picolibc

A C Library for Smaller Systems

Keith Packard
Senior Principal Technologist
Amazon
keithpac@amazon.com

amazon

Hello! My name is Keith Packard, I work in the Device OS group at Amazon as a Senior Principal Engineer. Today, I'm going to talk about picolibc, a C library designed for embedded 32- and 64- bit microcontrollers.

picolibc

- Why start another C library project?
- How picolibc was developed?
- What are the results?
- Who is using picolibc?
- Where is picolibc going in the future?

amazon

In this presentation, I'll describe what I see as the requirements for an embedded C library, the pieces from which picolibc was built, how development progressed, results of that development along with several projects that have picolibc support integrated into them. Finally, I'll briefly touch on future work that I hope to see done in picolibc.

Small System Considerations

- Small Memory (think kB, not GB)
  - RAM is more constrained than ROM
  - Avoid using the heap
- Limited floating point
  - May have only 32-bit floats
  - May have none at all
- Getting started is hard

amazon

Small systems are small. They can be really small. The embedded hardware I fly uses systems with 32kB to 128kB or ROM and 4kB to 32kB of RAM. They often use low-performance 32-bit cores, some have a 32-bit FPU, but many have no FPU at all. Developers writing long-running embedded software often want to avoid depending on dynamic allocation as it's easist to prove malloc will never fail if malloc is not linked into the application.

As a developer, one of the steepest learning curves with embedded development is getting the very first 'hello-world' application compiled, linked, installed and running. Arduino has had tremendous success by making that process easy enough for me to teach to middle school kids.

Embedded Libc Needs

- String Functions (strcpy, memset)
  - Ideally accelerated for architecture
- Math Functions (acos, j0, sqrt)
  - Often for soft-float processors
- Stdio (printf, sscanf)
  - debugging
  - data exchange
- Licensing
  - Share-alike licenses are challenging

amazon

Embedded systems generally make modest demands on the C library. The most commonly used functions are string and memory APIs; things like strcpy, memmove and the like. Applications doing computation may need some math functions, like sqrt, sin or even bessel functions.

Standard I/O will often be only used during debugging to send output to a serial console to help diagnose software and hardware issues. However, applications may also use these functions to handle JSON, YAML or other text interchange formats and so strict standards conformance can be critical.

While I am a strong proponent of share-alike licenses, a C library needs a pragmatic approach to be useful to all developers. A clear MIT or BSD license is best practice in this space.

Developer First Steps

- Startup code
  - Initialize RAM, setup clocks and interrupts
- Linker script
  - Memory layout
- Debug output
  - Serial ports require drivers and working clocks

amazon

To make starting with a new embedded platform easier, developers will benefit from extra support in the C library. First, the processor, memory and library need to be initialized.
Second, the linker needs to know how to lay out the application in ROM and RAM. This includes hardware-specific information, processor setup instructions and library initialization code.
Finally, providing debug output from applications using semihosting or other architecture-specific mechanism lets the developer see debugging information before clocks and serial port drivers are working.
With these three tools, a new SoC can be brought up only knowing the memory layout of the device as we'll see later in some examples.

Current 32-bit Libc Options

- newlib and newlib-nano
  - excellent API support, including optimized assembly code and solid math functions.
  - designed for systems with an OS
  - development focused on Cygwin target
  - libgloss wraps OS functions into POSIX-like API
  - stdio is fast, but large and malloc-intensive
- various proprietary options
  - closed source
  - often brought up from 8-bit environments

When I started doing embedded development in the modern era in 2009, I was using an 8051 SoC. The compiler, SDCC, came with a small C library so I didn't have to think about it.

In 2012, when we build our first cortex m3 board, I had to also find a new C library. For this project, newlib was too big and required too many support routines in the form of libgloss, including sbrk for malloc, which was required for stdio.

I initially settled on pdclib as that was at least reasonably small. However, it used excessive amounts of stack space for printf, and had no math support. This library seems to be an experiment in building a small C library, but not in building a C library for small systems.

Some SoC vendors provide proprietary C libraries. These are often ports from older 8- and 16- bit systems which can lack compliance with modern C standards in some areas. And, of course, they're not free software, which makes them uninteresting to me.

After a couple of years using pdclib, I decided to return to newlib and see what could be done to make that usable for my work.

newlib concerns

- Stdio
  - A lot of code and uses a lot of RAM
  - Requires a large number of POSIX(ish) APIs
- Libgloss
  - A shim between the POSIX apis and the underlying OS
  - Not relevant on embedded systems with no OS
- struct _reent
  - holds all per-thread state in the library
  - even for functions you don't use
  - an unexpected malloc user
- Out-dated coding practice
  - newlib still has K&R function definitions in places (!?)
  - Many signed/unsigned confusions, especially in wchar_t and stdio
  - Many (many) warnings
- No integrated test suite
  - Without testing, there's no way to know when things break

I want to say first that newlib is a great project; the code is easy to work on, the community is welcoming and supportive and I've been happy to collaborate with them. Everything here relates to using newlib for my small embedded projects, which is not what newlib is really designed for.

In looking at the changes I wanted to make, the first thing on my list was stdio. Newlib's stdio is designed to offer good performance and full compatibility on systems with an underlying POSIX api. It has internal buffering, and allocates space on the heap for them. Replacing the whole thing seemed like the best option.

I also wanted to get rid of the libgloss OS shim layer. libgloss was originally designed as an interface between newlib and whatever underlying operating system it was run on. With bare-metal environments, there is no OS, and so there really isn't a reason to have this library any more. Newlib uses POSIX interfaces internally, porting to an RTOS is usually a simple matter of making sure the necessary POSIX interfaces are available. Fortunately, most of the common uses for libgloss APIs were in stdio; by replacing that, the bulk of the problem was already solved.

One huge consumer of RAM in newlib is 'struct _reent'. This struct holds all per-thread state for the entire library. Replacing this with modern thread local storage techniques would free up a lot of memory while still allowing applications to take advantage of library APIs which needed persistent state.

Newlib is an old code base that has seen development extend back to the pre-ANSI era of compilers. The C language has changed a lot since then, and how we use the language has changed even more. It's been a long time since I defined a new API with a bare 'int' type; we know now that it's best to use a more specific type that won't surprise you with overflow issues or sign errors on some new platform. At a minimum, the library should compile cleanly with -Wall -Wextra.

Finally, it's unreasonable to expect code review alone to catch errors in changes to the library. An extensive and automated test suite is table stakes for a project of this size.

I got started hacking on the stdio changes that I wanted to make and sent them in to the newlib mailing list to see what feedback I might get. That was when the reality of what newlib was hit me. Newlib is **not** a library for embedded applications. Newlib is the Cygwin C library which happens to be used by many embedded developers lacking an alternative. In particular, radical changes to the API and ABI of the library could never be integrated as those would break binary compatibility for Cygwin applications. So I decided to fork newlib and start a new project

picolibc

- newlib math, i18n, strings
  - good performance, wide support
- stdio adapted from AVR libc
- 'struct _reent' removed
  - All thread-local values use toolchain TLS
- sample startup code, linker scripts
- semihosting support
- Fix warnings, switch to C18
  - Builds with --std=c18 -Wall -Wextra -Werror
- New meson-based build system
- Testing
- Pure BSD licensed library code.

amazon

Once I decided to abandon attempts to upstream my work, I felt free to make all of the changes I wanted, building the library as I thought best.

The good parts of newlib I kept; the math, string and locale implementations work great.

Having had experience with AVR libc on Atmel processors, I took that code as the basis for a new stdio implementation.

To get rid of struct _reent, I decided to replace it with thread local storage as supported by the C language. That involved changing nearly every file in the library, removing the parameter from internal functions, creating new thread-local versions of all global data.

The tests need to run on the target architectures. Picolibc includes enough infrastructure to build and run applications on bare-metal hardware without depending on any external software. That includes startup code, linker scripts and a mechanism to get error codes out of the test and up into the test framework running on a Linux host. These are also what developers need when starting work with a new chip.

Changes to move the code base from old-style C to more modern practice were made so that picolibc now builds with a C18 compiler and builds cleanly with GCC/Clang -Wall -Wextra.

Newlib also uses an old-style build system, with a maze of autotools bits and nested configure scripts. Instead of modifying those, I decided to overlay a new build system on top using meson. This had a couple of advantages: First, it meant that I could ignore the existing build system so that upstream changes to that wouldn't affect merging code changes in. Second, meson builds are dramatically faster. A full build for all embedded arm targets, compiling 28429 files, takes only 38 seconds with an empty compiler cache. With the cache hot, that same build takes but 6.3 seconds.

For some initial testing, I re-animated the newlib test suite and then started adding more tests.

Newlib includes piles of GPL code, but none of that is likely to be used in a non-Cygwin binary. To make the picolibc license status clear, all of the non-BSD licensed library code has been removed. The printf test cases came from a GPL licensed testing project, but those are not part of any possible library build.

Math code

- Can be compiled two ways
  - “IEEE” mode (no errno) or “POSIX” mode (errno)
- ARM added some new 'float' implementations using 'double':
  - Less accurate than old pure 32-bit versions
  - Not used in default picolibc builds
- Fixed issues with errno/exception values
  - modern compilers make exception generation difficult.

amazon

Now I'll start going into some of these changes in more detail, starting with the math code.

One of the most useful bits of newlib is the math library. That includes implementations of the standard math functions as specified in IEEE-754, ANSI-C and POSIX. Lots of this code came from Sun back from the early 1990's.

A couple of years ago, ARM provided some new float code using double precision intermediates to reduce the number of instructions necessary for these operations. Unfortunately, the resulting code is less accurate than the old Sun code. And, of course, it's really only useful on hardware that has 64-bit floats. As a result, picolibc disables this code in default builds.

Some of the math library code is split into two layers; a lower-level IEEE-754 API which does the computation including raising exceptions, and a higher-level C API which uses the lower-level function and then also updates errno as appropriate. The library can be built without errno support, which eliminates these C wrappers and exposes the IEEE functions using the C names. These functions should be squashed together, moving the errno bits into the lower level functions.

ANSI-C has slowly refined the use of errno and now has relatively consistent behavior based on the underlying IEEE-754 exception definitions. POSIX constrains behavior further, for instance C17 says that a domain error may occur for pow(0,0) and provides no hint to the value returned while POSIX requires a return value of 1.0 and that no errors are raised. Picolibc code has been modified to follow the POSIX spec as closely as practical.

Exception generation relies on executing instructions that raise the appropriate exception, and that is challenging when C compilers work hard to move computation from runtime to compile time. I don't know of a sure-fire way to force a particular computation to happen at run-time; we've adopted some techniques which work with existing compilers, but will need to keep testing the library to make sure compilers don't outsmart us in the future.

With the picolibc testing infrastructure in place, tests were added to verify errno and exception values for all math library functions and all exception causes.

stdio

```
typedef struct __file {
     ungetc_t unget;                    /* ungetc() buffer */
     uint8_t  flags;                    /* flags, see below */
     int  (*put)(char, struct __file *);  /* function to write one char to device */
     int  (*get)(struct __file *);        /* function to read one char from device */
     int  (*flush)(struct __file *);      /* function to flush output to device */
} FILE;
extern FILE *const stdin, *const stdout, *const stderr;
```

- Started with avrlibc code
- Added flush to allow for buffering
- Added POSIX backend
  - requires read/write/lseek/open/close
- Changed ungetc() to use atomics for re-entrancy
- FILE takes just 16 bytes of RAM

amazon

Picolibc inherits its stdio from AVR libc. This code uses a small in-RAM struct to hold per-FILE state. Picolibc stdio is built on two primitive operations – get a character, put a character. Any FILE struct includes pointers to functions which do that. This means that implementing I/O for another system involves implementing just these two functions, not a larger set of POSIX APIs.

To make picolibc's stdio code re-entrant, without requiring locking, the 'unget' field is manipulated with atomic operations. As this field gets modified at run-time, the struct itself needs to live in RAM.

There are macros which help applications initialize this structure correctly, also inherited from AVR libc.

stdin, stdout and stderr are global pointers to FILE structs; they are declared const as picolibc doesn't modify them and so that they may be placed in ROM. These can all point at the same underlying FILE object if desired.

For systems that do support POSIX functions, picolibc provides fopen and friends using those. fdopen creates a FILE struct with getc and putc pointing at functions inside picolibc which then use POSIX read and write. This support doesn't extend to fseek or rewind; those are not yet implemented, although could be added fairly easily.

The stdio code has made some obvious tradeoffs for size and simplicity vs performance; funneling all I/O through narrow 1-byte functions slows things down. There is also no multi-byte or wide character support. For my applications, those have seemed like good choices; I'd like to hear what other people think.

printf & scanf

- float code takes a lot of space
  - but is careful to avoid needing soft-float
  - offer “int-only” and “float-only” versions
  - Use Ryū malloc-free “exact” float conversions
- Most of C18 (except numbered arguments)

amazon

Printf, and scanf, to a lesser degree, are key functions for any C library. When used for debugging, they needs to be small and avoid interfering with other parts of the system. When used for data interchange, they need to be ANSI-C/POSIX compliant. The original AVR libc printf is remarkably tight, and has only a few issues with standards conformance.

The one major gap is in printing and scanning floating point numbers. These are fundamentally base conversion functions, converting between the base-10 printed form and the base-2 internal form. Think about how 0.1 is represented in binary form: 0.0{0011}, an infinite repeating fraction. This makes exact conversion impossible.

I re-define “exact” float/string conversion as one where a number printed and rescanned generates exactly the same bits. newlib, glibc and other C libraries use arbitrary precision numbers for “exact” conversion. This is slow, large, and generally requires malloc. Picolibc uses the Ryū fixed precision algorithms developed by Ulf Adams to perform these operations – 128 bits for double, 64 bits for float. This reduces code size and avoids malloc calls.

Picolibc has numerous additional fixes and improvements, including hex-format floats so that it nearly meets the C18 requirements, the notable exception being numbered arguments.

Thread Local Storage

- TLS instead of 'struct _reent'
- Linker limits TLS space to in-use vars
- Add API to manage TLS
  - _set_tls(): Set TLS base (thread switch)
  - _tls_size(): Get TLS area size (thread alloc)
  - _init_tls(): Initialize TLS area (thread create)
- Initial static TLS area setup by sample linker script
- Saves a ton of RAM, eliminates malloc calls from deep within the library.

amazon

Newlib was developed before threading was common in POSIX environments. In that era, there was no C language support for thread local storage. To work around this, newlib introduced 'struct _reent'. This struct contains every variable in the library which needs to be thread-specific according to the POSIX. That includes common things like 'errno' and rarely used items like 'signgam', the global variable used to hold the sign returned from lgamma. This uses 656 bytes on arm. Newlib-nano includes changes which dynamically allocate much of this struct as needed reducing the static structure to 96 bytes.

Picolibc eliminates this struct, instead using the C language __thread storage class. The linker gathers together all of the variables of this class, effectively creating a struct on the fly. Only TLS variables used by the application end up in the application. At run-time, access to these values uses a ABI-specific mechanism, for some architectures, Picolibc supplies helper functions. Picolibc also provides functions to allocate, initialize and set up the ABI-specific mechanism so that applications can manage multiple TLS blocks without needing a detailed understanding of the target architecture.

Example Startup Code (crt0)

- Initialize CPU
  - Stack pointer and other registers
  - FPU
  - MMU
- Initialize memory
  - Copy ROM bits to .data segment
  - Clear .bss segment
- Setup interrupt controller
  - provide handlers for system exceptions
- Arm, Thumb, Aarch64, RISC-V, x86, x86_64
- Three versions:
  - regular: constructors, spins if main returns
  - `--crt0=hosted`: constructors, destructors and calls exit when main returns
  - `--crt0=minimal`: no constructors, spins if main returns

amazon

I was a bit surprised when I wrote my first Cortex M3 code that I actually needed to initialize the processor, fill in the interrupt vectors and initialize memory before starting my application. On 8- and 16- bit processors, this code is usually provided for you along with the compiler and C library. Like so many others, I wrote an application-specific version of the necessary code and compiled that with my application.

For Picolibc, I decided to take that and adapt it to be a bit more general, adding a default interrupt vector, initializing any FPU as well as setting up TLS. Now it could be used on most Cortex-M processors.

After that, I wrote equivalent code for RISC-V and aarch64. Then discovered that there are non-Thumb ARM processors which require completely different initialization code, so I added that as well.

Finally, my friend Mike Haertel wrote startup code for x86 CPUs, both to i686 as well as amd64 mode. I bet very few of us have ever considered the x86 as a bare-metal processor.

As usual for embedded programming, these need to be customizable depending on the target environment. They are built three ways depending on the API support required; 'normal' which invokes constructors but spins if main returns, 'hosted', which also calls exit(ret), passing the return value from main() and 'minimal' which doesn't even bother to call the constructors. I use the 'hosted' version when running under qemu to capture the exit status from test applications and report that to the test framework. Most of my actual embedded applications use the normal variant as I often have a constructor or two.

Of course, some environments need even more control over startup and end up writing their own code. For them, the picolibc startup code can show what they need to do to support the library.

Linker script

- Provide defaults for simple applications
  - User specifies RAM/ROM memories
- Allows configure tests to succeed for other projects
  - gcc hello-world.c builds and links (but doesn't usually run)
- Builds initial TLS block out of static memory
  - With clever linker script hacks
- Demonstrates requirements for more advanced users

amazon

I sometimes think that linker scripts are a dark art in embedded development world. The scripting language is a bit unusual, and the contents depend greatly on the implementation details of the C and C++ compilers, along with details about how libc works.

Gathering all of those requirements together and writing a working script often takes many iterations and experiments, and even then many scripts I've written only work by accident.

For picolibc, I've tried to create a sample linker script that shows all of the pieces necessary to build successful C and C++ applications, including handling thread-local storage and C++ exceptions.

The only information the developer needs to provide is ROM and RAM addressing and sizes, and perhaps the desired size of the stack. For more sophisticated developers, the sample script should explain the language and picolibc requirements and help shorten the time to develop their own script.

This script expects applications to put objects in special sections, for instance the NVIC interrupt vector in Cortex-M processors usually needs to be the very first thing in flash so that the reset vector and initial stack pointer can be loaded. The script also expects many symbols to be defined so that the code knows how memory is arranged, for instance, the startup code that sets up RAM needs to know where the contents of initialized data are in flash and where they need to go in RAM. These are all defined in the picolibc documentation, but using the sample linker script can help clarify how to set things up.

semihosting

- Interface to host OS via debugger or QEMU
  - Full support for v2.0 on ARM and RISC-V
- Console and file I/O
  - Printf debugging even before clocks are running
- _exit
  - Passes exit status through qemu

amazon

Semihosting allows a program running in a debug environment to communicate with the debugger. This can be used to provide debug output and status information even before the system clock and I/O devices are running. Semihosting works with a hardware debug link like JTAG, or from within QEMU.

To use semihosting, the program places information in registers and memory and then invokes a breakpoint or other instruction which triggers a debugger-visible event that halts the processor. The debugger can then use the register and memory contents to perform an operation, be that writing to the console or reading and writing data from the debugger host file system.

ARM and RISC-V share the same spec which picolibc fully supports. x86 has a much smaller set of operations, limited to writing to the console and terminating execution.

Testing

- Target testing
- Running tests
- Test status

Testing is key to the success of any system software like picolibc. Because most of the library will not be used by any one application, it's difficult to do in-situ evaluation of the code. Instead, explicit low-level tests are needed to ensure the library conforms to the expected definitions.

Fortunately, C libraries have several comprehensive specifications that define library behavior. This makes building a test suite a matter of identifying statements in the specifications that can be converted to assertions, then writing code to test them.

Newlib had an existing test suite which was not working. Fixing those tests and getting the code passing again took several months. The code fixes were shared with newlib, but there was no way to fix the test suite in newlib itself as it doesn't have any way to run tests on the target systems.

To validate the tests, they are run against both picolibc and glibc.

Testing on Embedded Targets

- Run tests on bare metal systems
  - Test actual target architectures
  - Runs target-specific code (asm mostly)
- Requires built-in bare-metal support
  - Startup code
  - Linker scripts
  - Exit status reports
- Run on QEMU to avoid building an IOT farm.
  - Have extensions to QEMU to support more ARM architectures. Rejected upstream.
  - Using semihosting for ARM, RISC-V and x86

Picolibc contain numerous architecture-specific code paths, optimizing for space and speed using a combination of special C code and hand-written assembly. The target architectures have varying characteristics as well, from floating point support to sizes of basic data types and even signedness of 'char' and 'wchar_t' types.

Making sure all of this code operates correctly can only be done by executing it on all of the target architectures. Picolibc does this by building bare-metal tests for each architecture and then running them under QEMU which avoids needing a pile of sketchy embedded hardware for testing.

Bare metal code requires hardware specific support in the form of startup code and linker scripts. Finally, the tests also need some way to report status back to the test harness, which uses the semihosting support in QEMU.

QEMU includes support for numerous ARM architectures, but not all of the supported CPU types are actually available. Any non- Cortex-M CPU can be used in 'none' machine, but there is no equivalent for cortex-m cpus, which are only available in full SoC models. As a result, only the Cortex-M3 CPU can be tested in upstream QEMU. I submitted a patch adding a 'virtm' machine that enabled testing of all Cortex-M models in QEMU but that was rejected as out of scope for QEMU.

Test Status

- newlib includes over 74000 tests
  - Thousands (and thousands!) fail
  - Not obviously used in decades
- picolibc has fixed these
  - All pass on ARM, AARCH64, RISC-V and x86 today
- Many new tests added
  - printf/scanf validation
  - math errno/exceptions
  - malloc stress test

Within the upstream newlib source are thousands and thousands of tests. Essentially none of these work, having been neglected for years.

The math tests appear to have been translated a couple of times, corrupting the expected values; places which should have been NaN or Inf had been replaced with 64.0. Fortunately, the test code also contains paths to compute the expected values by calling the appropriate routines. I regenerated all of the tables by using glibc as an oracle.

New tests have been added to test new code as well as in response to bug reports. With a major re-write of the nano-malloc code, extensive malloc validation and stress testing was used to make sure it works in a wide range of environments. An external printf test suite was incorporated to make sure changes to printf continue to produce the expected results. Finally, a sequence of bug reports against newlib's math routines which produced incorrect exception and errno results led to the development of tests against all libm functions to make sure they produce the right errno and exception values.

To make sure the tests continue to work correctly, they are built against glibc and the output verified for that library as well as picolibc.

With the tests reconstructed and a test harness built that uses QEMU to run them, all of the tests now pass on ARM, AARCH64, RISC-V and x86 today.

GitHub actions

- Linux testing using GCC and Clang
  - GCC: native, aarch64, amd64, arm (23 arch), lx106, riscv (36 arch), rv32imac, i686
  - Clang: thumbv7m, thumbv7e+fp, rv32imafdc, rv64imafdc
  - 19 config flag settings
  - 59 * 19 = 1121 complete picolibc builds
- Mac OS X testing
  - Build 13 config flag settings using native compiler

While the official upstream host for picolibc is at keithp.com, there is an active picolibc project at github which accepts bug reports and merge requests. There are github actions that tests on each change to the 'main' branch and all merge requests to 'main'. This allows problems to be caught before being merged, offering confidence that the library continues to work as expected even across fairly significant changes.

github appears happy to do a **lot** of testing for the project. Tests are built and run on aarch64, amd64, arm, riscv and i686. Tests are built on lx106. In addition, most of the tests can run 'natively', using the underlying system C library to provide POSIX functions. The actions build and run tests on both Linux and Mac OS X systems using this mechanism. In total, picolibc is built 1121 times and the entire test suite is run 1026 times.

Recent Picolibc Work

- libm merged into libc
- hex format float support
- -Wextra cleanups
- C++ fixes

The math library was merged into libc.a because some of the libc functions wanted to use math functions. This means that applications need not use -lm to get access to math functions. A stub libm.a is included to provide backward compatibility.

Printf and scanf gained support for hex-format floats (%a in printf). The printf support is designed to produce the same output as glibc does, which is easy to understand for those of us familiar with IEEE 754 floating point formats.

An ongoing source of trouble has been making C++ applications build with picolibc. Some of that has to do with how GCC sets up include paths, and how the libstdc++ headers are written. For crosstool-ng, I've actually got a patch to GCC that removes uses of '#include_next' in header files as that would cause the compiler to find non-picolibc headers following the libstdc++ headers.

For the last week or so, I've been working to clean up the library so that it builds cleanly with the -Wextra compiler flag. Quite a bit of this work was caused by old style code which used inappropriate types, like 'int' for sizes or wint_t instead of wchar_t. This found a number of possible bugs, but nothing that would likely cause problems in real use.

hello.c

```
#include <stdio.h>

void main(void)
{
  printf("hello, world\n");
}
```

Now it's time to show some samples of how picolibc can be used. This first one should be familiar to anyone who has learned to program in C. hello.c is the classic 'First C Program' which can be used to verify a toolchain including the C library and the runtime environment. Because the string passed to printf is constant, and ends with a newline, GCC optimizes this into a call to puts, so the resulting binary doesn't contain printf at all.

stm.ld

```
__flash =      0x08000000;
__flash_size = 0x00020000;
__ram =        0x20000000;
__ram_size   = 0x00004000;
INCLUDE picolibc.ld
```

This linker script describes the memory layout of an STM32L152RBT6 Cortex-M3 SoC. It shows 128kB of flash starting at 0x0800_0000 and 16kB of RAM starting at 0x2000_0000. The picolibc linker script, included at the end of this file, takes those values and lays out the application in memory.

Flash contains instructions, constant data and initialization values for the RAM. At program start, those initialization values are copied to RAM and the zero-initialized area of RAM is cleared.

If your application is fairly simple, this style of linker script will probably work fine. If not, you can use the picolibc.ld script as a guide to what picolibc needs.

It might be nice to include a collection of linker scripts like this one with definitions for common microcontrollers; that would be one less step for starting a project.

Compiling

```
arm-none-eabi-gcc \
  --specs=picolibc.specs \
  --oslib=semihost \
  --crt0=minimal -Os -g \
  -mcpu=cortex-m3 -Tstm.ld \
  -o hello.elf hello.c
```

This command invokes gcc to compile the application and link it with picolibc. The –specs=picolibc.specs option loads a file adding command line parsing for picolibc options like --oslib= and --crt0= as well as telling the compiler how to find the right picolibc files.

--oslib=semihost adds the picolibc libsemihost.a file after libc.a to connect picolibc to the semihosting environment. There's also a dummyhost library which provides stubs for tinystdio so that you can link an application using stdio without any OS layer. -Tstm.ld uses the linker script to arrange the program in memory.

Size of hello.elf

File	ROM	RAM
hello.elf	352	24

amazon

This shows the amount of Flash and RAM used by the sample application. Because printf has been replaced with puts by the compiler, the binary is surprisingly small. This is a complete Cortex-M3 image that runs on real hardware and includes the interrupt vector and semihosting interface routines.

Running on hardware

```
openocd -f board/stm32ldiscovery.cfg \
   -c "program hello.elf" \
   -c 'arm semihosting enable' \
   -c 'reset run'
```

This script loads the program in flash and starts it running while monitoring the device for semihosting requests. Console output from the application appears on stdout. This example is unusually short because openocd includes information about the stm32l discovery board, which includes a built-in debug connection which avoids needing to specify both SoC and debug dongle information.

Running under qemu

```
qemu-system-arm \
   -chardev stdio,id=stdio0 \
   -semihosting-config enable=on,chardev=stdio0 \
   -monitor none -serial none \
   -machine mps2-an385,accel=tcg \
   -cpu cortex-m3 \
   -device loader,file=hello-qemu.elf \
   -nographic
```

You can also run this application under qemu. That requires re-linking using a different script as the memory addresses supported by the mps2-an385 QEMU model are different from the STM32L152RBT6 with flash starting at 0x0000_0000 instead of 0x0800_0000. Otherwise, the .elf file is identical. QEMU also supports semihosting, which can be enabled on the command line. A script like this is how picolibc runs all of the tests during development.

Using the float printf code

```
#include <stdio.h>

void main(void)
{
    printf("%g\n", printf_float(355.0f/113.0f));
}
```

amazon

Here's a program to show the size of picolibc's printf function. Unlike the first example, this one includes an argument so that it isn't converted into a call to puts. We'll use a floating point number to show the size of the different versions of printf provided in picolibc. Using a bit of symbol definition magic, picolibc lets you select which type of printf function your application will use at link time.

Because of the way floating point numbers are passed to varargs functions like printf, we need to use printf_float to handle the 32-bit float. When using the float-only printf, this inline function wraps the 32-bit float in a 32-bit int, preventing it from being extended to a 64-bit double. When not using the float-only printf, this function casts to a 64-bit double.

This example can be compiled and run like the hello-world example.

Comparing sizes (cortex m3)

Version	ROM (bytes)	RAM (bytes)
Full	6872	24
Float	5360	24
Integer	1400	24
Newlib Float	15584	1200+1084
Newlib Integer	5760	800+1084

amazon

This table shows the size differences of the variants. The first is the full version of printf, supporting 32 and 64 bit floats. The second only provides support for 32-bit floats, the third has no floats and doesn't support 64-bit ints. Adding 64-bit int support increases the size by 400 bytes.

I also included sizes for the original newlib code, both full float and integer-only. These show the initial RAM size plus the amount of memory allocated from the heap.

You can see from these numbers why many developers avoid printf and stdio, often writing custom I/O routines. Adapting the AVR stdio and printf code, along with the Ryu floating point conversion code, provides a stdio that fits in most ROM budgets while using very little RAM.

Blinky Lights!

```
#include <stdint.h>

typedef volatile uint32_t vuint32_t;

#define RCC_AHBENR              (*(vuint32_t *) 0x4002381c)
#define RCC_AHBENR_GPIOBEN      (1)

#define GPIOB_MODER             (*(vuint32_t *) 0x40020400)
#define GPIOB_ODR               (*(vuint32_t *) 0x40020414)
#define MODER_OUTPUT            (1)

void delay(int count) { while (count--) __asm__("nop"); }

void main(void)
{
    RCC_AHBENR |= (1 << RCC_AHBENR_GPIOBEN);
    GPIOB_MODER |= ((MODER_OUTPUT << (6 << 1)) | (MODER_OUTPUT << (7 << 1)));
    GPIOB_ODR = (1 << 6) | (0 << 7);

    for (;;) {
        delay(40000);
        GPIOB_ODR ^= (1 << 6) | (1 << 7);
    }
}
```

File	ROM (bytes)	RAM (bytes)
blink.elf	264	8

Here's one last example, which is more like a typical embedded application. This program blinks the blue and green LEDs on an STM 32 L Discovery board by accessing the GPIO controller on the SoC, programming the two pins connected to LEDs as outputs and then toggling them with a delay loop to slow things down enough to see.

This example doesn't use much of the library, but it does use the startup code and linker scripts, so even this short example is easier to write with picolibc than without.

Projects with picolibc support

- GCC
- crosstool-ng
- RIOT
- zephyr
- FreeRTOS
- qemu
- personal projects

amazon

Now I want to talk about work done in other projects related to picolibc, including toolchains, operating systems and hardware emulation.

Picolibc and C++

- libstdc++ iostream used a mix of stdio and POSIX apis
  - Directly accessed private FILE struct fields
  - Supported only glibc and newlib
- --enable-stdio=stdio_pure uses only stdio functions
  - Supports any stdio implementation
- picolibc also supports c++ exceptions

amazon

Lots of people are using C++ for embedded applications, and they often want to use higher level libraries, including libstdc++. One of the pieces of libstdc++ is <iostream> which provides abstractions on top of regular C stdio. The GCC libstdc++ implementation has performance optimizations which penetrate the stdio API abstraction with custom code for both glibc and newlib, directly manipulating private fields within the FILE struct, and bypassing the C library with direct calls to the underlying POSIX API. I submitted patches to GCC which eliminated these optimizations. This is selected by building GCC with –enable-stdio=stdio_pure and is included in GCC 11.

CompCert support

- CompCert is a formally verified C99+ compiler
- It is not GCC (or Clang) compatible
- Numerous changes were required in picolibc to make the code compatible with this standard.

amazon

The Inria CompCert project is building a formally verified C compiler for embedded software. Projects using this compiler would prefer a C library also built with the verified compiler. CompCert developers have provided patches to picolibc that allow it to be built with their compiler. Most of these changes fix GCC/Clang specific language constructs and so they have generally improved the portability and C standards conformance of the code.

Crosstool-NG

- Toolchain building system used by Zephyr (among others)
- Added support for simultaneously building newlib, newlib-nano and picolibc
- Also builds libstdc++ for all three C libraries

amazon

Crosstool-ng is a cross compilation toolchain building system. Select a target system and a suite of libraries and crosstool-ng will download the bits, build the host tools, compiler and libraries creating a self-contained toolchain ready to build your embedded projects.

Zephyr uses crosstool-ng to build its native toolchain.

One of the assumptions built into Crosstool-ng is that there is only one C library. For most users, that's pretty reasonable as they only need to use one C library to build their projects. And, that's how initially integrated picolibc into crosstool-ng; offering another C library alternative to newlib and newlib-nano. With the resulting toolchain, I managed to build and test numerous applications, including the Zephyr tests.

However, that's not how the Zephyr project was using Crosstool-ng. Within the Zephyr fork of crosstool-ng was a kludgy patch to build both newlib and newlib-nano. This allowed Zephyr users to download a single toolchain and decide while building their Zephyr project which C library to use.

I decided to work from there and create a mechanism that could be upstreamed into crosstool-ng that would build all three C libraries, newlib, newlib-nano and picolibc and offer users a mechanism to select between them while building an application with the toolchain. Working with the crosstool-ng community, we ended up using the 'companion library' mechanism for newlib-nano and picolibc to build these in addition to the core C library. You can also ask to build only one of the three, although the configuration mechanism is a bit klunky. Taking advantage of the libstdc++ patches incorporated into GCC, crosstool-ng also builds libstdc++ for all three libraries.

All of this work is now in crosstool-ng.

RIOT

- Embedded OS targeting IoT applications
- Save 5kB compared with newlib
- Adds TLS support using Picolibc routines

hello-world	ROM (bytes)	RAM (bytes)
newlib	10956	1880
picolibc	5284	1788

amazon

RIOT is a an OS for IOT devices. There's a fun group of people working on it and I discovered that they were already working on integrating picolibc in the summer of 2020. I joined in and helped answer a couple of questions. That code got merged in August 2020 and has seen a steady uptick in usage by RIOT developers. Because of the straightforward RIOT codebase, adding TLS support was easy as it could use the Picolibc TLS helpers unchanged. The whole integration work took about 500 lines of new code in RIOT, including the TLS bits.

Reviewing RIOT development over the last year, there have been quite a few patches related to picolibc, so it seems pretty clear that picolibc has taken root in the RIOT community. If you're looking for a great little IoT operating system, I'd encourage you to take a look here.

Zephyr

- LF embedded OS project
- TLS support added using custom code
- Support not upstream yet
- Spurred work on Crosstool-NG
- Uses less ROM than built-in libc
  - Especially if you use printf

amazon

Many of you are probably familiar with Zephyr, the embedded operating system sponsored by the Linux Foundation. I got interested when I discovered that they were getting serious about memory protection stuff. An embedded operating system that took advantage of memory protection hardware available in smaller SoC devices sounds like a good thing to me.

The picolibc integration work started about two years ago. I sent an initial PR that included TLS support for 32-bit ARM processors. Zephyr has a lot more processor-specific assembly code, including much of the context switch code where the architecture-specific TLS hooks needed to go. It was easier to just do that part inline. After submitting the PR, a couple of people who had been thinking about TLS support jumped in and made the rest of the TLS support part of Zephyr, and that series got merged in, but the main picolibc integration hasn't been accepted yet. The initial blocker was lack of toolchain support, which led to getting picolibc integrated into crosstool-ng, which led to fixing libstdc++ to support other libc implementations. By then, I was off working on other things and unable to get back to Zephyr. I've got patches rebased and working again; I'll try to get them posted and see if someone wants to help bring this series home.

Picolibc could easily replace the builtin 'minimal' C library in the Zephyr tree. The picolibc code is smaller and faster in all cases, while also including a math library and other parts of a standard C environment. This could also replace the cbprintf and printk paths in the system, reducing ROM footprints quite a bit while improving standards conformance.

Personal Projects

- AltOS
- ChaosKey
- LambdaKey
- Snek and SnekBoard

amazon

I started the picolibc project because I needed a solid C library for my own microcontroller projects. Here's a list of projects I've built using picolibc.

First is AltOS, which is a suite of firmware built into products that Bdale Garbee and I sell for use in high power amateur rocketry. This firmware is also included in several amateur radio satellites currently in orbit. While we started with the venerable 8051 microcontroller, we've almost completely migrated to ARM Cortex-M processors, ranging from the STM32F042 through the STM32L151. We've started looking at some Cortex-M4 processors for some applications requiring higher performance but haven't gotten there yet. If you're interested, check out altusmetrum.org

Next, we built ChaosKey, which is a USB-connected True Random Number Generator built on the STM32F042 processor. We built a thousand of these and sold them to people around the world. For that same hardware platform, I wrote a tiny Scheme interpreter and called the result LambdaKey. It was largely a gag, but it did create enough language infrastructure for my next project.

That next project was Snek, a subset of Python that runs on a huge range of small microcontrollers, from the Atmega 328P in the original Arduino to the SAMD21 chips found in a range of Adafruit products. Snek is also the primary firmware for SnekBoard, also based on the SAMD21 processor. SnekBoard has motor controllers and I/O pins designed to work with Lego PowerFunctions hardware. I ran a CrowdSupply campaign so that I could build a bunch of these boards and started using them in a middle school Lego robotics class just as the pandemic began. I'm hoping to get back to teaching kids as soon as I can.

Picolibc/newlib relationship

- Newlib changes regularly merged to picolibc
- Some picolibc changes merged to newlib
  - Bug fixes in common code
- Newlib cannot accept ABI/API changes

The picolibc code has tried to remain reasonably source compatible with newlib so that patches can be exchanged easily. For example, a bunch of bug fixes for exceptions and errno support in math functions have been passed from picolibc back to newlib. Some patches require editing to get them building in newlib again. Without a comprehensive test suite, it's difficult to know if they work though.

Changes from newlib are added into picolibc from time to time; few of the patches are relevant as most newlib work relates to cygwin support, which has been removed from picolibc.

I'm happy to continue supporting the newlib project; much of the code for picolibc had it's origins there, and giving back to a community that has continued to improve so much useful code for so many years seems like the best way to thank them.

Picolibc Aspirations

- Continue to improve the code
- More accurate math functions
- Integer-only math implementation
  - Faster for soft float cores
- Include SoC-specific configuration bits
  - linker script, compiler flags
- Broader adoption in embedded environments

There are always things to do with an old body of code like picolibc. Some of the functions are still declared using K&R syntax. Much of the older and less frequently used code (like the search stuff) is still written with a casual approach to typing leaving questions about integer overflow safety and the like.

Similarly, the math library is showing its age, with numerous functions less accurate than many other C libraries (like musl and glibc). This is especially true for the odd corners of the library like tgamma and the bessel functions. Picolibc can't simply adopt the glibc implementations given the licensing difference. I would also like to see integer-only implementations of some of the more important math functions for use on soft-float hardware.

To help people starting with a new device, it would be nice to include linker scripts and compiler options for a range of embedded SoC parts. That way, you wouldn't have to dig through datasheets to discover where memory was mapped and what kind of core is provided.

Finally, we're starting to see more activity with people using picolibc, but it's still a pretty quiet project.

And that's the end of my talk. I hope you found it useful and I want to thank you for taking time to listen today.