INL/EXT-11-23423

INL High Level Language Microcontroller Implementation Final Report

Jason Larsen
Jerel Culliss

September 2012

The INL is a U.S. Department of Energy National Laboratory
operated by Battelle Energy Alliance

INL/EXT-11-23423

INL High Level Language Microcontroller Implementation Final Report

Jason Larsen
Jerel Culliss

September 2012

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Electricity Delivery and Energy Reliability
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

EXECUTIVE SUMMARY

The overall goal of this project is to improve the security of embedded devices (microcontrollers) used in monitoring and control of energy-sector infrastructure. As these devices are essentially scaled-down personal computers (PCs), they are vulnerable to common low-level cyber-attacks such as *buffer overflows* and *insecure authentication*. The implementation of a modern high-level programming language on these microcontrollers would disallow such low-level attacks as the languages would have inherent safeguards against these types of cyber exploits.

Idaho National Laboratory (INL) has partnered with Siemens Corporate Research (SCR) to develop and test a 4th -generation language DNP3 implementation on real-world embedded hardware. These tests were carried out on control-system device models currently deployed in real-world applications. Testing was performed with existing C-based DNP implementation as well as the Python version developed by INL.

CONTENTS

FIGURES

TABLES

ACRONYMS

3GL	Third-Generation Language
4GL	Fourth-Generation Language
CPU	Central Processing Unit
DNP	Distributed Network Protocol
DOE-OE	Department of Energy – Electricity Delivery and Energy Reliability
ICMP	Internet Control Message Protocol
INL	Idaho National Laboratory
IP	Internet Protocol
JIT	Just-In-Time
MB	Megabyte
NSTB	National SCADA Test Bed
OS	Operating System
PC	Personal Computer
PLC	Programmable Logic Controller
RAM	Random Access Memory
RFC	Remote Function Call
RTOS	Real-Time Operating System
SCR	Siemens Corporate Research
SMII	Siemens Station Manager II
SQL	Structured Query Language
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
uIP	MicroIP
XSS	Cross Site Scripting

INL High Level Language Microcontroller Implementation
Final Report

1. PROGRAMMER ERRORS – 4GL VS 3GL

Programming errors in third-generation languages (3GL) such as C and Fortran often lead to memory corruption, which then lead to remote code execution on the device. The exploitation of buffer overflows is a well-known cyber-attack and will not be elaborated on at this point. On the Internet at large, Structured Query Language (SQL) injection and cross-site scripting (XSS) account for more compromises than memory corruption, but the use of web services and web browsers is restricted to a very small number of components in industrial control systems. In field equipment and embedded devices specifically, almost unheard of. fourth-generation languages (4GL) tend to have as many problems with XSS and SQL injection as their third-generation counterparts, but they have only a fraction of the memory-corruption bugs associated with them. Most memory-corruption bugs are the fault of the author of the language (which is usually written in C), rather than the fault of the application developer.

It is possible to extend a language such as C++ to automatically perform all the necessary bounds checking on each operation. Templates, such as Microsoft's SafeInt library, do these checks, and templates of this kind are currently in widespread use. It has been shown that once all the checks are in place, the performance of native code approaches that of managed code after optimization. Put simply, once a 3GL is modified to be "safe," it performs similarly to a 4GL.

Below are two code snippets. Figure 1 is C++ with range checking enabled—a 3GL with safety features. Figure 2 is from Python—a 4GL. In both cases, the code is creating an array and then attempting to access a value outside the bounds of the arra*y*.

```
#include <stdlib.h>
#include <stdio.h>

#define MAX_UINT 0xFFFFFFFF
#define OVERFLOW 0x80000005
#define NOMEM    0x80000006
#define OOB      0x80000007

class safeintarray{
private:
        int* data;
        unsigned int totalsize;

public:
        safeintarray(unsigned int number){
                data=NULL;
                totalsize=0;

                if (MAX_UINT/sizeof(int)<number){
                        /*integer overflow*/
                        throw OVERFLOW;
                }
                data=(int*)malloc(sizeof(int)*number);
                if (!data){
                        throw NOMEM;
                }
                totalsize=number;
        }

        int operator[](unsigned int index){
                if (index>=totalsize){
                        throw OOB;
                }
                return data[index];
        }
};

int main(int argc, char**argv){
        safeintarray s(10);
        int value;

        value=s[40];
        printf("%i\n", value);

        return 0;
}
```

Figure 1. C++ example.

```
#!/usr/bin/python

s=[1,2,3,4,5,6,7,8,9,10]
value=s[40]
print value
```

Figure 2. Python example.

The number of necessary lines of code is obviously greater in the C++ implementation, giving the programmer more opportunity to make a mistake. In the real world, even most programmers who have taken secure-coding classes miss the integer-overflow check in the allocation.

Fourth-generation languages are not immune to exploitable memory-corruption bugs. Figure 3 is a snippet of Perl that triggers a format string overflow, given that the attacker controls the variable *userdata*. This vulnerability has since been corrected in Perl, but it illustrates that moving to a higher-level language is not a complete fix.

```
#!/usr/bin/perl

$userdata="%n";
$result = sprintf($userdata, "hello");
print $result;
```

Figure 3. Perl exploit.

2. TRENDS IN MICROCONTROLLERS

Microcontrollers follow Moore's Law, which is an informal statement that the possible transistor density (which implies processing power), will double every 18 months. This statement has held true for over 30 years and looks to continue until at least 2020. As a result, it is no longer profitable to mass manufacture devices with very small microcontrollers. Ten years ago, everything was coded in assembly on bare metal[1]. Toolkits and stripped-down operating systems written in C have largely replaced this bare-metal approach.

A quick survey of devices in INL test beds shows that most of the older microcontrollers have 1–2 megabytes (MB) of flash on either a chip or an external SDCard. Newer devices have 8–32 MB of flash on either a chip or an external SDCard. Older devices have 4–8 MB of memory, and new devices have 16–1024 MB of main memory. That is a broad range, but it is possible to extrapolate the computation-power increase from the data.

The target size in memory for the 4GL and all of its dependencies is set at 2 MB. This size will allow compatibility with most of the embedded devices currently manufactured. Devices that adopt current-generation hardware should have no problems allocating 2 MB of space. Over the course of the project, the absolute minimum-memory footprint that a stripped down 4GL requires while still being able to run a protocol stack will be investigated. Unauthorized patches to some of the devices in inventory have shown that 900 kB patches can easily be incorporated into older devices without the need to change the overall functionality of the device. This would allow for devices already deployed in the field to be updated with new, secure software. It is therefore reasonable to assume that 2 MB can be cleared in a new revision of an industrial control device.

The PandaBoard (Figure 4) is being used as a demonstration target. It is an OMAP-4 advanced reduced instruction set computer (RISC) machine (ARM) processor running at 1GHz with 1GB of random access memory (RAM) and an external secure digital (SD) card for flash storage. This closely matches the specifications of some of the newer field devices in inventory. The ARM series of processors are the most popular in new deployments, mirroring the cell phone and tablet markets. The PandaBoard is inexpensive and can be purchased for around $180 at the time of this report.

[1] Bare Metal is slang for an embedded device that does not have an underlying operating system.

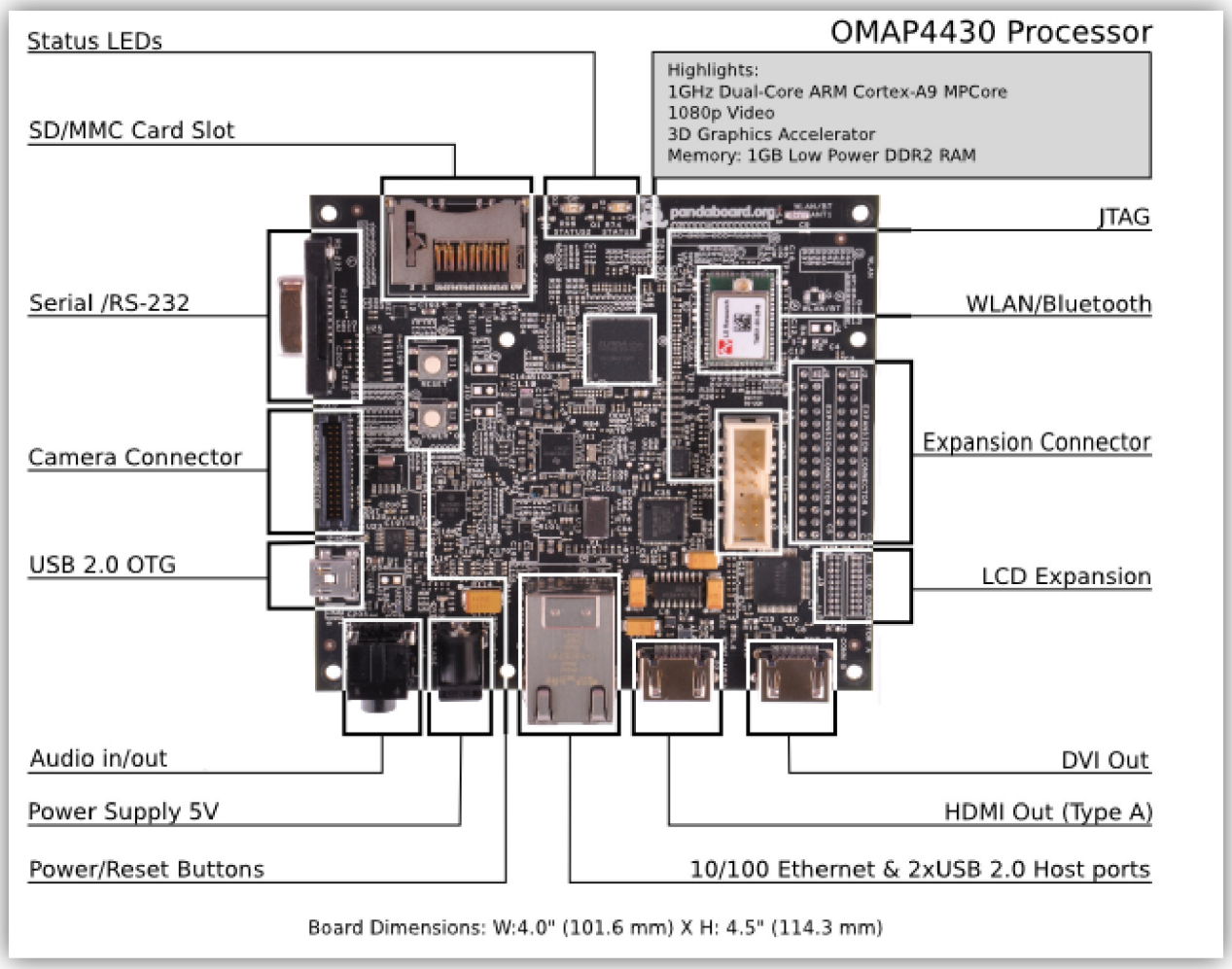

Figure 4. PandaBoard.

3. IDENTIFYING A CANDIDATE LANGUAGE

A number of common languages were investigated. The following criteria were set for our ideal candidate language: (1) the language must be open so that we could modify it and must compile down to a stable bytecode[2] or native assembly; (2) the size of the bytecode interpreter and its dependencies would be small in memory and fast to execute; and (3) the language must be in common use with a living user base and active maintenance.

3.1 Basic Size

A "*Hello World!"* application was written in each potential language to compare the basic size in memory of the interpreter and its dependencies. There was not adequate time to attempt to strip down each implementation and try every compiler option. Each language was compiled on an x86_64 Linux machine running Gentoo, and a best guess was made on which compiler flags would minimize the memory footprint of the application. Table 1 below describes the results.

[2] Bytecode is not human-readable and also not pure machine language.

Table 1. Basic size.

Language	Total Size (KB)
C	3736
GO	7212
Perl	14344
PHP	62400
Python	17988
Rubinius (RBX)	61624
Ruby	18752

The greatest amount of space required for any of the languages was for the libraries it loaded at runtime. For example, Ruby loads *libc, librt, libdl, libcrypt, libm, and libpthread* by default. Collectively, these libraries are eight times the size of the Ruby code itself. Hand-tuning the Ruby-build environment to eliminate these dependencies resulted in a memory footprint of 2.1 MB compared to the original 18.7 MB. Hand-tuning Python had similar results, dropping it from the original 17.9 MB to 1.2 MB. This demonstrates that the minimal size of the interpreter of any of the languages could not truly be determined without a deep investigation of each language.

3.2 Popularity

The popularity of a language varies depending on how it is measured. Some reports measure the number of applications produced in each language, while others measure the number of comments posted on various internet discussion boards. After consideration, it was arbitrarily decided to use the langpop (http://langpop.com) rankings for the various languages. Since the programming language D is still considered to be an experimental language and not used for production code yet, it was chosen as the lower cutoff point for popularity. The final popularity ranking is listed in Table 2.

Table 2. Popularity of various languages.

Language	Percentage
Java	64
PHP	43
JavaScript	38
Python	26
Perl	22
Ruby	17
ActionScript	7
TCL	2

3.3 Stable Bytecode

Most 4GLs compile the source into an intermediate bytecode language before executing it. Some then use just-in-time (JIT) techniques to translate that bytecode into native assembly. Ideally, the firmware for the target device would be written on a development machine, and only the compiled bytecode or native assembly would end up on the device. Any language that made it difficult or impossible to manage a bytecode interpreter was excluded from the running since it would be impossible to strip down the language sufficient to meet the space restrictions on the target devices. A summary of the findings is as follows:

- Java
  - Stable bytecode language
  - Small interpreters available
  - Large user base
- PHP
  - No bytecode language
  - Popular and scalable
- JavaScript
  - Very large user base
  - Many JIT compilers (with bad security)
  - Popular and scalable
- Python
  - Published bytecode that changes periodically
  - Popular
- Perl
  - No published bytecode
  - Popularity on the decline
- Ruby
  - Rubinius (RBX) project compiles to bytecode, but not to mainline Ruby
  - Popularity increasing
  - Very large in memory.

3.4 Python Justification

After consideration of the language performance and bytecode, Python was chosen for a number of reasons. While not the most popular of the 4GLs, it has a reasonably broad following and is an extremely easy language to learn. It has neither the history of incompatibilities that Java does nor the security problems of JavaScript. Also, unlike PHP, Python has bytecode language. Given the size constraints that we felt were necessary in order to implement a 4GL on an embedded device, the performance of Python seemed adequate, given the capabilities of modern microcontrollers. Specifically, Python offered the following benefits:

- Very small base memory footprint
- Ability to dynamically load libraries and keep memory usage low
- Automatic out-of-bounds checking on basic structures
- Very stable language
- Reasonable popularity and ease of learning.

While Python is by no means free of problems, a standard library created in Python should greatly help to reduce programmer error while enhancing security for embedded devices.

4. IMPLEMENTATION OF A PYTHON INTERPRETER

Two approaches were investigated for the implementation of the Python interpreter. The first approach used was a “bottom-up” approach. A Python virtual machine would be implemented from scratch, custom built for size and speed. The second was a “top-down” approach where the mainstream Python interpreter would be stripped down to be as fast and light as possible.

Several previous attempts at making an embeddable Python interpreter from the bottom up have been made.

The most mature of these attempts is PyMite. A test implementation of PyMite on an Atmel AVR microcontroller was built and tested. The PyMite implementation was examined in detail and found to have a number of shortcomings. The first shortcoming is that complex structures would need to be added in order to support recursion and exception handling. The second is that some of the Python opcodes rely on the specifics of the mainline Python interpreter. It would be necessary to implement a significant amount of the mainline Python interpreter into PyMite before it could execute arbitrary bytecode. While this was possible, the implementation could not be considered elegant by coding standards and would require significant maintenance in order to remain compatible with new Python releases, as well as patches or updates of PyMite itself.

In the top-down approach, the mainline Python source was compiled, and then a version of the library *newlib*—a standard library used for embedded systems—was adapted to fulfill its C-language library dependencies. This initially produced a fairly fast image that was 1.4 MB in size. This was within the original project goals, so the top-down approach was adopted.

4.1 Bare Metal Implementation

Not every embedded environment provides the same infrastructure. For example, some provide an underlying network stack, while others do not. Almost every implementation provides a heap, but some do not. In an effort to make the project portable to almost any environment, the initial implementation is being developed as a bare-metal environment. The firmware must perform absolutely all of the work, from initializing the hardware to implementing common C library functions.

It is unlikely that the implementation will ever be deployed to industrial devices on bare metal. Nearly every industrial control device is written on a type of real-time operating system (RTOS) or toolkit that does most of the work. However, providing a full bare-metal implementation gives two major benefits. The first benefit is that the reference implementation provides source code for any dependencies the RTOS does not supply. As an example, if the RTOS does not supply a secure socket layer (SSL) implementation, the developer can use the one from the bare-metal implementation and know it will work with minimal effort. By building a reference implementation without any underlying environment, the reference implementation guarantees that it does not have any dependencies on underlying quirks and that its modules are complete.

The second benefit is the availability of a reference implementation. Not every RTOS approaches a feature in the same way, and some glue code[3] may need to be written to unify the two approaches. The bare-metal implementation provides a module with the expected semantics. For many programmers, an example code implementation is almost always more understandable than a very long specification sheet.

[3] Glue code refers to code which integrates different functions that would not otherwise work together.

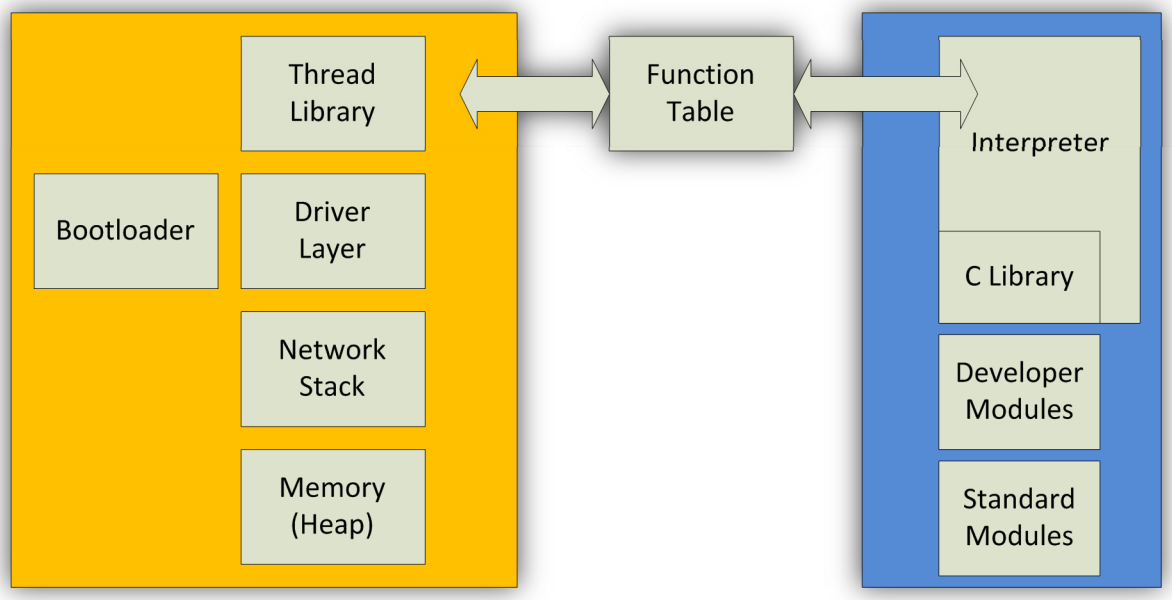

Figure 5. Overall architecture.

4.2 Hosting OS-provided Modules

The hosting environment will need to provide several subsystems or take them from the reference implementation. These modules are shown in the yellow block in Figure 5.

4.2.1 Bootloader

The bootloader is responsible for initializing the hardware, reading the base firmware from permanent storage into RAM, and then executing the base firmware. A custom bootloader was coded for the PandaBoard, but the determination was made that having a custom bootloader did not make the project more understandable or maintainable; therefore, uBoot was adopted for the reference implementation bootloader. The well-known uBoot bootloader is considered to be the most popular and widely supported bootloader, and was determined to be reliable and low-maintenance.

4.2.2 Thread Library

Most RTOS environments provide a thread library. The reference implementation provides a thread library for x86, x86_64, ARM, and Thumb. This covers the two most popular central processing units (CPUs) in their two most popular modes. A brief, informal survey of the devices in the INL's SCADA test bed showed that the two CPU architectures, x86 and ARM, accounted for nearly all of the embedded devices, with only a few running on an MSP430, PIC, or some other CPU.

The reference thread library is rudimentary and simply does round-robin scheduling based on a timer interrupt. The Python implementation is compiled without thread support and is intended to run as a single thread. The transmission control protocol/internet protocol (TCP/IP) stack and drivers need periodic access to the CPU to keep the network stack and hardware running. A simple round-robin thread model gives them predictable access to the CPU while minimizing complexity as well as being very understandable.

The thread library provides both preemptive and cooperative multithreading so that any module can simply yield to the processor if it has nothing to do.

4.2.3 Driver Layer

The driver layer is a very simple subsystem where code can be added to interact with specific or unique hardware. A reference Ethernet module and a reference serial module are provided. The Ethernet module feeds packets to the TCP/IP stack while the serial module exposes a serial port to the main Python implementation. There, it can be used for a variety of protocols.

4.2.4 Network Stack

The network stack is a custom TCP/IP implementation that supports TCP, user datagram protocol (UDP), and Internet control message protocol (ICMP) messages. It is neither remote-function-call (RFC) compliant nor as small as some other implementations, such as microIP (uIP). Its purpose is to be easy to understand, rather than compliant, small, or fast. In most cases, the RTOS will supply a TCP/IP stack.

4.2.5 Heap

The heap is a hardened variant of Doug Lea's malloc (dlmalloc). Dlmalloc is fast, small, and flexible, but a vulnerability existed that could be used to perform double-overwrites during a heap overflow. This implementation has been hardened with unlink checking and heap cookies, but does not use allocation randomization. To properly implement allocation randomization, a strong pseudo-random routine would be required as well as a more complex de-allocation routine. In the future, a more hardened heap may be a compile-time option.

Python has a built-in heap based on dlmalloc that is not hardened. The reference implementation does not enable this feature in Python.

4.2.6 Secure Socket Library Implementation

Many RTOSs do not supply an SSL library, so this is the most likely module to be used in a real-world deployment. The SSL implementation is based on PolarSSL (http://polarssl.org). The calling semantics of the various SSL libraries varies widely and is difficult to abstract. This interface is likely to change in the near future.

4.3 Embedded Modules

The Python implementation is deployable as a single binary blob[4] and does not require recompiling by the end developer. This is done for ease-of-use in an effort to save development time for the end user. A thread is created with the first byte of the blob as the entry point. The entry point takes two arguments. The first is a function table that links to the hosting operating-system (OS) modules, and the second is a pointer to the compiled Python code that implements the developer's code. These modules are separate from the hosting OS modules and are shown outside the yellow block in Figure 5.

4.3.1 Function Table

The function table provides the interface between the RTOS and the Python implementation. There are currently 32 functions exposed, but only four functions need to be implemented for the Python interpreter to run. The number and definition of the function table is still changing and has not yet stabilized.

4.3.2 Interpreter

The interpreter is the Python interpreter. It is a stripped down version of Python with the compiler, optimization and other routines removed. The environment is designed so that it will be easy to substitute

[4] Binary blob refers to a single file which is already compiled and can be used by the end developer. The term "blob" refers to the fact that the file appears to be random garbage when viewed by a human.

other languages in the future. For example, it should be a small effort to replace Python with Ruby if such a need arises.

4.3.3 Interpreter C Library

The interpreter calls into a number of C-library routines. This is done so that the developer does not have to deal with all of the implementation details of the C library. A stripped down version of newlib is used to simplify the function interface. When Python calls *printf()*, *fprintf(), puts()* or any of the other routines that normally write text to the screen, those routines eventually call the *write* system call, which then calls the function table. The interpreter C library hides all of those details from the implementer at the cost of around 30 kB of flash.

4.3.4 Embedded Standard Library

The Python standard library is large and mostly useless for embedded development. It is unlikely that an embedded industrial control device will require an embedded web browser or MP3 parser. The Python standard library has been removed and replaced with one that is oriented towards protocol parsing.

4.3.5 Developer Modules

The developer modules are passed as an argument to the Python thread. The developer can write and test his code on a normal PC and then deploy that code without needing to recompile the Python environment on the embedded device. This will likely allow for significantly reduced development and testing costs when compared to traditional embedded device development processes. The DNP3 implementation created for this effort is an example of such a module.

5. TESTING PROCEDURE

Siemens had several candidate devices to choose from, ranging from programmable logic controllers (PLCs) to control-system firewalls. The top concern when selecting the device was ensuring that the device was widely deployed in real-world operational conditions. Most of the candidate devices, although quite powerful, were under development or were very modern, supporting the team's assertion that newer embedded devices will greatly expand on the capabilities of existing hardware. The selected device, the Station Manager II (SMII), is among the most widely deployed and is not built on modern (less than 5 years since design) hardware.

From Siemens:

> *SICAM StationManager II combines extensive IED integration capabilities, an IEC 1131 compliant programming interface, some of the best configuration and diagnostic tools in the industry, and various advanced features like WAN support, data encryption and HTML server capabilities, to make it a perfect Substation Data Concentrator.*

5.1 Device Characteristics

The device itself is based on the PC104 standard. The standard specifies an Intel 486-compatible processor on a standards-based board. Daughter modules are stacked through the use of a standard connector. The hardware deployed in the SMII has evolved over its lifespan. The originally introduced version was a 486-compatible 100Mhz-class machine, and the latest versions are 486-compatible 800Mhz-class machines. Both the original and current versions were used in testing to examine best-case and worse-case scenarios.

The operating system is Phar Lap. This is a cooperatively multi-threaded operating system with no separation between kernel space and user space. The 486 processor supports a full user space implementation, but Phar Lap does not use this capability. Threads must utilize a system call to yield to

the next thread, and each thread has its own watchdog timer. If a particular thread uses too much CPU or does not yield quickly enough, the device resets itself. (This is typical of industrial control devices.)

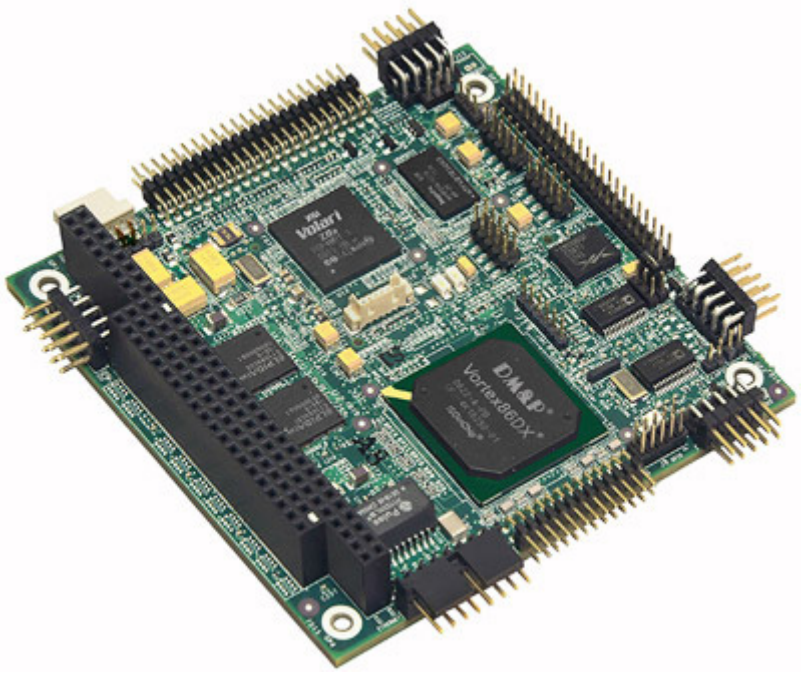

Figure 6. A typical PC-104 board.

5.2 Test Setup

Three configurations were used during testing. In the first configuration, the SMII (slower processor), a monitoring machine, and a DNP tester were connected to a network switch. The second configuration utilized the updated SMII with a faster processor, but was otherwise identical. Under the third configuration, the DNP tester was connected to an SSL server that was then connected to the SMII. The SSL server encrypted the DNP messages from the DNP test equipment and forwarded them to the SMII.

Round-trip times were measured by the test equipment and also measured by network-monitoring equipment. In some cases, the two times varied by as much as 8ms. All reported times, unless otherwise noted, are the times noted by the test equipment.

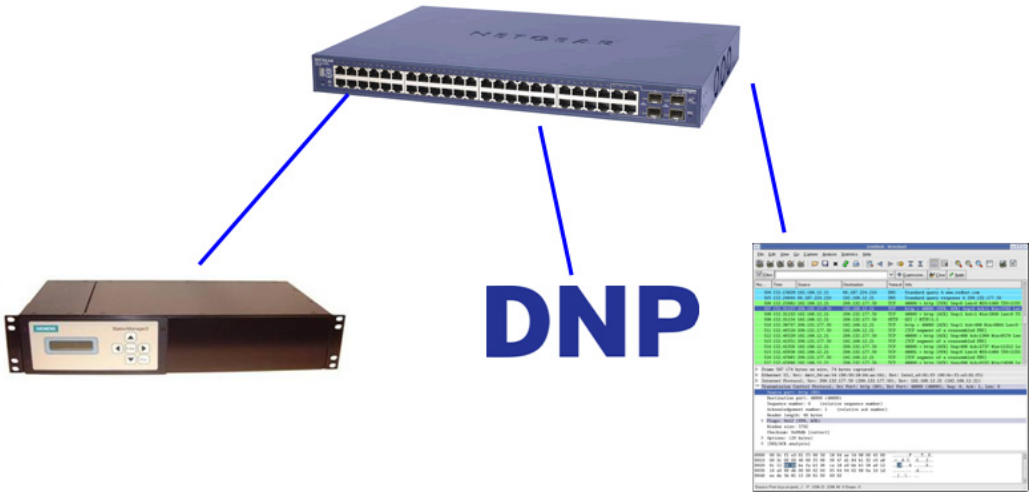

Figure 7. Test configurations I and II.

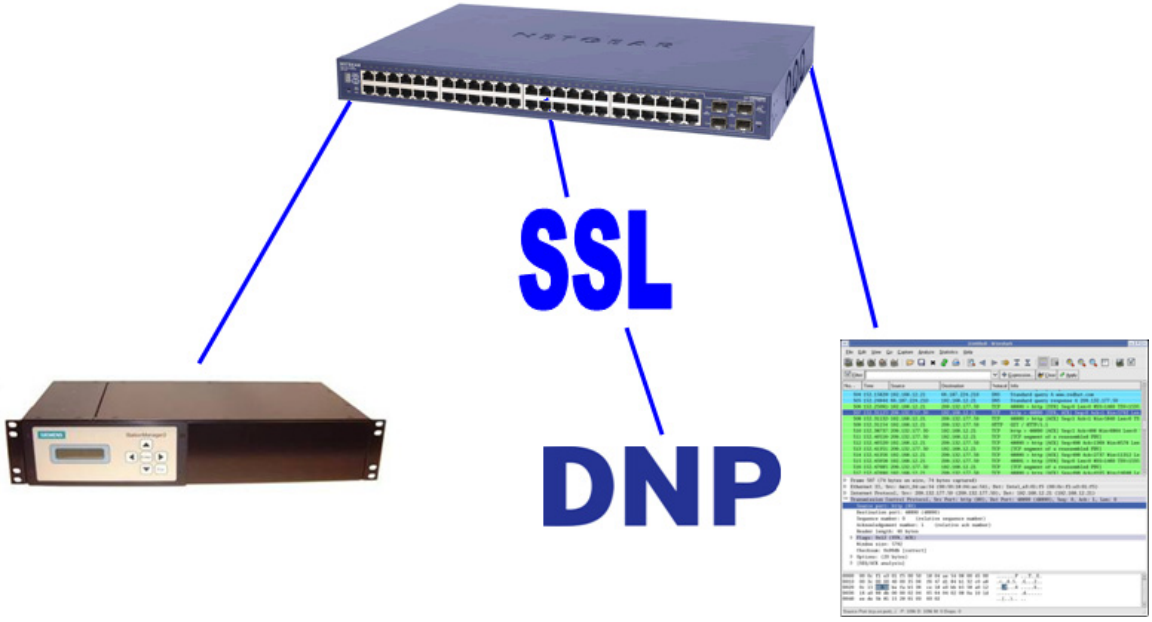

Figure 8. Test configuration III.

5.3 Speed Tests

The first test performed was round-trip time. The DNP test equipment generated a DNP Reset message. This is the simplest message and generates an immediate response. Under this test, the message leaves the test equipment and is received by the TCP stack of the Phar Lap operating system. It is then

passed to the Python environment. The Python environment parses the messages, generates the reply, and then calls into the crypto library to calculate the checksum. The message is passed to Phar Lap TCP stack and across the network to the DNP test gear. The round-trip time encompasses all these operations.

Reset Python	**Reset Native**	
7	7	
8	7	
7	6	
10	7	
7	6	
12	7	
5	7	
16	6	
8	7	
7	6	
8.7	6.6	Average
3.198958164	0.516397779	STD Dev

The table above shows the round-trip time for the Reset message in both the Python implementation and the unmodified device's implementation. These tests were performed using the faster CPU. Doing the work in Python added about roughly 2ms of delay.

The second speed test used the same setup, but instead generated a Read Class 0 and a Direct Operate message. These messages are more complex to parse and must call into native code to interact with the raw data structures on the device.

Read Class 0 *Python*	**DirectOperate *Python***	**Read Class 0 *C***	**DirectOperate *C***
22	23	9	8
28	24	8	8
23	22	8	9
26	22	14	5
26	24	7	9
26	22	7	8
27	22	8	8
25	23	13	9
27	22	8	9
25	23	8	8
30			
26			
25.91666667	22.7	9	8.1
2.108783938	0.823272602	2.449489743	1.197219

The test results show that transitions from Python to C are expensive in terms of latency. Most of this latency is in the thread-locking mechanisms. The size and the complexity of the message is small

compared to the overhead of sending a message from the Python environment and then picking up the reply in another thread. This overall penalty is around 16ms, which is expensive, but still well within acceptable range for a normal DNP-utilizing device.

5.4 SSL Tests

While the Python-based DNP code supports DNP/s operation, neither SCR nor INL had a DNP/s device available for use in testing. In order to test the performance penalty of encrypting messages in the Python implementation, a non-standard DNP over SSL setup was used.

A server was built that listened on the DNP port. It accepted messages from the test equipment and then forwarded them over an SSL link it had established with the SMII. All the SSL handshaking and decrypting was done on the SMII. All the encrypting was done on the server. Round-trip times were measured by the test equipment and include the extra network hop as well as the encryption/decryption.

Reset SSL	Read Class 0	DirectOperate
16	30	29
14	37	26
19	27	25
16	30	27
16	32	29
16	30	26
15	32	26
14	30	25
15	30	27
15	31	26
16	30	28
15.6363636	30.8181818	26.72727273
1.36181697	2.44205577	1.420627262

The addition of SSL encryption as well as the additional network hop added roughly 6 ms to the round-trip-time compared to the Python implementation alone.

5.5 Memory Tests

Determination of total memory usage is often inexact. In theory, the total memory usage should be the total of the Python interpreter, the DNP code written in Python, the maximum amount of stack memory used, and the maximum amount of heap memory used. This number will vary somewhat depending on what processes are running, but the numbers presented here are considered to be a good baseline.

Memory Usage (kb)	Python	C	SSL
Total Image	943	0	980
Total Heap Busy	682	349	796
Total Heap Idle	670	349	714
Stack	40	8	40
Python Image	1618	0	1604
DNP Image	19	0	23
Total	2347	357	2381

The Python image sizes were taken directly from the size of the files loaded into memory and should be accurate. The heap memory usage was taken by instrumenting the calls to malloc/free/realloc. Instrumenting the stack in a meaningful way was not possible; the numbers instead reflect the total amount of memory reserved for the threads running in each configuration. The resulting analysis should be close enough for planning purposes; replacing a DNP stack in C with one written in Python adds roughly 2MB of memory usage to the device.

6. CONCLUSION

This project has shown that a 4GL, Python in this case, can be used successfully in the embedded space on existing devices. The test data show that under normal (unencrypted) operation and SSL encryption, the data were correctly transferred between test devices within an acceptable timeframe for the embedded device.

While the tests involving the Station Manager II did show the 4GL (Python) implementations to be slower than existing methods, the speed difference is actually quite minor given the workload of the devices in question. It should be noted that one of the project's goals was to develop a framework for future embedded devices and that the levels of operation achieved on existing hardware show that this framework can also be applied to existing devices.

Suggested future efforts would be the establishment of a more robust development cycle for Python-based applications and general guidelines for creating embedded applications under a 4GL.