Eaton

PowerChain Software/Firmware Portability Guidelines

Copyright © 2006 Eaton Corporation. All Rights Reserved.

NOTICE TO PERSONS RECEIVING THIS DRAWING AND/OR TECHNICAL INFORMATION

Eaton Corporation claims proprietary rights to the material disclosed hereon. This drawing and/or technical information is issued in confidence for engineering information only and may not be reproduced or used to manufacture anything shown or referred to hereon without direct written permission from Eaton Corporation to the user. This drawing and/or technical information is the property of Eaton Corporation and is loaned for mutual assistance, to be returned when its purpose has been served.

THIS DRAWING AND/OR TECHNICAL INFORMATION IS THE PROPERTY OF EATON CORPORATION

APPROVED BY/DATE

<table>
<thead>
<tr>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiated By:</td>
<td></td>
</tr>
<tr>
<td>Prepared By:</td>
<td></td>
</tr>
<tr>
<td>Approved By:</td>
<td></td>
</tr>
<tr>
<td>Approved By:</td>
<td></td>
</tr>
<tr>
<td>Approved By:</td>
<td></td>
</tr>
<tr>
<td>Approved By:</td>
<td></td>
</tr>
</tbody>
</table>
## Revision History

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>12-Sep-05</td>
<td>Matt Creech</td>
<td>Initial revision</td>
</tr>
<tr>
<td>0.2</td>
<td>15-Sep-05</td>
<td>Matt Creech</td>
<td>Added more features and a “best practices” section</td>
</tr>
<tr>
<td>0.3</td>
<td>21-Sep-05</td>
<td>Matt Creech</td>
<td>Various cleanups</td>
</tr>
<tr>
<td>1.0</td>
<td>1-Sep-06</td>
<td>Matt Creech</td>
<td>Clarify STL requirement</td>
</tr>
</tbody>
</table>
# Table of Contents

1. PURPOSE................................................................................................................................................... 4

2. TARGET PLATFORMS.................................................................................................................................... 4

3. BASIC REQUIREMENTS................................................................................................................................ 4

4. PORTABLE FEATURES.................................................................................................................................. 5

   4.1. Guaranteed Features.................................................................................................................................. 6
       4.1.1. C99 Support........................................................................................................................................ 6
       4.1.2. C++ Support.................................................................................................................................... 6
       4.1.3. BSD-compatible Sockets.................................................................................................................. 6
       4.1.4. Descriptor-based Character I/O...................................................................................................... 6
       4.1.5. Volatile and Nonvolatile Filesystem Access................................................................................... 6

   4.2. Recommended Features.......................................................................................................................... 7
       4.2.1. Descriptor-based Block I/O............................................................................................................ 7
       4.2.2. POSIX-like Threading Support....................................................................................................... 7
       4.2.3. POSIX/SysV IPC Support................................................................................................................. 7

5. SOFTWARE PRACTICES.................................................................................................................................. 7

   5.1. Bounded String Functions..................................................................................................................... 8
   5.2. Multi-language Support......................................................................................................................... 9
1. Purpose

When developing a general-purpose toolkit for use by multiple products, there are two competing goals which often dictate different design decisions. The first goal is to maximize portability: given requirements for a set of software tools, we would like the tools to be as generic as possible so as not to limit the choices of hardware and operating system. The second goal is to maximize efficiency: given these same requirements, we would also like the tools to be developed as quickly as possible. In order to reach a compromise between these two goals, a core set of minimal functionality must be established such that:

- The tools are available to any platform the provides this functionality
- The minimal functionality level is not so low as to drastically increase development time
- The minimal functionality level is not so high as to drastically limit platform choice

This document describes this minimal software functionality level in order to set a guideline for developers (who are guaranteed certain features, but nothing more) and hardware designers or OS vendors (who only need to provide certain features, but nothing less). It is not necessary that this chosen set of features be truly universal - there may be compelling reasons to, for example, create a low-cost device that does not meet minimum specifications. But in such a case, the decision to use a reduced-functionality platform will require significant development work in order to port existing software to the system. This guide may be used to provide an approximation of how much work is needed, since software will likely assume that the described functionality is present.

Otherwise, assuming that product engineers provide the minimum functionality and developers do not exceed it, the selected set of features is a point of balance at which:

- Adding more features would increase per-device cost without an appropriate reduction in development costs
- Removing some features would increase development costs without an appropriate reduction in per-device cost

Finding such a balance is obviously not an easy task, and it requires some guesswork. A side-effect of this document is to spur discussion and ensure that all parties are in agreement on the selected platform feature set; if not, it provides enough detail to fuel discussion and help fine-tune requirements for future revisions.

2. Target Platforms

Although any platform which provides the functionality described in this document will support the PowerChain “toolkit”, such functionality is chosen with certain target platforms in mind. Specifically, Linux and Windows XP Embedded are the operating systems of choice for a PowerChain device. Some known configurations which are supported are:

- Windows XP Embedded
- Linux 2.4/2.6 with glibc 2.2+

Primary development for these platforms will occur in C and C++.

3. Basic Requirements

The basic system requirements for a PowerChain “node” (a generic term used in this document that includes PowerChain “devices”, “gateways”, and “microservers”) are as follows:

- True multi-tasking operating system
- Permanent backing store, e.g. flash memory
- Hardware abstraction for peripherals, e.g. BSD socket calls for networking or character device operations for serial communication
- Basic memory management
- Thread library allowing in-process concurrent execution
4. Portable Features

A minimum set of functionality is guaranteed to be provided by system architects and is guaranteed not to be exceeded by the software developer. These “features” include data type definitions, macros and variable declarations, library calls with corresponding declarations, system calls1 with corresponding declarations, header file names, and conformance with any additional standards or behavior specified. Additionally, some features will typically be present (they are “recommended”), but are not guaranteed. Note that “guaranteed” does not indicate that no PowerChain device will ever be made that lacks these features; rather, it indicates that software developers will assume the presence of these features, thus software will fail to build on systems that lack these features unless modification is performed.

From a developer’s perspective, operations should always be performed using guaranteed features when possible. If this is not possible (i.e. if the guaranteed features are too primitive or would otherwise require excessive work), recommended features provide a standard alternative.

From a system architect’s perspective, guaranteed features should always be provided, and recommended features should be provided whenever possible.

Using the previous illustration as an example, the PowerChain toolkit software will function properly on the illustrated PowerChain node for those components that reside within the green and red sections. Ideally the blue section will be non-existent, since it represents available functionality that cannot be implemented on this node. In reality, it will at least be minimized if developers move as much as possible into the red section.

Also note that some of these features may impose requirements on the build environment or toolchain rather than the platform itself.

---

1 Library calls and system calls are only distinguished here for clarity; whether an implementation chooses to provide certain functions as one or the other is irrelevant to the developer.
4.1. Guaranteed Features

The functionality described below is common to all PowerChain nodes and software. All system architects should provide these features, and all software developers should restrict their coding to use only these features.

4.1.1. C99 Support

Functions, data types, enumerations, macros, global definitions, and behaviors as specified in the C99 standard (ISO 9899) are guaranteed PowerChain features. Examples of such features include:

- File operations (`fopen()`, `fflush()`, `stdout`, etc.)
- Time operations (`time()`, `localtime()`, etc.)
- Math operations (`sin()`, `log()`, etc.)
- Memory operations (`malloc()`, `free()`, etc.)
- String operations (`strcpy()`, `strchr()`, `snprintf()`, etc.)
- Character operations (`toupper()`, `isdigit()`, etc.)
- Variadic operations (`va_arg()`, `va_start()`, etc.)

4.1.2. C++ Support

Functions, data types, enumerations, macros, global definitions, and behaviors as specified in the C++ standard (ISO/IEC 14882) are guaranteed PowerChain features. This includes all parts of the standard, including the "Standard Template Library" (STL).

4.1.3. BSD-compatible Sockets

Functions, data types, enumerations, macros, global definitions, and behaviors related to the traditional BSD socket layer, as specified in the POSIX standard (IEEE 1003.1g), are guaranteed PowerChain features. One notable exception exists for the purpose of supporting Windows-based platforms: Incompatibilities between the POSIX standard and the WinSock2 library should be accounted for by developers. In practice these differences are relatively minimal. BSD functions such as `bind()`, `accept()`, etc. are to be used whenever possible, with WinSock-specific functions used only when necessary (i.e. when getting error codes). Any such deviations should be wrapped in `#ifdef` tags to limit their use to Win32.

4.1.4. Descriptor-based Character I/O

Operations on a generic character device descriptor are guaranteed PowerChain features. This requirement is fairly flexible in the details of how the operations are defined and performed, but they must conform to the descriptor-driven character device paradigm. The preferable interface is the POSIX `open()` / `read()` set of calls; other types of calls can often be aliased to these POSIX calls to work with no additional modification (e.g. the Visual C++ RTL wraps Win32's `ReadFile()` interface with `_open()`).

The intent of these character device operations is to drive communication on external ports, such as an RS-232 port. The provided implementation must therefore allow control of external ports via a character device mechanism, so that device handlers need only be written once to work with all PowerChain devices.

4.1.5. Volatile and Nonvolatile Filesystem Access

Access to local files provided by a standard filesystem interface is a guaranteed PowerChain feature. Such access should be able to store files in backed, non-volatile storage as well as volatile (in-memory) storage. These storage types will likely, but not necessarily, be provided as two separate filesystems on two separate devices. The filesystem used for non-volatile storage should be suitable for reliable storage on the medium used; i.e. a removable device (such as a flash card) should use a journaling filesystem, or some other filesystem that is not easily corrupted by unexpected device removal or power-down.
4.2. Recommended Features

The functionality described below is optional for all PowerChain nodes, and only used as necessary for all PowerChain software. System architects should provide these features when possible, and software developers should use these features when no reasonable “guaranteed” alternative exists. In order to preserve portability and speed development, however, it is highly recommended that these features be taken advantage of (as toolkit requirements become clearer, some of these features may be moved to “guaranteed” status).

4.2.1. Descriptor-based Block I/O

Operations on a generic block device descriptor are recommended PowerChain features. This is similar to the descriptor-based I/O feature, and most systems that implement descriptor control of character devices will likely do the same for block devices, but the intent of this feature is different. While character device access is necessary to control external communication ports, block device access is desirable in order to perform raw I/O on local storage. A specific example of an operation that needs block device access is a firmware update: raw device access is necessary to directly flash new firmware. As with character device access, the preferred mechanism is the POSIX family of device I/O calls such as open() and read().

4.2.2. POSIX-like Threading Support

Functions, data types, enumerations, macros, global definitions, and behaviors related to the portable threading specification (pthreads), and as detailed in the POSIX standard (IEEE 1003.1g), are recommended PowerChain features. Exceptions are made for implementations whose performance can be easily made to mimic pthreads. At least the following behaviors must closely conform to the POSIX specification:

- Thread creation and destruction
- Thread detachability
- Basic mutex operations (lock, unlock, trylock)
- Condition variable operations (wait, signal, broadcast)

4.2.3. POSIX/SysV IPC Support

Functions, data types, enumerations, macros, global definitions, and behaviors related to inter-process communication (IPC), and as specified in the POSIX specification (IEEE 1003.1g), are recommended PowerChain features. Exceptions are made for implementations whose performance can be easily made to mimic POSIX IPC, such as System V IPC. The forms of IPC included in this features are specifically:

- Shared memory operations
- Message queue operations
- Semaphore set operations

5. Software Practices

The previous sections have covered platform requirements, which is of interest both to toolkit developers and product engineers. Another issue relevant to both of these parties is consistency in following software “best practices”. This obviously applies to developers when designing the toolkit, since all in-house code should follow these practices. It also applies to engineers working on various PowerChain products, since they will likely be writing code of their own in order to integrate the toolkit with their specific lower-level protocols. The following items are not intended to be at all comprehensive; rather, they are a few examples of particular practices that are often overlooked during design and development, and which may negatively impact PowerChain products if they are not followed.

Along with the increased functionality of internet-enabled devices comes the additional responsibility of good security. Many security issues are simple to fix once they are recognized, but such fixes can be costly when applied after the fact to a product that was designed without
security in mind. As such, several of these practices are outlined here to ensure that PowerChain products are secure by design, rather than as an afterthought. These practices apply to C code unless stated otherwise.

5.1. Bounded String Functions

Buffer overflows typically occur when some assumption is made (knowingly or unknowingly) about the format of user-supplied input. This may not occur when the input is directly processed, but at some later stage. Therefore, it is appropriate to use bounded string functions at all times possible, since seemingly innocuous program-internal code may provide an unseen path for attack. For example, consider an internal function `blah()`, which prefixes a message and writes it to the system log. The following implementation appears innocent at a glance:

```c
void blah(char *msg)
{
    char msgbuf[256];

    sprintf(msgbuf, "blah: %s\n", msg);
    log(msgbuf);
}
```

This function does exactly what it is supposed to do. Now suppose a programmer wants to use this function to do something extremely simple: log the name of a program. He would call `blah()` from `main()`, which looks like this:

```c
int main(int argc, char *argv[])
{
    blah(argv[0]);
    return 0;
}
```

Note that this function also does exactly what it’s supposed to do. But there is now a trivially exploitable hole in the program. For example, on a UNIX/Linux system an attacker might do the following:

```bash
$ ln –s /sbin/logger ./[really long string]
$ ./[really long string]
```

If `[really long string]` is 250 bytes or more, it will overflow `msgbuf` in `blah()`, and by choosing the length and content of the string carefully an attacker can use `/sbin/logger` to place arbitrary code onto the stack. In this example such a hole may not be a big deal, but consider what would happen if the logger program were setuid with root permissions, or its output was fed as input to some other process executing as root, etc.

The point of the example above is that even code which is expected to be “private” should enforce bounds checking. Thus in this example, `sprintf(msgbuf, "blah: %s\n", msg);` should have been `snprintf(msgbuf, sizeof(msgbuf), "blah: %s\n", msg);` instead. Note, however, that merely using the “n” version of string functions is not sufficient to ensure correctness alone. For instance, if the post-formatted string is exactly 256 bytes long, this check would pass, but the string would have no trailing nul character, which could be utilized by an attacker (to cause `strlen(msgbuf)` to return a number greater than expected, for example). But using bounded functions is at least a necessary start toward security, and therefore this practice should be followed by all PowerChain-related software.
5.2. Multi-language Support

Since PowerChain software may be ported to devices that use languages other than English, all user-visible text and messages should be set aside as a separate resource. The goal is to allow a third party (someone not familiar with the details of the code) to easily build a version of the software that uses an alternate language. Note that this need not apply to debugging messages, service interfaces, or other areas not visible to an end-user.