mmb2csub.py - User Manual

Turning an MMBasic SUB or FUNCTION into a CSUB, without writing any C.

Requires PicoMite firmware 6.03.02b9 or later.

1. What it is for

Your program is too slow. Somewhere inside it is a routine doing the real work, and the interpreter is spending most of its time reading that routine rather than running it. mmb2csub compiles that routine to machine code and puts it back into your program as a CSUB, so the interpreter calls it once instead of interpreting it a million times.

```
python mmb2csub.py myprogram.bas PlotJulia
```

That rewrites myprogram.bas: the original routine is commented out, the CSUB is appended, and your call sites do not change -- MMBasic calls a CSUB exactly as it calls a SUB.

What you can expect:

what the routine does	speedup
loops, array work, integer and float arithmetic	10-35x
transcendental maths (SIN, LOG), string formatting	3-4x
mostly calling the firmware already (graphics, files)	very little

The Julia set demo in examples/ is the top row: 119 seconds interpreted on an RP2040 and 5.9 converted, 86 seconds and 2.5 on an RP2350 - 20x and 35x, from the same file, drawing a byte-identical image on both. The RP2350 gains more because its processor runs compiled code much faster than it runs the interpreter.

The second row is not a disappointment, it is the arithmetic: SIN and STR$ run the same firmware code whether your BASIC calls them or the CSUB does, so only the interpreting overhead goes away. A routine that spends its time inside the firmware has little for this tool to remove.

The result runs on every PicoMite

One converted program runs on both the RP2040 and the RP2350, and on every firmware variant, unchanged. You do not build a version per chip, and you do not have to know what your users have.

That is not a happy accident. The code is compiled for the Cortex-M0+, which the RP2350 also executes; it is position-independent, so it does not care where the firmware loads it; and it finds the firmware's routines through a table it locates at run time rather than at an address baked in when it was built. Nothing in the blob depends on the chip or on the variant. In particular, the generated code never reads the firmware's variable table or its Option structure, the two things whose layout does differ between the chips and variants. (A hand-written CSUB that reads them has to be built for the board it runs on; see the CSUB manual, armcfgen.)

Tested rather than assumed: the same file -- same bytes, same checksum -- produces byte-identical output on a PicoMiteVGA running on an RP2040 and a PicoMiteHDMIWEB running on an RP2350B, across strings, STATICs, recursion, LOCAL arrays and floating point.

So a converted program can be published, posted on the forum or shipped to somebody else exactly as an ordinary .bas file, and it will work on their board. The only requirement is the firmware version.

2. Setting up

mmb2csub is a Python program that drives a C compiler. Nothing is installed and nothing is configured: once the five pieces below are present it works from wherever you unpacked the firmware source.

2.1 What you need

Python 3.6 or later	3.8+ recommended
Arm GNU toolchain	arm-none-eabi-gcc -- the same compiler that builds the firmware
pyelftools	a Python package; armcfgen.py reads the linked ELF with it
The PicoMite firmware source	for PicoCFunctions.h and mmcsub.h
mmb2c.py	the MMBasic-to-C translator
Firmware 6.03.02b9 or later	on the board itself -- see below

Two things you do not need: the Pico SDK, and any ability to build the firmware. mmb2csub compiles a small freestanding blob, not a firmware image.

The firmware version matters, and nothing on your PC can check it. A CSUB reaches the interpreter's own string and maths routines through the CallTable, and the entries it needs were added in 6.03.02b9. On the board:

```
? MM.VER
 6.030209
```

6.030209 is b9; anything lower is too old. 6.03.02b10 or later is worth having if you expect to put a large CSUB in the library: before b10 that route cost 3.4x the blob rather than the blob itself, which is the difference between a 37 KB routine fitting a 100 KB board and not (A.8). Everything else works on b9.

An older firmware does not refuse the CSUB -- the blob is built on your PC and cannot know what it will run on, so the call goes to a table entry that does not exist yet and the board crashes or resets with no message. If a CSUB that built cleanly misbehaves from the very first call, check this first.

pyserial is needed only if you use xsend.py to send programs to the board (section 7).

2.2 Where the pieces have to live

mmb2csub.py finds everything relative to itself, so there are no include paths to set:

```
PicoMite/                    <- the firmware source tree
??? PicoCFunctions.h         <- found automatically
??? user-tools/
    ??? mmb2csub.py          <- you run this
    ??? mmcsub.h             <- found automatically
    ??? armcfgen.py          <- called automatically
```

Keep mmb2csub.py inside user-tools/. Copied elsewhere it can no longer find the header it compiles against.

mmb2c.py is the exception. It is a tool in its own right -- the same translator that, converted to C, runs as a native application under the Fuzix port -- so it is kept in one place rather than copied into every project that uses it. It is looked for in this order:

- 1. --mmb2c PATH on the command line
- 2. the MMB2C_PATH environment variable
- 3. mmb2c.py sitting next to mmb2csub.py in user-tools/
- 4. a couple of default locations

If none matches you get a message naming the places it looked. The simplest answer for most people is (3) -- copy mmb2c.py into user-tools/.

2.3 Windows

Python. Install from python.org or the Microsoft Store, ticking Add Python to PATH. Check:

```
python --version
```

The Arm toolchain. Download the arm-none-eabi toolchain from Arm's Developer site and install it. The installer offers Add path to environment variable -- tick it. If you have already built the firmware you have this already. Check:

```
arm-none-eabi-gcc --version
```

If that says it is not recognised, add the toolchain's bin to your PATH. It is typically:

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

pyelftools:

```
pip install pyelftools
```

mmb2c.py. Copy it into user-tools\, or point at it once per session:

```
set MMB2C_PATH=C:\path\to\mmb2c.py
```

To set it permanently, use Edit environment variables for your account in the Start menu.

If you keep it under WSL rather than on Windows, Windows can read it directly through \\wsl.localhost\<distro>\home\<you>\... -- no copying needed.

2.4 Linux

Python is already present on any current distribution. Check it is 3.6 or later with python3 --version. Use python3 in place of python throughout this manual if your system has no python.

The Arm toolchain and pyelftools, on Debian/Ubuntu/Raspberry Pi OS:

```
sudo apt install gcc-arm-none-eabi python3-pyelftools
```

On Fedora:

```
sudo dnf install arm-none-eabi-gcc-cs arm-none-eabi-newlib python3-pyelftools
```

On Arch:

```
sudo pacman -S arm-none-eabi-gcc python-pyelftools
```

If your distribution's toolchain is old or missing, Arm's own .tar.xz build works anywhere -- unpack it and add its bin to your PATH:

```
export PATH=$PATH:/opt/arm-gnu-toolchain-<version>/bin
```

Put that line in ~/.bashrc to make it permanent.

pip install pyelftools also works, but on distributions that manage Python packages themselves you may need pip install --user pyelftools or a virtual environment.

mmb2c.py. Point at wherever you keep it:

```
export MMB2C_PATH=~/src/FUZIX/Applications/mmb2c/mmb2c.py
```

2.5 Check it works

This converts nothing and writes nothing -- it compiles and links every routine in a program and reports what it found:

```
python mmb2csub.py yourprogram.bas --list
```

A listing means the whole chain works: Python found mmb2c.py, translated your program, called the compiler, and linked the result.

2.6 If it does not

message	cause
cannot find mmb2c.py	see 2.2 -- the message names every place it looked
arm-none-eabi-gcc: not found, or is not recognized	the toolchain is not on your PATH
No module named 'elftools'	pip install pyelftools
mmcsub.h: No such file	mmb2csub.py has been moved out of user-tools/

message	cause
No module named 'serial'	only xsend.py needs this: pip install pyserial

A failure here is always one of these five. Once --list produces a listing the environment is finished with, and every later problem is about your program rather than your setup.

3. The recommended workflow

Step 1 -- make sure there is a routine to convert

This tool converts a SUB or a FUNCTION. If your hot code is not in one, the first job is to put it in one, and how you draw that boundary decides how well the conversion goes -- more than anything else you do later.

The Julia set example began as inline code at the top of the program:

```
For y = 0 To 239
  For x = 0 To 319
    ' ... the escape-time loop ...
  Next x
Next y
```

Nothing there can be converted. Wrapped as a routine, all of it can:

```
Sub PlotJulia(w%, h%, cx!, cy!, maxiter%)
  Local x%, y%
  For y% = 0 To h% - 1
    For x% = 0 To w% - 1
      ' ... the escape-time loop ...
    Next x%
  Next y%
End Sub
```

Four rules for drawing the boundary. They are the same rules, really: give the routine what it needs and let it get on with it.

Wrap the outermost loop, not the innermost. A CSUB is one call, and the whole benefit is the interpreter not reading your code a million times. Put the boundary outside every loop you can, so one call does all the work. A routine called from inside a hot loop saves you almost nothing, because the interpreter is still running the loop.

Pass what it needs as parameters, not as globals. The routine above takes its width, height and constants as arguments; it would have worked equally well reading them from globals, and converted far less comfortably. Every global a routine reaches costs an argument and makes the wrapper bigger, and past ten they have to be gathered into a table. Parameters are free.

Move the variables it only uses itself to LOCAL. A variable that lives in the routine is compiled into the blob and costs nothing. The same variable left global is one more argument, one more line of wrapper, and one more thing that can go wrong. If a loop counter is global because the code used to be inline, make it LOCAL on the way in.

Keep the display and file work outside if you can. A routine that spends its time in PRINT or SAVE IMAGE is running firmware code either way, so there is nothing for the tool to remove -- see the table in section 1. Compute in the routine, report outside it.

None of this is wasted effort if the conversion does not work out, because it is the same shape that makes a program readable: a routine with a clear boundary, taking what it needs and owning its own variables.

Two things that are worth knowing before you carve:

- If the code uses a global that nothing ever DIMs -- common in older programs, where a variable springs into being on first assignment -- declare it at program level while you are there. A.7 explains why this one bites only after conversion.

- A FUNCTION is as convertible as a SUB, numeric or string. You do not have to reshape one into the other.

Step 2 -- find out where the time goes

Run your program with profiling on. You are looking for the routine with the most self time, not the most calls: a routine called 1,500 times that does nothing much matters less than one called 500 times that does everything.

If your profile only gives call counts, treat them as a hint and use step 3 to check the shape of the call graph before deciding.

Step 3 -- ask what can be converted, and what it costs

```
python mmb2csub.py myprogram.bas --list
```

This changes nothing. It compiles and links every routine in the program, so the answer is the toolchain's, not a guess. It takes a minute or two on a large program.

```
Convertible, and not already inside another one's blob:
  routine          args     blob    prog closure  also brings in
  DrawFrame           6     4280     14k       3  Rotate, Project, Clip
  Julia              10     1136      3k       1  -

Also convertible, and worth having if a root above is out of reach:
  routine          args     blob    prog closure  carried by
  Rotate              4      980      3k       1  DrawFrame

Not convertible:
  AskUser        needs mm_input_line, mm_input_next
  Report         parameter 'pt' is a user TYPE
```

The four numbers are the whole decision:

- args -- how many arguments the CSUB will take. Your parameters, plus one for every global the routine reaches. Past ten, the globals stop being separate arguments and travel as a single array of their addresses, so a routine reaching twenty of them still takes only a handful.
- blob -- the machine code.
- prog -- what it costs in program memory, which is what actually has to fit. A CSUB is stored as hex text and as the binary built from it, so this is about 3.4x the blob. Compare it with the Free figure from MEMORY on the board.
- closure -- how many routines travel with it.

Step 4 -- pick ONE routine, as high up as fits

This is the step people get wrong. You do not pick a list of hot routines -- you pick one root, and everything it calls comes with it. The first section of the listing is exactly the routines that are not already inside somebody else's blob.

Converting two routines from the same call tree gives you two CSUBs each carrying its own copy of the shared code. The tool will say so:

```
note: DrawFrame already carries Rotate - converting both duplicates it
```

If the root you want is out of reach -- too many arguments, or more program memory than you have -- drop to the largest routine below it that fits. That is what the second section of the listing is for.

Step 5 -- convert it, keeping everything, and check it works

```
python mmb2csub.py myprogram.bas DrawFrame
```

The default keeps the original routine commented out and the generated C as comments, so you can read what happened. Your previous version is saved as myprogram.bas.bak regardless.

Load it and run it. If it works, go on. If it does not, --dry-run builds and reports without touching your source, and --keep-c leaves the C on disk.

Step 6 -- if it does not fit, make it lean

The comments cost program memory -- the interpreter stores the text.

```
python mmb2csub.py myprogram.bas DrawFrame --lean
```

--lean drops the commented-out original, the embedded C, and the CSUB's own internal comments, and refills the hex eight words to a line. For a small program that is the difference between 7.9 KB of program text and 4.5 KB. Nothing is lost: the .bak has the original and the C regenerates.

--no-c and --no-original do one each if you want to keep the other.

If it still does not fit, put the CSUB in the library, which is what --library is for:

```
python mmb2csub.py myprogram.bas DrawFrame --library mylib.bas --lean
```

The CSUBs and their wrappers go to mylib.bas instead of into your program, and your program keeps only the originals, commented out. A bare filename like that one is written beside your program, not into whatever directory you happened to run the tool from; give a path if you want it somewhere else. The tool prints where it went. Then, on the board:

```
LIBRARY LOAD "mylib.bas"
LOAD "myprogram.bas"
RUN
```

The library holds the binary alone, so the program is left with almost all of program memory to itself. --lean applies to the library file -- there is nothing in your program to make lean. Several routines named in one command all go into the same library file.

LIBRARY LOAD reads the file straight into the library, converting the hex to binary as it goes, so what it costs is the blob and nothing more. A program can do it for itself, as its first statement:

```
LIBRARY LOAD "mylib.bas", 0
```

It installs the library and restarts the program, and on every later run it sees the library is already the right one and does nothing -- so this is a reasonable thing to leave in a program you give somebody.

Step 7 -- prove it gives the same answer

This is not optional. A CSUB that is subtly wrong does not print a helpful message; it hard-faults with no line number, or quietly returns a different number.

- If it draws, SAVE IMAGE before and after and compare the files. The comparison can be done on the board -- see Bas/filecmp.bas.
- If it computes, print the results both ways over the same inputs and diff them. Keep the .bak, which still has the interpreted version.
- Then time it.

Do the correctness check before the timing. A fast wrong answer is not progress.

4. What can be converted

Yes: SUBs and FUNCTIONs, numeric or string; numeric and string scalars, and numeric arrays, as parameters or globals; LOCALs including arrays and strings; STATICs, which the wrapper keeps for you; recursion; IF/FOR/DO/WHILE/SELECT/EXIT; all arithmetic and comparison; CONST; and the MMBasic functions with firmware support -- SIN COS TAN LOG SQR ATAN2 ASIN ACOS POWER INT FIX ABS SGN RND TIMER, LEFT$ RIGHT$ MID$ UCASE$ LCASE$ CHR$ SPACE$ STRING$ INSTR STR$ VAL LEN ASC, RGB (including the colour names),

MM.HRES and MM.VRES, PRINT, and the drawing statements PIXEL, LINE, BOX, CIRCLE and TRIANGLE; and PIN(n) and PIN(n) = v for reading and driving a pin.

RGB of a colour name, or of three literal components, costs nothing at all: the translator works it out where it stands and it becomes a constant. RGB(r, g, b) of anything else checks each component is 0 to 255, as the interpreter's RGB does, before the shift-and-or. The drawing statements call the same firmware routines the interpreter calls, so what you get on screen is identical -- a converted routine and the original produce the same image byte for byte.

PIN is worth a word. Setting a pin up -- SETPIN, OPTION -- stays in BASIC, where it happens once; what converts is the reading and writing, which is the part that goes in a loop. It reads whatever the pin's mode means, through the interpreter's own routine, so a CSUB and PIN(n) cannot disagree. Needs firmware 6.03.02b10.

Which modes that covers, for PIN(n):

SETPIN mode	**PIN(n) gives you**
DIN, INTH, INTL, INTB	0 or 1
DOUT, and a PIO output	the level the pin is driving
CIN	the count
FIN, PIN	frequency, period -- the pin must be one of the four INT pins
AIN	volts, filtered exactly as the interpreter filters them
AINRAW	the raw count, 0 to 4095

and for PIN(n) = v: a pin set DOUT, a pin not configured at all (which becomes an output, as in BASIC), and CIN, where the write sets the counter rather than the pin.

Anything else -- a pin assigned to a UART, SPI, I2C, PWM or an INT, a heartbeat, a pin number that does not exist on the board -- is where the difference from BASIC shows. The converted code does not stop with the interpreter's error there: a read gives 0 and a write does nothing. Check the mode in BASIC before you call, or read ExtCurrentConfig[] from the CSUB if you want to branch on it.

PIN(TEMP) and PIN(BOOTSEL) do not convert. They are keywords rather than pin numbers and the firmware handles them before any pin exists; leave them in BASIC.

Not yet: string arrays as parameters; INPUT; ON ERROR; interrupt handlers; file I/O; user-defined TYPEs; MAP; SETPIN; and the graphics beyond the list above -- RBOX, POLYGON, sprites, framebuffers, and a LINE with a width, which is four different algorithms in the firmware and has no CallTable slot of its own.

If what you need is in that second list, or is something MMBasic cannot express at all, a hand-written CSUB can still do it -- that is a different job and has its own manual, docs/armcfgen.md, whose Appendix A lists every firmware routine a CSUB can call. The two tools produce the same kind of CSUB ... END CSUB block and can be mixed freely in one program.

String work inside a CSUB runs on the firmware's own string routines -- the same code MID$ uses when your BASIC calls it -- so a string-heavy routine gains what the interpreting overhead was costing and no more. See the table in section 1.

You do not have to memorise that list. Anything unsupported is refused, by name, before anything is written -- either by the tool or by the C compiler, which is the real scope check. The tool will not produce a CSUB it cannot account for.

Appendix A explains where each restriction comes from, and covers two things this list cannot: what a CSUB does instead of stopping with an error, and what is deliberately not a limitation.

5. The limits, and what to do about them

Arguments. MAX_CSUB_ARGS is 16 on RP2350 and 10 on RP2040, and that counts your parameters plus the globals the routine reaches. Past ten globals the tool gathers them into one array of addresses instead, which lifts the limit for practical purposes -- the wrapper fills it with PEEK(VARADDR x), which is the same call the interpreter makes to build an argument pointer. It costs a little wrapper text and nothing at run time.

Program memory. text in the listing, against what MM.INFO(PROGRAM SIZE) tells you is free. Remedies, in order: --lean, then crunching the rest of the program (XMODEM C, AUTOSAVE C, LOAD ,C), then LIBRARY SAVE.

Self-contained is best. A routine that only touches its parameters converts cleanly and cheaply. Every global it reaches costs an argument and makes the wrapper bigger.

Interrupting it. The interpreter tests the break key between statements, and your converted routine is now one statement. The tool puts a break check inside every loop so Ctrl-C still works; without that a long CSUB makes the board stop answering.

No subscript checks. The converted code does not check what the interpreter checks for you, so an out-of-range subscript writes to memory instead of reporting Index out of bounds. Convert code that already works. Appendix A.2 has the detail.

6. Options

--list	**report what can be converted and what it costs; change nothing**
--dry-run	build and report, leave the source alone
--library FILE	write the CSUBs and wrappers to FILE, for LIBRARY SAVE. A bare filename goes beside your program; give a path to put it elsewhere
--lean	keep nothing but the CSUB (with --library, applies to FILE)
--no-c	do not keep the generated C as comments
--no-original	delete the original routine rather than commenting it out
--keep-c	also leave the generated .c on disk
--no-backup	do not write <source>.bak
--name NAME	name the CSUB yourself
-O LEVEL	optimisation level (default s; prefer it for anything large)
--target cmm2	build for the Colour Maximite 2 instead of the PicoMite (§10)

When nothing needs wrapping, the CSUB takes the original routine's name and your call sites go straight to it. A wrapper appears only when something must be marshalled -- a FUNCTION's result, an array's BOUND(), or globals -- and then the wrapper carries the original name instead.

7. Getting the program onto the board

user-tools/xsend.py sends a program by XMODEM straight into program memory:

```
python xsend.py COM7 myprogram.bas
python xsend.py COM7 myprogram.bas --crunch    # strip comments on the way in
```

Four seconds rather than a minute for a large program -- and, more importantly, it is acknowledged. Pasting a program at the prompt can fail silently and leave the previous program running, which looks exactly like a successful run of the new one. If you benchmark that, you benchmark the old code.

8. When something goes wrong

undefined reference to mm_xxx -- that MMBasic feature has no CSUB runtime yet. The name tells you which: mm_input_next is INPUT, mm_map_get is MAP(). Take it out of the routine, or convert a different one.

undefined reference to __aeabi_xxx -- an arithmetic helper is missing from mmcsub.h. This one is a gap in the tool;

report it rather than working around it.

would need N arguments -- see the limits above.

X calls Y ... / X is a string FUNCTION -- the closure includes a routine that cannot be converted. The message names the real culprit, which is often not the routine you asked for.

It built, but the board crashes or resets on the first call -- check the firmware version before anything else: ? MM.VER must be 6.030209 (b9) or higher. See section 2.1. Nothing on your PC can detect this, because the blob is built without knowing what it will run on.

It compiled but the answer is wrong -- go back to step 7 and find the smallest input that differs. The .bak has the interpreted original, so you can run both. Please report it: the generated C is meant to be a faithful translation, and a difference is a bug in the tool, not in your program.

9. A worked example, and an honest one

solar_eclipse.bas -- 3,200 lines, 33 routines -- profiles like this: findleap 1,455 calls, utc2tdb 1,455, jdfunc 1,454, nut2000_lp 954, then a dozen routines around 480, and sefunc at 474.

The call-count ordering suggests converting findleap. The listing says otherwise: findleap is inside sefunc's blob, and converting sefunc covers 15 of the 20 profiled routines and 99.8% of the calls.

sefunc reaches twenty globals, which used to put it out of reach on argument count alone. It no longer does -- they travel as one array, so the CSUB takes three arguments: x!, fx! and the table. It converts, and the blob is correct. What stops it is size, in a way worth following through because it is the case --library was added for.

Its blob is 37 KB. In the program that is about 89 KB of text, which does not fit -- so it goes to a library instead:

```
  python mmb2csub.py solar_eclipse.bas sefunc --library sefunc.bas --lean
```

and on the board, LIBRARY LOAD "sefunc.bas". That makes a 37 KB library entry and leaves the whole of program memory for the program, on either board. The result:

	interpreted	**with sefunc in the library**
RP2350B at 252 MHz	13 s	2.7 s
RP2350B at 378 MHz		2.48 s
RP2040 (PicoMiteVGA)		5.7 s

One conversion, covering 15 of the 20 profiled routines and 99.8% of the calls, and the answers are identical on both boards to the last digit.

Note how little the clock speed matters once the routine is converted: half again as much CPU buys under 10%. Before the conversion the program was waiting on the interpreter, and that is what has gone.

One thing has to be done first, and it is the trap A.7 describes. The globals sefunc reaches must exist before the wrapper can take their addresses, so the five this program never declares need a Dim at program level.

On firmware older than 6.03.02b10 the library route cost 3.4x the blob and sefunc would not go onto a 100 KB board at all (A.8). The answer there is several smaller conversions from the second listing: tdb2utc has a 3.5 KB blob and carries jbrent, jdfunc, utc2tdb and findleap -- between them nearly 5,000 calls; gast2 carries nut2000_lp; eci2topo carries six more. Choose them so their closures do not overlap and put them all in one library file with a single command.

Two things to take from that. The listing, not the profile, tells you what to convert. And when a routine is out of reach it is now nearly always about size rather than shape -- which is a question of where you put the blob, not whether the tool can build it.

10. The Colour Maximite 2

```
python mmb2csub.py myprogram.bas PlotJulia --target cmm2
```

--target cmm2 builds the CSUB for the CMM2's Cortex-M7 (STM32H743) instead of the PicoMite's Cortex-M0+. Everything else in this document applies, with these differences.

- The code is M7 code, with the hardware FPU. Double arithmetic, square root, the int32 conversions, 64-bit multiply and 32-bit divide are single instructions, not calls into the firmware. Only 64-bit divide, the int64 to double conversions and the transcendental functions still go through the CallTable. The CMM2 finds the table the same way, through vector slot 7, with the same offsets, so PicoCFunctions.h serves both boards.
- A CMM2 CSUB runs only on the CMM2, and a PicoMite CSUB only on the PicoMite. To tell them apart, the first line of a CMM2 CSUB ends with the comment ' Cortex-M7 (CMM2).
- The CMM2 has no LIBRARY, so --library is not for it. The CSUB goes in the program, and the CMM2's 512 KB of program memory has room for large ones.
- The CMM2 has no / / block comment yet, so the original routine is kept with an apostrophe on each line instead.
- Firmware: the CMM2's PicoMite-layout CallTable (MMBasic 6.00.00b7, from 16 October 2026).

solar_eclipse.bas converts the same way as in §9: sefunc and the 18 routines it carries make a 26 KB blob, 37 KB for the M0+. With the five DIMs of §9 and no library:

	interpreted	**with sefunc as a CSUB**
CMM2 at 480 MHz	7.8 s	0.48 s

The output is identical to the interpreted run, including the golden JD values.

Appendix A -- Limitations, and where they come from

Section 4 lists what converts and section 5 the practical limits. This appendix explains why, because the reasons are more useful than the list: almost every restriction below follows from one of four facts about what a CSUB is, and once you know those four you can usually predict the answer without looking anything up.

The four facts.

- 1. To the interpreter, a CSUB is one statement.
- 2. The blob is code and constants only -- it has no writable data.
- 3. It links without a C library.
- 4. It has no path back to the interpreter's error handling.

A.1 No writable data

The firmware loads the blob at an address of its choosing and gives it no data segment. There is nowhere to put a variable that outlives a call.

STATIC still works, but not by living in the CSUB. The wrapper declares an MMBasic STATIC of its own and passes its address in, which gives exactly the lifetime the original had -- one instance, initialised on the first call, surviving every later one. Array bounds and initialisers are copied from your declaration, so this:

```
Static hits%, acc! = 1.5
Static tab%(4)
```

becomes a wrapper holding the same three variables and passing all three. The cost is one argument each, against the ceiling in A.5.

What you cannot do is keep state anywhere else. There is no equivalent of a C static inside the routine, and the 256 bytes of CFuncRam that a CSUB does own are already spoken for by the string scratch stack and the globals table.

You will not hit this by accident. Since 6.03.02b9 the linker checks it:

```
this CSUB has writable static data (.bss); a CSUB has no data segment
```

That check exists because the failure it prevents is silent. The C is valid, the compile is clean, the blob builds and runs -- and every access goes to memory belonging to something else. It is worth knowing about if you also write CSUBs by hand, because it applies to those identically.

A.2 Errors: some, not all

error() in the firmware does a longjmp back to the interpreter. A converted routine holds nothing that has to be unwound on the way: its strings live in the interpreter's temporary memory, its LOCALs on the C stack, and every call starts its scratch arena afresh. That is why the break key can already stop a converted loop - CheckAbort jumps out the same way. So the errors the translator itself checks for stop the program with the interpreter's message, exactly as the BASIC would: RGB(r, g, b) with a component outside 0 to 255. (END in a converted routine stops too, with "END in a CSUB".)

BOUND() of an array passed in as a parameter is not reliable. The wrapper hands the CSUB only the array's first bound, so BOUND() of such an array can give a wrong value, a wrong error, or no error where BASIC stops. Find the bound in BASIC and pass it in as an argument.

What a CSUB cannot do is everything the interpreter checks and plain C does not, and that has consequences worth taking seriously.

ON ERROR is refused. Nothing to add -- the tool tells you.

Array subscripts are not checked. This is the one that catches people. MMBasic checks every subscript and stops with Index out of bounds. The generated C does not check at all:

```
  __L->v_s[(int)(v_k)] = v_k;     /* no range test, by design */
```

A subscript one past the end writes to whatever is next in memory. In the interpreted version that same line gives you a clean error and a line number. Convert code that is already correct, and do not use a CSUB to find bugs.

Stack overflow resets the board. Recursion is supported and works, but without the interpreter's depth check. A recursive routine with a LOCAL array was measured both ways: MMBasic stopped it cleanly at depth 9 with Stack overflow, at depth 9, while the converted CSUB ran past depth 100 and reset the board somewhere before 150, with no message. Deep or unbounded recursion is the wrong thing to convert.

Most arguments are not range-checked. RGB is, as above; the string functions are not. They run on the same cores as the interpreter, which clamp rather than raise, so a position the interpreter would refuse gives a clamped answer instead. This only bites code that was already erroring in the interpreter, which is the general shape of the problem -- convert code that works.

The general rule: a CSUB fails the way C fails, not the way BASIC fails.

A.3 One statement

The interpreter checks the break key between statements, and your routine is now a single statement. The tool puts a break check inside every loop, so Ctrl-C still works -- without it a long CSUB makes the board stop answering. The check costs a counter increment per iteration and fires once every 1024.

Two smaller consequences: a profiler cannot see inside a CSUB, so convert before you profile again rather than after; and a run-time fault has no line number to report, which is the other half of A.2.

A.4 No C library

Everything routes to the firmware's own routines through the CallTable, so there is no second copy of soft-float or string code in the blob. When something has no route, you get a link error naming it:

- undefined reference to mm_xxx -- an MMBasic feature with no CSUB runtime yet. The name says which:

mm_input_next is INPUT, mm_fb_copy is the framebuffer.

- undefined reference to __aeabi_xxx -- an arithmetic helper is missing. That one is a gap in the tool rather than in your program; please report it.

This is why the tool can be trusted not to produce a CSUB it cannot account for: the compiler and linker are the scope check, so the supported list cannot quietly drift out of date.

A.5 The argument ceiling

MAX_CSUB_ARGS is 16 on RP2350 and 10 on RP2040, counting your parameters, one per global the routine reaches, and one per STATIC.

Past ten globals the tool stops passing them separately and gathers them into a single array of addresses, which lifts the limit for practical purposes. The wrapper fills it with PEEK(VARADDR x) -- the same call the interpreter makes to build an argument pointer -- so it costs a little wrapper text and nothing at run time.

A routine reaching twenty globals is still usually the wrong thing to convert, for the reason in section 9: it is a sign the work is spread across the program rather than contained in the routine.

A.6 Strings

String work runs on the firmware's own routines -- the same code MID$ uses when your BASIC calls it -- so results are identical and the gain is only the interpreting overhead. Three limits:

A statement's temporaries are bounded. String expressions build temporaries in a 4 KB scratch arena, about sixteen full-length strings, thrown away at the end of each statement. A single statement that builds more than that wraps round and reuses the space rather than overrunning it, which would corrupt that statement's own earlier temporaries. Splitting a monster concatenation across several statements is the fix; in practice this is very hard to reach.

String arrays cannot be parameters. A string array's element stride follows the LENGTH it was declared with, and the CSUB is never told what that is, so indexing one would be wrong silently. Refused for that reason. Scalar string parameters are fine -- they are already exactly what the ABI passes. The same applies to a STATIC string array.

LENGTH on a simple string variable is ignored by MMBasic itself, which always gives those 256 bytes -- so nothing is lost in translation there.

A.7 Globals the program never declares

A wrapper reaches a global with PEEK(VARADDR x), which needs the variable to already exist. Without OPTION EXPLICIT, MMBasic creates a global the first time something assigns to it -- so a global that only ever came into being inside the routine you are converting will no longer be created by anything, and the wrapper stops with:

```
  Error : Cannot find DECL
```

The fix is one line at program level, naming the variables the message complains about:

```
  Dim decl, rasc, rb, rlsun, rmm
```

This is worth checking before you convert rather than after. solar_eclipse has five such globals out of the twenty sefunc reaches: they are assigned inside sun and never declared anywhere, so they exist only because sun has run. Programs written with OPTION EXPLICIT cannot have the problem.

A.8 Program memory

A CSUB costs program memory twice: once as the hex text of the CSUB ... END CSUB block, and again as the binary

the interpreter builds from that text. The board shows them on separate lines:

```
 Program:
    9K ( 8%) Program (126 lines)     <- the hex text, about 2.4x the blob
    3K ( 2%) 1 Embedded C Routine    <- the binary,   about 1x
   88K (90%) Free
```

So budget about 3.4x the blob size, which is what the prog column of --list reports.

A missing Embedded C Routine line is the tell that a CSUB's code did not make it. The program still lists and the CSUB still appears, because the text fit; only the binary was dropped. Calling it then gives Error : Internal fault 5(sorry).

In order of what to try: --lean, then crunching the rest of the program (XMODEM C, AUTOSAVE C, LOAD ,C), then --library and LIBRARY LOAD, which puts the blob in the library and leaves the program almost all of program memory.

The library costs the blob and nothing more. LIBRARY LOAD "file.bas" reads the file itself, converting each CSUB's hex to binary as it streams past, and never stores the hex text at all. So the limit is simply the library area -- a slot the size of program memory, 100 KB on an RP2040 and 144 KB on an RP2350 -- against the blob alone, not against 3.4x of it.

solar_eclipse is the case that makes the difference plain: its 37 KB sefunc becomes a 37 KB library entry and loads on either board (section 9).

Older firmware is the thing to watch. Before 6.03.02b10 the only route was LIBRARY SAVE, which takes the CSUB out of program memory -- so the library file had to load there first, costing its hex text and the binary at once, about 3.4x the blob. That put the ceiling near a 29 KB blob on an RP2040. Past it the failure was quiet and is worth recognising if you meet it on an older board: the load stops early with no Saved nnn bytes, the save reports a byte count far smaller than the blob, LIBRARY LIST still shows the CSUB because its declaration fitted, and calling it gives Error : Internal fault 5(sorry).

A.9 What is not translated at all

As of 6.03.02b9, 132 of the 166 routines in the test corpus convert. What the remaining 34 are waiting on, largest group first:

interrupt handlers (SETTICK, ON KEY, pin interrupts)	**a CSUB cannot be one**
user-defined TYPEs, as parameters or locals	not yet translated
INPUT	no CSUB runtime yet
graphics beyond PIXEL, framebuffers, file I/O	no CSUB runtime yet
DIM of a LOCAL array at run time	needs a heap the blob does not have
routine names containing dots	a tool bug, not a limitation

You do not have to memorise any of it. Everything unsupported is refused by name before anything is written, so the failure mode is a message rather than a surprise.

A.10 What is not a limitation

Worth stating, because it saves chasing imagined problems:

- Arithmetic is identical. Integer and floating-point operations, and every maths function, call the same firmware routines the interpreter calls. There is no reduced-precision path and no second implementation.
- String functions are identical, for the same reason.
- Recursion works, within A.2's caveat about depth.
- LOCAL arrays and strings work, including in recursive routines, each call getting its own.
- Call sites do not change. The CSUB takes the routine's name, or a wrapper does when something has to be marshalled.
- One blob runs everywhere. RP2040 and RP2350, every firmware variant, no per-chip build -- see the end of section 1. A converted program is as portable as the BASIC it came from.

So if a converted routine gives a different answer from the interpreted one, that is a bug in the tool, not a rounding difference or a documented limit. The .bak still has the original; please report the smallest input that differs.