SEE TEXT ONLY

FreeBSD and RTEMS, UNIX in A REAL-TIME OPERATING SYSTEM

By Chris Johns, Joel Sherrill, Ben Gras, Sebastian Huber, Gedare Bloom

RTEMS developers have a long history of using FreeBSD code in the RTEMS source base and the results have been very good. RTEMS is not alone in doing this, and the challenges and issues RTEMS faces using FreeBSD source code are the same that others face.

RTEMS is a POSIX real-time operating system, or RTOS, and like all open-source projects has limited resources. All available effort needs to be focused on the real-time parts of the operating system. RTEMS looks to use quality, suitably licensed source code where possible—a combination of code from Newlib, RTEMS, and the FreeBSD TCP/IP stack—to provide a surprisingly robust subset of POSIX.

What follows is a summary of the history, what has been accomplished over the last 20 years, and how integration problems were solved while minimizing the maintenance burden.

RTEMS

RTEMS stands for Real-Time Executive for Multiprocessor Systems and has been open-source software since 1991. RTEMS has been used in some significant applications over the last 25 years and the project has a healthy community. It was developed by On-Line Applications Research (OAR) in Huntsville, Alabama, under contract from the U.S. Army in response to program managers watching missiles head down a test range and explode—each bundled with an operating system license.

RTEMS started life as a traditional embedded RTOS kernel typically found in the late 1980s and early 1990s. The user API was based around the VMEBus Industry Trade Association (VITA) Real-Time Executive Interface Definition (RTEID) 2.1 standard. The kernel's C source code was cross-compiled to a library, and a single standalone static executable was created by linking the application's object files with the RTEMS library. Embedding usually involved burning the binary image into an EPROM using some type of external programming tool.

An application and the kernel shared the same address space, and there was no memory management or virtual memory. An RTEMS application was statically linked with the OS and operated in the same single address space. The single-address-space environment was like a single process with no protection and a 1:1 mapping for the virtual addresses to the physical address space. The first versions of RTEMS had no C Library or filesystem, no networking, and device drivers and hardware support were often custom for the specific hardware and considered part of the application.

RTEMS has supported multiple processors for 20 years. The multiprocessor support is based around separate address spaces running on separate processors interconnected using a bus architecture and message passing. Applications can access and manage resources distributed across a number of nodes.

RTEMS has evolved and grown and is now much more sophisticated. However, some of these fundamental aspects of embedded, real-time systems remain and are valid.

Today RTEMS has symmetric multiprocessing (SMP) support. In some respects, the easy part is implementing SMP support in the OS because designing deterministic, real-time SMP applications is even more challenging. The problems in the OS are solved once, but each application faces similar challenges related to race-condition bugs, resource contention, and effective utilization of multiple cores while having predictable, correct, and safe behavior.

Real-time

Today, real-time software and real-time operating systems are as important as ever. The development of applications for them is alive and well, and so is the development of the operating systems to support these applications. The increase in computing power in smaller and smaller devices has brought powerful, server-grade operating systems such as FreeBSD to small, embedded devices, and these devices can have the performance to meet a range of real-time applications. However, these systems are still not fully deterministic, and the ability to complete a full schedulability analysis is problematic. RTEMS's performance can be deterministic, and it provides a rich suite of scheduling algorithms to meet the demands of real-time applications.

RTEMS has extended its support for high-performance, real-time computing by adding SMP for multi-core processors. Real-time software on SMP hardware is a difficult and complex challenge. To aid the application architects, RTEMS provides a powerful set of configuration interfaces. Cores can be grouped and assigned a specific scheduling algorithm, and threads can be assigned to a core affinity set and therefore associated with a specific scheduling algorithm. The ability to partition the components of a real-time design lets the architects manage the difficult task of schedulability analysis. RTEMS lets interrupts be assigned to specific cores letting the user manage latency when under load.

Kernel Architecture and APIs

The RTEMS Kernel source can be found in the git://git.rtems.org/rtems.git repository. It has four major components:

1. SuperCore
2. Application Programming Interfaces
3. Services
4. Board Support Packages (BPS)

SuperCore

The SuperCore is a super set of functionality exported to users via the various application programming interfaces or APIs. The SuperCore contains all the important, real-time algorithms in RTEMS. It contains the scheduling, locking, synchronization, clock, interrupt and SMP support. It also contains the architecture and CPU support for context switching and low-level interrupt management for each of the supported architectures.

The resources created and managed by the various user interfaces can coexist because everything maps to a SuperCore resource. This means a thread created in one user interface can block on a mutex created by a different user interface. This is a powerful feature, as software components can be implemented by different user interfaces and can be combined into a single executable. The support extends to the scheduling and time domain. This means the run-time profile of an application is independent of the API used to write it.

Application Programming Interfaces

RTEMS currently supports three major application programming interfaces or APIs. They are:

1. RTEMS Classic API
2. POSIX
3. High Performance API

RTEMS Classic API

The Classic API is the original API in RTEMS and is based on the VITA RTEID 2.1 standard. It is a classic, real-time operating system programming interface and provides:

1. Tasks
2. Semaphores
3. Message Queues
4. Events
5. Barriers
6. Interrupts
7. Time
8. Timers
9. Rate Monotonic Periods
10. Fixed Allocation Memory Pools
11. Variable Allocation Memory Pools

The RTEMS Classic API has a low overhead and is often used in small resource-limited targets needing a small footprint.

POSIX

RTEMS POSIX support is divided into three parts based upon where they are implemented. They may be implemented by the:

1. C Library,
2. RTEMS, or
3. TCP/IP stack.

RTEMS uses the same Newlib C library as Cygwin to provide the core C Library and most of the non-thread related POSIX capabilities. This provides a robust implementation of core POSIX and C Library services including math, stdio, and strings, and even wide character support. Much of the Newlib source code originated from BSD operating systems, but has been ported to many target architectures.

Importantly, RTEMS relies upon Newlib header files defined by the C and POSIX standards. Historically, Newlib did not provide a complete set of C and POSIX header files. It only provided those where it implemented some methods. However, there has recently been a push by the RTEMS developers to grow this set and to ensure they are compatible with FreeBSD-kernel and user-space source code.

RTEMS provides the implementation of all concurrency and synchronization capabilities including threads, mutexes, condition variables, semaphores, and message queues. It also implements system calls such as `open(2)`, `read(2)`, etc. In addition, RTEMS provides other POSIX capabilities such as termios, clocks, and timers.

RTEMS relies upon the FreeBSD TCP/IP stack to provide the networking APIs required by the POSIX standard. The older IPV4-only stack does not provide all of the capabilities required by POSIX. However, the new TCP/IP stack provided by the LibBSD project deserves credit for providing complete support.

As a single-process, multithreaded operating system, RTEMS is aligned with POSIX profiles PSE 51 and 52. These profiles define the services provided by single-process, multithreaded POSIX implementations. PSE 52 includes filesystem support, while PSE 51 does not. RTEMS applications may optionally disable all filesystem support. Thus, RTEMS can be user configured to align with either profile. FreeBSD is a multiprocess, multiuser POSIX implementation and aligned with POSIX profile PSE 54.

The Open Group Future Airborne Capability Environment (FACE™) Consortium has defined four new POSIX profiles to address the requirements of the avionics software community. These profiles reflect existing real-time operating systems and applications that have achieved certifications in industries such as avionics and medical devices. These profiles reduce the approximately 1,300 methods in the POSIX standard to meet the certification and application requirements found in existing avionics applications:

• The Security Profile is small with only 163 methods required. It is designed for multithreaded, single-process applications such as an information gateway device.
• The Safety Base Profile is larger with 246 methods required. It is designed for multithreaded, single-process applications.
• The Safety Extended Profile is larger with 335 methods required. It is designed for multithreaded, multiprocess applications.
• The General Purpose Profile is larger with 812 methods required. It is designed for multithreaded, multiprocess applications which may not have any certification requirements or less rigorous ones.

Being a single-process, multithreaded operating system with a long history of standards support, RTEMS naturally aligns with the Safety Base Profile. When initially evaluated, RTEMS was missing less than 10 methods from this profile. There is currently an effort to integrate RTEMS and the ARINC 653 Deos RTOS and achieve FACE conformance for the Safety Base Profile.

Interestingly, when evaluated against the single-process FACE General Purpose profile, RTEMS does surprisingly well, supporting approximately 90% of the methods required. Proper support for required capabilities such as `fork(2)/exec(3)` and process groups are beyond the target profile for RTEMS. However, most of those missing methods are methods that do not require multiprocessing. Newlib does not support fenv.h or long, double-complex math. These account for most of the missing methods that RTEMS could support.

High-Performance API

The close coupled High-Performance API is new and not considered a general API for use in applications. It is used in places where speed and compatibility can be traded off. The FreeBSD port is an example as well as back ends for C11/C++11 threads and OpenMP. C11/C++11 threads currently have the lowest space and time overhead in RTEMS.

Services

Services provide functionality to help developers create useful applications. The support can range from implementation of specific protocols such as

SNMP, additional languages such as Lua or Python, or important services like NTP.

Board Support Packages

A Board Support Package or BSP implementation is the code and support needed to implement RTEMS on a specific piece of hardware. RTEMS contains over 170 BSPs on 17 architectures. The BSP manages the entry from the boot loader, setting up of memory, caches, console, and it contains a timer driver for the system tick. If the processor has more than one core, then it also manages the starting of the extra cores.

While most FreeBSD users will never need to delve into the details of the device drivers for their hardware, RTEMS users often use custom hardware and thus are responsible for developing the drivers for their own boards.

FreeBSD in RTEMS

FreeBSD is an important part of RTEMS and its history. BSD code in RTEMS can be traced back to code and files being added to the Newlib C library. Since then RTEMS has taken FreeBSD directly into its source base in a number of areas. As well as the C Library, RTEMS shell commands such as `rm`, `cp`, and even `dd` are among the growing number of consumers of this code.

Networking Stack

In the late 1990s, Eric Norum, working at the Canadian Light Source in Saskatoon, started to look for a networking stack for RTEMS. Eric is a member of the EPICS project, which requires a networking stack. His first efforts included porting a stack called KA9Q, which was not that successful from both a performance and licensing viewpoint. At that point in RTEMS's history, applications were always statically linked with the OS. Also, embedded real-time applications are shipped integrated with hardware devices. There is generally no desire to redistribute any source code for the application or any supporting software. These deployment characteristics have always led the RTEMS Project to carefully evaluate the license for software incorporated directly or provided as third-party add-ons. For KA9Q, in addition to performance issues, there was never clarity on precisely what the license of the KA9Q stack actually was. This resulted in the KA9Q effort being abandoned.

Eric Norum started to look at porting the Linux network stack. However, that effort did not last because of the license, so it was suggested he look at the FreeBSD stack. Over the course of six months Eric ported the FreeBSD networking stack to RTEMS. This stack is still present in RTEMS and is essentially unchanged with specific bug fixes made as required. This network stack has been very successful and has given RTEMS all the networking sophistication available from a standard FreeBSD installation.

FreeBSD's stack is fully featured, and when ported to an RTOS users can create applications which have a system-level robustness. There are very few system configurations that cannot be done in RTEMS—from a single flat network endpoint to routing, DHCP, radio VoIP, and IP over GRE tunnels over SDH management channels.

This initial port, or what is now called the legacy port, of the FreeBSD networking code used a three tasks model. A networking task or thread runs within the networking stack and lets the stack handle things like ICMP packets and TCP retransmission. Each interface has a receive and transmit task. The receive task receives data from the MAC delivering it to the stack and the transmit task empties the output queue sending the `mbufs` to the MAC. All networking tasks have the same priority and need to hold a single networking semaphore when running. This design means there is limited networking concurrency, and throughput with more than one interface is limited, as the semaphore serializes all processing. However, the implementation is safe and reliable in a re-entrant threaded system and in practice it works very well.

Maintenance

In the years following the completion of this port, some issues began to be observed. The first and most obvious was the availability of drivers. The custom driver support required custom drivers to be written for RTEMS. Code could be borrowed from FreeBSD; however, drivers often needed to be written and tested as if new. As time moved on and FreeBSD evolved, it was frustrating and confusing to RTEMS users that a port of the FreeBSD stack did not support drivers from newer FreeBSD versions.

When the code was ported to RTEMS, it was copied into a simplified directory structure. The FreeBSD directory tree is large and wide and there seemed no point in creating a wide, sparse tree to host a small collection of files. This made it difficult to easily compare files with the FreeBSD originals because it was never really clear which files in the RTEMS source should be compared against which files in the FreeBSD source. Code was changed to get it to build and this was done without any clear indication of what was original and what was changed. As time went on and FreeBSD improved its stack, for example, adding IPv6, it was practically impossible to move from the snapshot in the RTEMS source tree. Specific bug fixes were brought in by hand for localized fixes, but this was a time-consuming process and narrowly focused. This resulted in a maintenance problem that grew increasingly worse as the code aged.

Till Straumann, working at Stanford's SLAC National Accelerator Laboratory, built an additional library called `libbsdports` that allowed drivers from FreeBSD to be used with minimal changes. This was remarkably successful and it raised the idea of being able to use FreeBSD code with minimal changes. His work was limited to networking drivers and on a specific range of architectures, but was the first public example of this being achievable.

There were other FreeBSD activities happening around this time. Chris Johns took the USB stack and ran it on RTEMS on a NIOS-II. However, these were isolated, specific, and of no long-term value to the RTEMS Project. It also highlighted the issue of having separate ports of specific pieces of the FreeBSD kernel. How are the separate pieces brought together in a single static executable? RTEMS needed a single, unified port of the FreeBSD kernel code for all the different parts of interest. Consequently, a plan was developed to address the use of FreeBSD source in RTEMS. The first decision was to take a step back and not fragment the effort. The various parts of the FreeBSD code base that would be used by RTEMS needed to be together and maintained as a single entity. This project was named RTEMS LibBSD.

RTEMS LibBSD

The most recent FreeBSD porting project in RTEMS is called RTEMS LibBSD or simply LibBSD. The project is hosted in a separate Git repository within the RTEMS Project's Git server. The repository is at https://git.rtems.org/rtems-libbsd.git. It is a combined effort led by Joel Sherrill and Sebastian Huber.

The project creates a single port of FreeBSD for RTEMS and provides a range of features present in FreeBSD useful to RTEMS such as networking, USB, SATA, and MMC devices.

As the effort progressed, a broad set of rules was developed to guide developers working on RTEMS LibBSD. The rules slowly took shape as the team found what worked and what did not work. The rules can be summarized as:

1. The directory structure of the RTEMS LibBSD code must match the source tree in FreeBSD.
2. All changes in the RTEMS version of the code must be bounded by a standard conditional defined syntax. This allows removal of the RTEMS changes and comparison of the source with the original FreeBSD code using Python scripts.
3. Do not edit the FreeBSD code including any white-space changes. Make all edits in preprocessor conditionals.

The ability to use original FreeBSD source transparently is central to the work done in LibBSD and the term "source transparency" has been adopted to describe the approach. Anyone embedding FreeBSD code with their own system and machine headers wants to be able to take a subset of the FreeBSD source files and build them without any changes. Currently this is not possible. When viewing the FreeBSD source code from the RTEMS Project's point of view, any transparent source has no changes, and as changes are made, the original FreeBSD becomes less visible.

The major items to resolve when embedding FreeBSD kernel code are:

1. Header files and required declarations. The system and machine header files for an RTEMS target do not match the header files used by the FreeBSD kernel code. The improved standardization of headers has helped on both sides, but there is a range of kernel types and defines that need to be added.
2. The use of standards-based, userland function names with differing signatures in the kernel, for example, malloc. RTEMS is a single address space, statically-linked executable, and these name clashes need to be managed and often are—with horrible hacks.
3. Supporting the SYSINIT initialization used in FreeBSD. This requires linker support to get the correct section management in place. The way this is done in FreeBSD is so good that something similar was adopted in the RTEMS kernel and with impressive results. RTEMS has adopted the linker mechanism to initialization used by FreeBSD with SYSINIT. Previously RTEMS required users to manage in their build system the parts of RTEMS they needed linked in, and for those parts they did not want, they needed to link dummy versions. With the linker set initialization this is all automatic, including the order in which the initialization happens, making RTEMS simpler to use. Although there had always been a focus on small executable size, the size of the RTEMS minimum reference application decreased in size thanks to the change to SYSINIT style initialization.
4. SMP support requires that some parts of the FreeBSD port be managed in the context of RTEMS SMP support.

CONTINUES NEXT PAGE

5. FreeBSD userland code in a single, statically-linked executable requires some interesting hacks to avoid global symbol clashes and to make initialized variables work. This is especially obvious when porting shell commands into a single address space. Each command has its own `main()`, and calling `exit()` does not exit the command invocation—it exits the entire RTEMS application.

Source Code Management

The source tree consists of four major directories. They are:

1. `freebsd` – RTEMS's FreeBSD source code
2. `rtemsbsd` – RTEMS's FreeBSD support source code
3. `testsuite` – Tests for RTEMS LibBSD
4. `rtems_waf` – RTEMS's waf support for building against RTEMS BSPs.

When using LibBSD, the Git sub-module for `rtems_waf` must be initialized to bring its support files into the cloned repository. The module helps configure and build LibBSD for a BSP. A RTEMS toolchain and suitable board support package (BSP) also needs to have been built and installed.

Developers working on LibBSD will also need to clone the FreeBSD source tree from the snapshot point. This creates an extra directory called "freebsd-org."

The RTEMS LibBSD project has around 850 build targets. To manage this number of files and the complex set of compile time defines, architecture specific files, and header files, the definitions are isolated into a single file that is independent of the build system. The initial development generated a `makefile` and recently this was changed to generate a `waf` script (https://www.waf.io). The `waf` build script integrates with the "`rtems_waf`" support, which makes it easier to extract and use the various machine specific compile flags a BSP has. As RTEMS has a single address space, it is critical that the operating system, user code, and any libraries like LibBSD be built using the same ABI.

The LibBSD build definition is a single Python file called `libbsp.py`. The file contains a number of module definitions and these modules are passed to the source management tool to manage moving source in and out of the original FreeBSD source tree to the RTEMS FreeBSD source tree. The module data is also passed to the build script generator to create the `waf` build script.

There is support for a number of different types of modules. Modules can be for RTEMS headers and source, FreeBSD kernel space headers and source, or FreeBSD user space header and source. The module definitions can be given specific compiler flags, or they can be specialized for a specific architecture. A lot of the source is generic to all architectures; however, some files are specific to some architectures, for example, an architecture-specific IP checksum routine.

To add new source files to LibBSD, first initialize the FreeBSD Git sub-module, which populates the freebsd-org source tree with the specific version LibBSD is currently based upon. Run the "freebsd-to-rtems.py" script in reverse to move the modified source in RTEMS FreeBSD to the original FreeBSD source tree. Edit the source definition to add the new files and run the same script in the forward direction. The source will be copied across to the LibBSD tree and the build script will be updated. LibBSD can now be built and the source edited so it can run.

The user space symbol clashes are managed by maintaining a header file that redefines the symbols to a different name space. This is not ideal, but it can be made to work. The clashing symbols, however, raises a common issue, the need to include some header files before any FreeBSD code appears. Having to change the standard code to include headers accounts for a reasonable percentage of the changes made to the original source code. An empty header in a normal FreeBSD kernel tree would help this type of porting exercise because it could include an RTEMS-specific version that includes the symbol refines. It is a simple change in the kernel that makes a larger difference to the RTEMS project.

FreeBSD Core APIs and RTEMS Mappings

The FreeBSD kernel support is implemented in terms of RTEMS services. The following describes those mappings:

1. Shared/exclusive locks `SX(9)` map to binary semaphores. This neglects the ability to allow shared access.
2. Mutual exclusion `MUTEX(9)` maps to binary semaphores. Non-recursive mutexes are not supported this way.
3. Reader/writer locks `RWLOCK(9)` map to binary semaphores. This neglects the ability to allow multiple reader access.
4. The sleep queues `SLEEPQUEUE(9)` use the FreeBSD implementation with adjustments for RTEMS.
5. Condition variables `CONDVAR(9)` use the FreeBSD implementation (no signals support).

CONTINUES NEXT PAGE

6. Timer functions **CALLOUT(9)** mainly use the FreeBSD implementation. The wheel driver is a RTEMS's timer server routine.
7. Tasks **KTHREAD(9), KPROC(9)** map to RTEMS tasks.
8. Device management **DEVCLASS(9)**, **DEVICE(9)**, **DRIVER(9)**, **MAKE_DEV(9)** uses the FreeBSD implementation.
9. Bus and DMA access **BUS_SPACE(9)**, **BUS_DMA(9)** maps to Board Support Package implementations. A default implementation for memory-mapped, linear access is provided and the current RTEMS heap implementation supports all the properties demanded by **bus_dma**.
10. The Universal Memory Allocator **UMA(9)** is supported using a simple page allocator as backend allocator.

From a maintenance perspective, the RTEMS developers hope that these FreeBSD kernel APIs remain stable. This reduces the risk that updates to FreeBSD require significant work to the RTEMS implementation of the core kernel APIs.

User Space

An important part of using FreeBSD is the user-space support. The commands such as **sysctl**, **ifconfig**, and **netstat** provide an important user interface. Being able to support these types of commands not only makes using the FreeBSD software possible, it allows RTEMS to tap into the wealth of existing FreeBSD documentation. This is an important, but often overlooked, source of reuse.

Porting separate user space programs into a single executable is an involved and delicate operation. As RTEMS has only a single address space available, neither **fork(2)** nor **exec(3)** are available to set up a new process context. Many common assumptions that C code makes cannot be relied on. RTEMS has to emulate a subset of both **fork(2)** and **exec(3)** semantics. This is accomplished with a combination of build-time, link-time, and run-time techniques. LibBSD provides some abstractions in the form of wrapper functions to help.

When bringing this type of code into RTEMS, it is critical that there are no globals. A range of prepossessing tricks can be used and none of them are nice. Any global data needs to initialized for each run of the command. The use of **getopts** needs to be replaced with a nonstandard re-entrant version provided by Newlib. Finally, **main()** is replaced with a function name such as **main_dd()** for the command. A few important other functions like **exit()** or **perror()** are also replaced with calls to the LibBSD command support code.

The generic support for the commands in LibBSD only allows a single command to run at once. While this is not ideal, multi-user access to RTEMS is normally not an issue. The generic support wraps the call to the specific **main()** in a **setjmp()** call. An **exit()** call is implemented by a **longjmp()** call. Since RTEMS is a single-process, multithreaded environment, executing the native **exit(3)** in an application shuts down the entire application.

The resulting code is not pretty in places; however, the results are rather impressive. There is a working network stack with related FreeBSD commands including **ifconfig**, **netstat**, **ping**, and **tcpdump**, which is an interesting command to run on a real-time operating system. Full Gigabit Ethernet performance has been sustained for both transmit and receive on ARM, PowerPC, and x86 systems.

Initialization

As the code in LibBSD becomes more stable and supports more BSPs, users are beginning to migrate to it and RTEMS developers are working on the boot-time initialization. To solve this problem, the decision was made to adopt the standard methods used by FreeBSD.

Recently Chris Johns added code to support **rc.conf(5)** into the source tree. This is a C implementation, as RTEMS does not have a POSIX sh. The benefit of supporting **rc.conf(5)** to initialize the code is the large amount of documentation and solutions available on the Internet. This is a large saving in effort for the RTEMS project in terms of documentation.

Chris Johns also ported **sysctl(8)** as a command and is in the process of adding support to read **sysctl.conf(5)**. As this support matures, RTEMS-specific code to manage various memory size configuration options will be removed. This will both improve portability and lessen the differences in using and configuring LibBSD and FreeBSD.

The Future with FreeBSD

How good is LibBSD currently? The answer to this question depends on the perspective taken.

Any FreeBSD user who needs to use an RTOS will be pleased by the LibBSD efforts. They have a full-featured, high-performance, stable networking stack with IPv6, packet filtering, virtual LAN support, and more with world class performance. Users have access to a wide range of drivers, and device drivers tend to port quickly with few changes.

The RTEMS developers are far more critical. The ideal is being able to use the FreeBSD code with no changes. While this may never happen, it is critical to ensure that the team strives for it because it lowers the maintenance burden. LibBSD is currently stuck on a FreeBSD 9.x version because updating will require more effort than is available. This puts LibBSD two major FreeBSD releases behind (e.g., FreeBSD 10 and 11). This is due to a combination of issues, some changes in FreeBSD and some code that is not clearly tagged.

There are currently 1,295 source files in LibBSD, and, of those, 797, or 61% of the files, have no changes. Of the 498 files with changes, there is an average opacity level of 1.6%. The level of opacity is a simplistic calculation made by taking the number of inserts and deletes as a percentage of the total lines of code including the inserts and deletes. There are less than 50 files with an opacity level greater than 10%. It is fair to say this is a reasonably successful result.

What remains to be done? (list follows)

• Upgrade the current code to better track FreeBSD. It will always be necessary to spend some effort on this. However, RTEMS developers hope that by making a strong case to FreeBSD kernel maintainers for some simple and helpful changes, the LibBSD maintenance burden will be manageable.
• The number of architectures supported by LibBSD needs to be increased. LibBSD currently is known to work on three architectures, while RTEMS itself supports 18 architectures. RTEMS is a good base reference for building on architectures not normally accessible.
• Although not critical from a functional viewpoint, compile-time warnings in LibBSD are currently ignored. The RTEMS build environment is significantly different from the normal FreeBSD kernel environment. However, it would be nice to see the warnings addressed. As always, many warnings are harmless, but some could be indicative of real bugs.

The FreeBSD code base is important, and there are a number of users of the source code reusing it in ways the original authors never imagined. The RTEMS Project openly and at every possible opportunity acknowledges the code RTEMS has taken from FreeBSD. It is an incredible resource and the RTEMS Project is thankful it is available for use. ●

References

VMEBus Industry Trade Association (VITA) Real-Time Executive Interface Definition (RTEID) 2.1 standard. The RTEID 2.1 is archived at ftp://ftp.rtems.org/pub/rtems/people/joel/RTEID-ORKID/RTEID-2.1/.

Newlib C Library. https://sourceware.org/newlib.

Cygwin. https://www.cygwin.com.

G. Gilliand, J. Sherrill. "A Unique Approach to FACE conformance." U.S. Army Aviation FACE Technical Interchange Meeting. http://face.intrepidinc.com/wp-content/uploads/2016/01/DDC-I-OAR-A-Unique-Approach-to-FACE-Conformance.pdf. (Feb. 2016)

"Proceedings of Technology Showcase Held in Huntsville, Alabama on 7–9 August 1990," includes a presentation by a representative of the Army Missile Research Development and Engineering Center at Redstone Arsenal, Alabama, on RTEMS. http://www.dtic.mil/dtic/tr/fulltext/u2/a247043.pdf. (Aug. 1990)

KA9Q. http://www.ka9q.net/code/ka9qnos/.

A. Subbarao. "POSIX—25 Years of Open Standard APIs." http://www.rtcmagazine.com/articles/view/103514.

FACE Consortium products including the Technical Standard, Conformance Test Suite, and other supporting artifacts. https://www.opengroup.org/face.

CHRIS JOHNS is a RTEMS developer and real-time software engineer with an interest in low-level tools and debuggers. He lives in Sydney, Australia, with his family and two dogs. chrisj@rtems.org

JOEL SHERRILL is Director of Research and Development at OAR Corporation in Huntsville, Alabama, and a member of the original RTEMS team. joel@rtems.org

SEBASTIAN HUBER is a RTEMS developer currently focused on the SMP support. He lives in Munich, Germany. sebh@rtems.org

BEN GRAS is a systems security researcher at VU University in Amsterdam, Netherlands, and an RTEMS contributor. He lives in Amsterdam. beng@rtems.org

GEDARE BLOOM is an assistant professor of computer science at Howard University in Washington, D.C., and a maintainer of RTEMS. gedare@rtems.org