

Writing CSUBs for MMBasic on the PicoMite

A Comprehensive Guide to Embedding C Code in MMBasic Programs

For Raspberry Pi Pico (RP2040) and Pico 2 (RP2350)

Important Notice

Writing CSUBs is a Complex Subject

This document provides guidance for creating simple, single-function CSUBs. However, it's important to understand that CSUB development is inherently complex and challenging:

- **Complex CSUBs:** For anything beyond simple functions, comprehensive CSUB development requires deep understanding of ARM assembly, memory management, and MMBasic internals. The helper scripts referenced in this document are limited to single-function CSUBs.
- **Testing is difficult:** Other than compilation errors, CSUB testing is primarily go/no-go. Many bugs simply cause a system crash with no diagnostic information. This makes debugging extremely challenging.
- **Limited automation:** Current automated tools are best suited for simple, single-function CSUBs. More complex scenarios require manual development and a deep understanding of the MMBasic CSUB interface.
- **AI limitations:** At present, AI tools are unlikely to be able to write correct CSUBs beyond simple examples. Human expertise in C programming, ARM assembly, and the PicoMite architecture is essential.

Recommendation: Begin with very simple CSUBs to understand the workflow. As you gain experience, gradually increase complexity. Always maintain thorough documentation and backup your working code, as debugging complex CSUBs can be extremely time-consuming.

Introduction

CSUBs (C Subroutines) are a powerful feature of MMBasic that allows you to embed compiled ARM machine code directly into your BASIC programs. This provides native processor execution speed for performance-critical sections of code while maintaining the ease and flexibility of BASIC for the rest of your program.

Key Benefits:

- 7-10x performance improvement over interpreted BASIC for simple math operations
- No firmware rebuild required
- Drop-in replacement for slow BASIC loops
- Native ARM Cortex-M0+ (RP2040) or Cortex-M33 (RP2350) execution
- Can be saved to library for reuse across programs

For example, a Mandelbrot set renderer can be accelerated from 70 seconds to just 10 seconds using a CSUB implementation. However, this level of performance comes at the cost of significantly increased development complexity.

What is a CSUB?

A CSUB contains ARM processor machine code instructions stored in hexadecimal format within your BASIC program. When your program runs, MMBasic automatically

loads this hexadecimal code into memory as binary and you can call it just like a normal BASIC subroutine or function.

Basic Structure:

```
CSUB MyFunction integer, float
00000000 B085B480 6078AF00 687B6039
1AD368FB 687A3301 6839441A
End CSUB
```

The hexadecimal values represent compiled ARM machine code. You never edit these values directly - they are generated by compiling your C code using the ARM toolchain.

Scope and Limitations of This Guide

This guide focuses on creating simple, single-function CSUBs suitable for accelerating mathematical computations or simple data processing tasks. It uses helper scripts that streamline the basic CSUB creation workflow.

What This Guide Covers

- Creating single-function CSUBs for mathematical operations
- Basic parameter passing and result handling
- Using helper scripts to automate compilation and conversion
- Simple fixed-point arithmetic implementations
- Basic testing and troubleshooting

What This Guide Does NOT Cover

- Multi-function CSUBs with complex interactions
- CSUBs that access MMBasic internal functions
- Advanced memory management and DMA operations
- Interrupt handling from within CSUBs
- Direct hardware access and register manipulation
- Sprite conversion and complex graphics operations

Note: For comprehensive CSUB development, particularly for sprite/graphics operations or multi-function CSUBs, refer to the MMBasic source code on GitHub (github.com/UKTailwind/PicoMiteAllVersions) and examples by Peter Mather (matherp) on The BackShed forum.

Getting Started

Required Tools

ARM Toolchain:

- macOS: brew install arm-none-eabi-gcc
- Linux: sudo apt-get install gcc-arm-none-eabi
- Windows: Download ARM GCC toolchain from ARM website

Helper Tools (For Simple CSUBs):

The **picocalc_csub_helpers** toolkit from GitHub provides build scripts and examples for simple, single-function CSUBs:

https://github.com/jvanderberg/picocalc_csub_helpers

Important: These helper scripts are designed for single-function CSUBs only. For more complex multi-function CSUBs, comprehensive manual development is required. Peter Mather has raised an issue with the script author regarding expanded capabilities.

C Function Structure

Basic Template

Every CSUB function follows this basic structure:

```
#include "PicoCFunctions.h"

long long your_function_name(void *arg0, void *arg1, ...) {
    // Your C code here
    return result;
}
```

Key Components

- **Return type:** Always `long long` (64-bit integer)
- **Parameters:** Always `void *` pointers to arguments
- **Header file:** Must include `PicoCFunctions.h` from the PicoMite source (github.com/UKTailwind/PicoMiteAllVersions)

Critical: The `PicoCFunctions.h` header file must match your firmware version exactly. Using a mismatched header can cause crashes or unpredictable behavior.

Complete Working Example

Integer Power Function

This example implements an integer exponentiation function (calculating base raised to the power of exponent). This is a simple, single-function CSUB suitable for the helper scripts:

```
// pow_int_cf.c - Integer exponentiation
#include "PicoCFunctions.h"

long long pow_int_cf(void *result_ptr, void *base_ptr, void *exp_ptr) {
    // Extract parameters from pointers
    int base = *(int*)base_ptr;
    int exp = *(int*)exp_ptr;

    // Calculate result
    int result = 1;
    for(int i = 0; i < exp; i++) {
        result *= base;
    }

    // Write result back to first parameter
    *(int*)result_ptr = result;

    return 0;
}
```

Using in BASIC

Once compiled and converted to a CSUB, you can call it from your BASIC program:

```
' Define the CSUB
CSUB pow_int integer, integer, integer
00000000 B085B480 6078AF00 687B6039
... (hex code continues) ...
End CSUB

' Call the function
DIM result%, base%, exp%
base% = 2
exp% = 10
pow_int result%, base%, exp%
PRINT result% ' Prints 1024
```

Testing and Debugging Challenges

Important: CSUB testing is inherently difficult. Unlike normal programming where you get helpful error messages and debugging tools, CSUBs provide very limited diagnostic information.

The Reality of CSUB Testing

- **Go/No-Go testing:** Most CSUB testing is binary - it either works correctly or crashes. There are few intermediate states.
- **Silent crashes:** Bugs in CSUBs typically cause the PicoMite to hang or reset without any error message or diagnostic output.
- **No debugger:** Traditional debugging tools like breakpoints, step-through execution, and variable inspection are not available for CSUBs.
- **Difficult to isolate:** When a CSUB crashes, it can be very difficult to determine which line of code or which operation caused the failure.

Testing Strategies

Given these limitations, careful testing strategies are essential:

1. **Test C code independently first:** Before creating a CSUB, thoroughly test your C algorithm in a standard development environment where you have full debugging capabilities.
2. **Start extremely simple:** Begin with the simplest possible CSUB (e.g., adding two numbers) and verify it works before adding complexity.
3. **Build incrementally:** Add one feature at a time. Test each addition before proceeding. Never add multiple features simultaneously.
4. **Use known-good test cases:** Test with inputs where you know the exact expected output. Verify basic functionality before testing edge cases.
5. **Check compilation warnings:** Compilation warnings often indicate problems that will cause runtime crashes. Fix all warnings before testing.
6. **Verify parameter types:** Ensure the CSUB declaration in BASIC matches your C function signature exactly. Type mismatches are a common cause of crashes.
7. **Save working versions frequently:** When you have a working CSUB, save both the source code and the compiled hex immediately. Recovery from crashes can be time-consuming.

When Things Go Wrong

If your CSUB crashes the system:

- Remove the CSUB from your program and verify the BASIC code works
- Comment out sections of the C code to isolate the problem area
- Check for array bounds violations and null pointer dereferences
- Verify all pointer casts are correct for your data types
- Ensure you're not using too much stack space
- Check that you're targeting the correct processor (RP2040 vs RP2350)

Reality Check: Debugging complex CSUBs can take hours or even days. The lack of diagnostic tools means that troubleshooting often involves educated guessing and systematic elimination. This is why starting simple and building incrementally is absolutely essential.

Compilation Process

Compiling for RP2040 (Cortex-M0+)

For the original Raspberry Pi Pico with RP2040 processor:

```
arm-none-eabi-gcc -c -mthumb -mcpu=cortex-m0plus \  
    -O2 -fno-exceptions -ffunction-sections \  
    -o output.o your_code.c
```

```
arm-none-eabi-ld -nostdlib -lgcc -T link.ld output.o -o output.elf
```

```
arm-none-eabi-objcopy -O binary output.elf output.bin
```

Important: RP2040 lacks hardware 64-bit multiply, so you must link with `-lgcc` to include the `__aeabi_lmul` helper function.

Compiling for RP2350 (Cortex-M33)

For the Raspberry Pi Pico 2 with RP2350 processor:

```
arm-none-eabi-gcc -c -mthumb -mcpu=cortex-m33 \  
    -O2 -fno-exceptions -ffunction-sections \  
    -o output.o your_code.c
```

Converting to CSUB Format

After compilation, convert the binary to hexadecimal format for embedding in MMBasic:

```
# Using the helper script (for single-function CSUBs)  
./emit_cfunction_block.sh output.bin "function_name integer, integer"
```

This generates the CSUB block ready to paste into your BASIC program.

Parameter Handling

Parameter Types

When declaring a CSUB, you specify the type of each parameter:

- **integer** - 64-bit signed integer
- **float** - Double precision floating point
- **string** - String variable

MMBasic will automatically convert types if possible or throw an error if incompatible types are provided.

Accessing Parameters in C

Parameters are passed as void pointers. You must cast them to the appropriate type:

```
// For integers  
int value = *(int*)param_ptr;
```

```
// For floats
double value = *(double*)param_ptr;

// For strings
char *str = (char*)param_ptr;

// Writing results back (command-style)
*(int*)result_ptr = calculated_value;
```

Best Practices and Recommendations

8. **Start extremely simple:** Your first CSUB should do almost nothing - add two numbers, multiply two numbers. Build confidence before complexity.
9. **Test C code thoroughly first:** Debug your algorithm in a standard C environment with full debugging tools before converting to a CSUB.
10. **Keep functions small:** Smaller CSUBs are easier to debug. Complex CSUBs should be avoided unless absolutely necessary.
11. **Use fixed-point math when possible:** Often faster than floating-point on microcontrollers and simpler to debug.
12. **Document everything:** Include comments explaining your algorithm, parameter types, expected ranges, and known limitations.
13. **Save working versions:** Keep both source code and compiled hex for every working version. Version control is essential.
14. **Understand your limitations:** Recognize when a problem is too complex for your current CSUB development skills.
15. **Use the library for proven code:** Only save thoroughly tested CSUBs to the library.
16. **Join the community:** The BackShed forum has experienced CSUB developers who can provide guidance.

Advanced Topics

Using the Helper Toolkit (Simple CSUBs Only)

The `picocalc_csub_helpers` toolkit simplifies the workflow for single-function CSUBs:

```
# 1. Clone the repository
git clone https://github.com/jvanderberg/picocalc_csub_helpers
cd picocalc_csub_helpers

# 2. Set target processor
export CPU_TARGET=rp2350 # or rp2040

# 3. Compile your simple function
./build.sh my_func.c

# 4. Generate CSUB block
./emit_cfunction_block.sh my_func.bin "my_func integer, float"
```

Beyond Simple CSUBs

For complex CSUBs that go beyond single-function mathematical operations, the helper scripts are insufficient. You will need to:

17. Study the MMBasic source code thoroughly
18. Examine existing complex CSUBs by experienced developers
19. Understand the MMBasic internal function calls
20. Learn ARM assembly language for optimization
21. Engage with the community on The BackShed forum

Note: Peter Mather (`matherp`) has written comprehensive BASIC programs for converting sprites and other complex operations into CSUBs. These are available on GitHub in the MMBasic source repository and serve as excellent examples of advanced CSUB programming.

Saving to Library

Once your CSUB is working correctly and thoroughly tested, you can save it to the library:

```
LIBRARY SAVE
```

Benefits of using the library:

- CSUB is permanently available to all programs
- Hexadecimal version is discarded (only binary kept)
- Frees up program space
- Can be called from any program without re-embedding

Critical: Keep a separate copy of your source code and hex CSUB. The library only stores the binary form, and you cannot recover the source from it.

Troubleshooting Guide

Problem	Solution
CSUB crashes immediately with no error	Check parameter types match declaration exactly. Verify compilation for correct processor (RP2040 vs RP2350). Check PicoCFunctions.h matches firmware version.
System hangs when CSUB runs	Likely infinite loop or deadlock in C code. Test algorithm in standard C environment first. Add timeout mechanisms where possible.
Incorrect results	Test C code separately first. Verify pointer dereferences and type casts. Check that results are being written back to correct location. Test with known-good inputs.
Compilation errors	Ensure PicoCFunctions.h is in include path and matches firmware version. Check ARM toolchain is correctly installed. Verify CPU target matches hardware.
MMBasic syntax error when loading	Check CSUB declaration syntax matches parameter types. Ensure End CSUB is present. Verify hexadecimal format is correct with no missing lines or corrupted data.
Linker errors on RP2040	Add -lgcc flag to linker command to include 64-bit multiply helper (__aeabi_lmul). This is required for RP2040.
Intermittent crashes or corruption	Check for buffer overruns, uninitialized variables, stack overflow. Verify memory alignment requirements. Test with different input values to isolate conditions.

When All Else Fails: If you cannot resolve a CSUB crash, simplify your function to the absolute minimum and rebuild from there. Sometimes the only solution is to start over with a better understanding of what went wrong.

Performance Comparison

CSUBs provide significant performance improvements for computationally intensive tasks:

Task	Pure BASIC	CSUB
Mandelbrot 320x320	~70 seconds	~10 seconds
Simple math loops	Baseline	7-10x faster

Task	Pure BASIC	CSUB
Fixed-point operations	Baseline	10-15x faster

These improvements make CSUBs valuable for graphics rendering, signal processing, and compute-intensive applications. However, the development time and debugging difficulty must be weighed against the performance benefits.

Additional Resources

Official Documentation:

- PicoMite User Manual: geoffg.net/picomite.html
- MMBasic Website: mmbasic.com
- PicoMite Firmware Source: github.com/UKTailwind/PicoMiteAllVersions

Helper Tools (Simple CSUBs Only):

- CSUB Helper Toolkit: github.com/jvanderberg/picocalc_csub_helpers

Community Resources:

- BackShed Forum (Primary Support): thebackshed.com/forum/Microcontrollers
- FruitOfTheShed Wiki: fruitoftheshed.com/wiki

ARM Development:

- ARM GCC Toolchain: developer.arm.com
- RP2040 Datasheet: raspberrypi.com/documentation
- RP2350 Datasheet: raspberrypi.com/documentation

Conclusion

Writing CSUBs for MMBasic on the PicoMite is a powerful but challenging endeavor. This guide has focused on simple, single-function CSUBs that can provide significant performance benefits for specific computational tasks.

However, it's essential to understand that CSUB development is inherently complex. The lack of debugging tools, the go/no-go nature of testing, and the potential for difficult-to-diagnose crashes mean that CSUB development requires patience, careful planning, and incremental development.

For simple mathematical operations and well-defined algorithms, CSUBs can provide dramatic performance improvements. For more complex operations, the development time and difficulty may outweigh the benefits unless you have significant experience with embedded systems programming.

If you choose to pursue CSUB development, start simple, test thoroughly, document extensively, and engage with the community on The BackShed forum. The experienced developers there, including Peter Mather (matherp) and others, are an invaluable resource for guidance and support.