever mode is active — so it is the safe way to plot.

The full list is in Appendix A.

> ***Hot loops: each header wrapper re-reads the table base. In a tight loop cache it once — `unsigned int base = BaseAddress;` then use `(base + 0x90)` —***

or, for maximum speed, write directly into the framebuffer via `WriteBuf`

(Appendix A; note its bytes are mode-specific).

1.6 Worked examples

A. Square an integer — no firmware calls, no header

Pure integer work reaches nothing in the firmware, so no header is needed.

```
long long sqr(long long *a)
{
    int v = (int)(*a);          /* 32-bit multiply, avoids __aeabi_lmul */
    *a = (long long)(v * v);
    return 0;
}
```

```
x% = 5 : sqr x% : Print x%      ' 25
```

B. Square a double — header + FMul (merge)

```
#include "PicoCFunctions.h"
long long sqrf(double *a)
{
    *a = FMul(*a, *a);
    return 0;
}
```

```
x! = 1.5 : sqrf x! : Print x!   ' 2.25
```

C. Uppercase a string (merge)

The §1.3.3 `supper` example. Build it and call it:

```
a$ = "hello" : supper a$ : Print a$     ' HELLO
```

D. Several functions, one calling another (merge)

Merge mode puts every function in **one** CSUB so they can call each other:

```
__attribute__((noinline)) static int triple(int x){ return x + x + x; }
long long scale3(long long *a){ *a = (long long)triple((int)*a); return 0; }
```

`scale3` calls `triple`; both end up in a single CSUB block with the call between them resolved. The entry function (`-e scale3`) goes first.

E. A file of independent CSUBs (join)

Join mode emits a **separate** `CSUB … END CSUB` for each function — handy when you keep a library of unrelated CSUBs in one `.c`:

```
long long sqr(long long *a){ int v=(int)*a; *a=(long long)(v*v); return 0; }
long long neg(long long *a){ *a = -*a; return 0; }
```

`armcfgen.py lib.c --compile -m join` produces a `CSUB sqr … ` block and a

`CSUB neg …` block, ready to paste individually. Functions in join mode **cannot**

call each other (each is its own standalone blob).

F. Constant data — `.rodata` (merge)

`const` data just works; the tool builds it position-independent:

```
#include "PicoCFunctions.h"
const char test[] = "12345678";
void main(void)
{
    MMPrintString((char *)test);
}
python armcfgen.py rotest.c --compile -n rotest -e main -I d:\Dropbox\PicoMite\PicoMite
rotest          ' prints 12345678
```

The `"12345678"` string is appended to the blob and reached PC-relative, so it

prints correctly wherever the CSUB lands in flash. Note `main` may be `void` —

the C return value is ignored either way.

Part 2 — Building a CSUB with armcfgen.py

`armcfgen.py` turns compiled ARM code into the `CSUB … END CSUB` text. It can

drive the compiler for you, so you go straight from `.c` to a pasteable block.

2.1 Prerequisites

- **Python 3.8+** — `python --version`
- **pyelftools** — `pip install pyelftools`
- **Arm GNU toolchain** (`arm-none-eabi-gcc`) on your `PATH` — the same one that

builds the firmware. On Windows it is typically at

`C:\Program Files (x86)\Arm GNU Toolchain arm-none-eabi\<version>\bin`.

- The **header** `PicoCFunctions.h` if your source `#includes` it. It ships in

the firmware tree; point at it with `-I` rather than copying it.

(Only `--compile` and the link step use the toolchain; an already-linked `.elf`

needs just Python + pyelftools.)

2.2 Quick start

```
python armcfgen.py sqrf.c --compile -n sqrf -e sqrf -I d:\Dropbox\PicoMite\PicoMite
```

Add `-o sqrf.bas` to write to a file instead of the screen, then copy the

`CSUB … END CSUB` block into your program.

2.3 Command-line reference

```
python armcfgen.py INPUT [INPUT ...] [options]
```

Option	Default	Meaning
`INPUT`	—	A linked `.elf`, a relocatable `.o`, or `.c` source (with `--compile`). Several inputs are linked together.
`-c, --compile`	off	Compile the inputs as C first, with the built-in flags.
`-I, --include DIR`	—	Header search directory for `--compile` (the firmware tree, for `PicoCFunctions.h`). Repeatable.
`-m, --mode {join,merge}`	`merge`	Output style — §2.4.
`-e, --entry NAME`	`main`	Entry symbol: sets the offset word in merge; dropped as a dummy in join.
`-n, --name NAME`	input stem, upper-cased	CSUB name in merge mode.
`-O, --opt LEVEL`	`0`	Optimisation level for `--compile`. `0` is the simple, reliable default; use `s` (or `2`) for compute-heavy CSUBs.
`-o, --output FILE`	stdout	Write to a file.

Built-in compile flags: `-mcpu=cortex-m0plus -mthumb -O0 -ffreestanding

-fno-exceptions -fpie -mpic-data-is-text-relative -msingle-pic-base`. The

text-relative PIC flags are what make the blob self-contained (rodata reached

PC-relative, resolved at link — no GOT, no firmware fixup). The list of functions

found is printed to stderr.

2.4 join vs merge

merge (default) — one CSUB containing every function in address order, with

the entry-offset word pointing at `--entry`. Functions can call each other. Use

for any single-function CSUB and for multi-function drivers.

```
python armcfgen.py driver.c helpers.c --compile -n driver -e main -I
d:\Dropbox\PicoMite\PicoMite
```

The entry function must be **4-byte aligned** — make it the first function (the offset word is word-granular; the tool errors if the entry lands mid-word).

join — a separate `CSUB name … END CSUB` per function, each with its own `00000000` offset; the entry symbol is dropped. Functions cannot call each other, and constant data is rejected.

```
python armcfgen.py lib.elf -m join
```

2.5 Automatic linking

If you pass more than one object, or any input is a relocatable `.o`, `armcfgen.py` links them into one resolved image (gathering `.text.startup`, `.text*` and `.rodata*` into a single `.text` at address 0). This is what makes **merge with internal calls** work: an un-linked `.o` still has the calls between functions stored as unresolved relocations, and GCC puts `main` in a separate `.text.startup` section. Linking fixes the calls and merges the fragments. A fully linked `.elf` is used as-is.

2.6 Output, and the type list you must add

```
CSUB sqrf FLOAT
  00000000
  B5704B08 0004681B 68006841 ...
END CSUB
```

- line 1 — `CSUB`, the name, and the **argument type list you add yourself** (the tool only knows the name). Types are `INTEGER`, `FLOAT`, `STRING`; arrays are written `INTEGER`, `FLOAT`, `STRING` too and passed as `name()`.
- line 2 — the entry-offset word (`00000000` = entry at the first code word).
- then the machine code as little-endian 32-bit words, eight per line.

Match the type list to your call, e.g. a CSUB taking `(double*, long long*)` is declared `CSUB myfunc FLOAT, INTEGER`.

2.7 Using it in BASIC

Paste the block anywhere (conventionally at the end, like a SUB) and call it:

```
Dim Float x! = 3
sqrf x!
Print x!          ' 9
```

No CallTable argument is needed — the blob finds the CallTable itself.

2.8 Troubleshooting

Message	Cause / fix
`arm-none-eabi-gcc: not found`	Toolchain not on PATH (§2.1).
`No module named 'elftools'`	`pip install pyelftools`.
`PicoCFunctions.h: No such file`	Add `-I` pointing at the firmware tree.
`… is not fully linked. Re-run with --compile…`	An un-linked `.o` with unresolved relocations — use `--compile` or a linked `.elf`.
`undefined reference to '__aeabi_…'`	A `double`/64-bit operator slipped in — route the maths through the CallTable (§1.4).
`entry '…' is not 4-byte aligned`	Put the merge entry function first in the source.
`constant data (.rodata) not allowed in JOIN mode`	Remove `const` tables / big literals, or use merge.
`no symbol table`	The object was stripped — rebuild without stripping.

Appendix A — The CallTable

Every routine and data pointer a CSUB can reach, with its slot offset from the table base. With `PicoCFunctions.h` you use the name directly; the offset matters only if you resolve the table by hand. **Slots are append-only** — a given offset always means the same thing across firmware versions, so a compiled CSUB keeps working.

Console, formatting and program control

Name	Offset	Prototype / use
`uSec`	0x00	`void uSec(unsigned long long us)` — busy-wait microseconds
`putConsole`	0x04	`void putConsole(int ch, int flush)` — write a character to the console
`getConsole`	0x08	`int getConsole(void)` — read a character, −1 if none waiting
`MMPrintString`	0x20	`void MMPrintString(char`

		`*s)` — print a null-terminated C string
`IntToStr`	0x24	`void IntToStr(char *dst, long long n, unsigned int base)`
`FloatToStr`	0x70	`void FloatToStr(char *dst, MMFLOAT f, int intDig, int decDig, char pad)`
`CheckAbort`	0x28	`void CheckAbort(void)` — service the interpreter; honours Ctrl-C
`error`	0x58	`void error(char *msg)` — abort the CSUB with a BASIC error (C string)
`SoftReset`	0x54	`void SoftReset(void)` — restart the firmware
`RunBasicSub`	0x74	`void RunBasicSub(char *name)` — call a BASIC SUB (name ends in **two** nulls)

Pins and PIO

Name	Offset	Prototype / use
`ExtCfg`	0x0C	`void ExtCfg(int pin, int cfg, int option)` — configure a pin (`EXT_DIG_IN`, …)
`ExtSet`	0x10	`void ExtSet(int pin, int val)` — drive a digital output
`ExtInp`	0x14	`int ExtInp(int pin)` — read a configured pin
`PinSetBit`	0x18	`void PinSetBit(int pin, unsigned int offset)` — low-level pin register write
`PinRead`	0x1C	`int PinRead(int pin)` — read a pin level
`ExtCurrentConfig`	0x48	`int ExtCurrentConfig[]` — current configuration of each pin
`PIOExecute`	0xD8	`void PIOExecute(int pio, int sm, uint32_t instruction)`

Memory

Name	Offset	Prototype / use
`GetMemory`	0x2C	`void *GetMemory(size_t n)` — allocate from the MMBasic heap (persists)
`GetTempMemory`	0x30	`void *GetTempMemory(int n)` — temporary allocation

		(auto-freed)
FreeMemory	0x34	void FreeMemory(void *p) — free a GetMemory block
ProgFlash	0x5C	int * — base of program memory
CFuncRam	0x7C	int * — persistent static RAM reserved for CFunctions

Maths

Name	Offset	Prototype / use
IntToFloat	0x84	MMFLOAT IntToFloat(long long)
FloatToInt	0x88	long long FloatToInt(MMFLOAT) — rounds (x≥0 ? +0.5 : −0.5)
Sine	0x90	MMFLOAT Sine(MMFLOAT) — sin
FMul	0xA0	MMFLOAT FMul(MMFLOAT,MMFLOAT) — a×b
FAdd	0xA4	MMFLOAT FAdd(MMFLOAT,MMFLOAT) — a+b
FSub	0xA8	MMFLOAT FSub(MMFLOAT,MMFLOAT) — a−b
FDiv	0xAC	MMFLOAT FDiv(MMFLOAT,MMFLOAT) — a÷b
FCmp	0xB0	int FCmp(MMFLOAT,MMFLOAT) — −1 / 0 / 1
LoadFloat	0xB4	MMFLOAT LoadFloat(unsigned long long bits) — reinterpret a bit pattern
IDiv	0xC8	int IDiv(int a, int b) — integer divide
Cosine	0xF4	MMFLOAT cos(MMFLOAT)
Sqrt	0xF8	MMFLOAT sqrt(MMFLOAT)
Atan2	0xFC	MMFLOAT atan2(MMFLOAT y, MMFLOAT x)
Power	0x100	MMFLOAT pow(MMFLOAT base, MMFLOAT exp)

Graphics

Name	Offset	Prototype / use
DrawPixel	0xEC	void DrawPixel(int x, int y, int rgb) — plot one pixel; firmware packs the colour for the active mode (format-independent)

DrawLine	0x40	void DrawLine(int x1,int y1,int x2,int y2,int w,int c)
DrawRectangle	0x38	void DrawRectangle(int x1,int y1,int x2,int y2,int c) — filled (inclusive)
DrawCircle	0x94	void DrawCircle(int x,int y,int r,int w,int c,int fill,MMFLOAT aspect)
DrawTriangle	0x98	void DrawTriangle(int x0,int y0,int x1,int y1,int x2,int y2,int c,int fill)
DrawBitmap	0x3C	void DrawBitmap(int x,int y,int w,int h,int scale,int fg,int bg,unsigned char *bmp)
DrawBuffer	0x68	void DrawBuffer(int x1,int y1,int x2,int y2,char *src) — blit a pixel block in
ReadBuffer	0x6C	void ReadBuffer(int x1,int y1,int x2,int y2,char *dst) — read a pixel block out
ScrollLCD	0x80	void ScrollLCD(int lines, int blank)
FontTable	0x44	const unsigned char *FontTable[] — installed font data
HRes	0x4C	unsigned int — current horizontal resolution
VRes	0x50	unsigned int — current vertical resolution

Framebuffer (direct access)

Name	Offset	Prototype / use
WriteBuf	0xDC	unsigned char * — current **write** framebuffer. Bytes are mode-specific (RGB121 = 4 bpp / 2 px-per-byte; RGB332 = 1 byte/px), so use only when you know your mode. Prefer DrawPixel otherwise.
FrameBuf	0xE0	unsigned char * — frame layer
LayerBuf	0xE4	unsigned char * — overlay layer
DisplayBuf	0xE8	unsigned char * — display layer
Display_Refresh	0xF0	void Display_Refresh(void) — push direct writes to a buffered/SPI panel (no-op on DMA-scanned VGA/HDMI)

Audio

Name	Offset	Prototype / use
`AudioOutput`	0xC4	`void AudioOutput(uint16_t left, uint16_t right)`
`AUDIO_WRAP`	0xCC	`uint16_t` — current audio PWM wrap (full-scale) value

System data and interrupt vectors

Name	Offset	Prototype / use
`Option`	0x8C	`struct option_s *` — the firmware option/settings structure
`g_vartbl`	0x60	the BASIC variable table
`g_varcnt`	0x64	number of variables in the table
`Timer`	0x9C	`unsigned long long Timer(void)` — microsecond timer
`CFuncmSec`	0x78	vector slot for a millisecond-tick CFunction
`CFuncInt1`…`CFuncInt4`	0xB8,0xBC,0xD0,0xD4	vector slots for interrupt CFunctions
`CSubComplete` / `Interrupt`	0xC0	`char` — interrupt-in-progress flag

Slots `0x84`+ assume current firmware. The framebuffer/maths group at `0xDC`–`0x100` was added later; older firmware stops at `PIOExecute` (0xD8).