Flits: Pervasive Computing for Processor and Memory Constrained Systems

William Majurski, Alden Dima, Mary Laamanen
National Institute of Standards and Technology, Gaithersburg MD, USA
{william.majurski,alden.dima,mary.laamanen}@nist.gov

Abstract

Many pervasive computing software technologies are targeted for 32-bit desktop platforms. However, there are innumerable 8, 16, and 32-bit microcontroller and microprocessor-based embedded systems that do not support the resource requirements of these technologies. We describe ongoing research that explores the feasibility of creating a portable runtime-environment capable of executing across a wide-variety of 8, 16, and 32-bit processors and that offers mobile code and high-level language support. Our solution adapts a version of the language Forth called FCode and its environment to fit these needs. We introduce the concept of a "flit", a software construct similar to an applet but tailored to the needs of lower resource solutions to pervasive computing. We also describe work currently in progress.

1. Introduction

Pervasive computing is the result of the convergence of three areas of traditional computing: personal computing, embedded systems and computer networking. It can be distinguished from computing in general by its emphasis on ubiquity, interconnectedness and dynamism. Pervasive computing is intended to be ubiquitous; the goal is to create low-cost, embedded, distributed and non-intrusive computing technology. Networking via both traditional wired and newer wireless technologies plays a central role. Its dynamic nature is a result of mobile and adaptive applications that are able to automatically discover and use remote services. Since the notion of information appliances (small, specially-designed computing devices) plays a central role in pervasive computing, the rapid rise of personal digital appliances (PDAs) is but one example of the emergence of this new computing paradigm.

The overall goal of our pervasive computing project, which we call Aroma [1], is to explore key technical, standardization and measurement issues in pervasive computing and to work with the community to begin resolving these issues. A key tenet is our belief that within five years, low-cost systems on a chip (SOC) will be available that include a pico-cellular wireless transceiver and a sufficiently rich run-time environment to be capable of running sophisticated virtual machines. This belief has led us to focus our research in the following areas:

- Investigating the connection of portable wireless devices to traditional networks;
- Researching service discovery, self-configuration and dynamic resource sharing;
- Exploring the use of mobile code and data in pervasive computing; and
- Developing a software infrastructure to create and manage pervasive services and applications.

The focus of this paper is to introduce a software platform for pervasive computing which has small resource requirements, is based on mature standards, and can be leveraged to

bring devices based on small embedded processors into the mainstream of pervasive computing.

While developing an adapter capable of emulating future SOCs and of providing a platform to examine pervasive computing challenges [1], we began to notice that a curious dichotomy exists in pervasive computing. Many observers point to the increasing use of embedded microprocessors in commercial products as evidence of the emergence of pervasive computing. However, many of these embedded processors are lower-end 8- and 16-bit microprocessors and microcontrollers. At the same time, key pervasive computing technologies such as Sun Microsystem's Java™ [2] and Jini™ [3][4][5] appear to implicitly require high-end 32-bit platforms similar to modern personal computers.

Even when low-cost 32-bit SOCs become widely available, there are a host of applications for which they will be too expensive. The "Internet toaster" is a commonly used metaphor in the pervasive computing community, although the same argument applies for other consumer products. Most toasters are in the $10-$50 price range. The consumer products industry is extremely competitive – expensive 32-bit processors will not be used if they do not sufficiently enhance the revenue generated from a product. The manufacturer will be content to use lower cost 8- or 16- bit processors and to develop non-portable, embedded applications in assembly language. This in turn makes it difficult to realize some of the key software aspects of pervasive computing such as mobile code and data.

These issues are not new to pervasive computing. Java™ was originally developed to implement embedded software in consumer electronics [6]. Oak, as it was first called, ran on a stripped-down SPARC processor that was intended for embedded systems. However, the 32-bit SPARC processor proved too expensive for its intended audience. Java ultimately became successful in the desktop personal and server-based distributed computers.

Current thinking about low-end processors in pervasive computing centers on either using stripped-down runtime environments or using high-end processors as a bridge to more sophisticated technologies. If pervasive computing is to become truly pervasive, then low-end processors must be able to fully participate in pervasive applications. Stripped-down runtime environments will become a maintenance issue when they require updates to track development of the "standard" runtime. Systems will become heterogeneous as the older standard runtimes are kept in service to support an installed based of embedded devices while newer versions are deployed. Introducing proxy processors creates a two-tier architecture which increases the cost, the complexity and the potential failure modes. It should be avoided, if possible. A homogenous pervasive computing architecture would also simplify the creation of tools for measuring, testing and debugging pervasive applications.

There are many efforts in pervasive computing, but there is only enough space to mention a few. Service discovery technologies allow networked clients to dynamically discover services on a network. Sun Microsystem's Java™/Jini™ technologies [3][4][5] and the Service Location Protocol [7], a product of the SRVLOC Working Group of the IETF [8], both define this capability. Many pervasive computing technologies are intrinsically targeted for 32-bit architectures. A subset of the Java platform has implemented for the 16-bit processors found on the higher-end smart cards [2].

2. Background

We began thinking about how mobile code could be implemented in resource-constrained environments. The underlying technology should, in principle, be able to run efficiently on an 8-bit microcontroller as well as the latest 32-

bit embedded microprocessor. It should be network-ready, standards-based, compatible with existing pervasive technologies and able to support modern language facilities without "reinventing the wheel". After surveying existing language and runtime technologies, we've concluded that only Forth-based solutions (such as Open Firmware [9]) meet the requirements.

2.1. Forth

Originally created by Charles Moore to control telescope movements, Forth [10] enjoys great success in the embedded/hardware control community [11]. One testimony to its success is the list of current NASA space projects that employ Forth [12]. Forth is both an extensible language and an interactive program development methodology [13]. While it is well suited for small microcontrollers and microcomputers used in embedded systems, Forth has been implemented on nearly every available processor. It has been used in a wide variety of applications, including spreadsheets, expert systems, multi-user databases, and distributed real-time control systems.

Forth is a directly executable language for an abstract stack-based machine. The Forth abstract machine has a program counter, memory, arithmetic logic unit, a data evaluation stack, and a subroutine return address stack. Data evaluation in Forth is accomplished on the Data Stack using Reverse Polish Notation (RPN), also called postfix notation. For example, the following sequence typed from the keyboard:

```
3 4 + 5 * . 35 ok
```

interactively pushes the value 3 on the stack, pushes the value 4 on top of the 3, destructively adds 3 and 4 to get 7, then multiplies by 5. The `.` operation displays the single resultant top value on the stack, 35 (computer output is underlined). `ok` is the Forth command prompt. Operations such as `SWAP` and `DUP` (duplicate) reorder and replicate the top few Data Stack elements.

Objections to Forth's peculiar syntax can be addressed by using compilers and translators for high-level languages that produce a Forth-related byte-code (FCode) as output. This would give the Forth interpreter a similar role to the Java Virtual Machine, which is also a stack-based machine.

Forth is appropriate for pervasive computing for many reasons. It is standardized and available from many vendors and for many environments. Forth is an exceptionally portable computer language which can run on hardware ranging from small 8-bit microcontrollers to modern 64-bit processors. Its small footprint makes it appropriate for desktop applications as well as embedded devices. It has a significant following in embedded systems including mission-critical applications such as NASA flight systems. Forth offers a good trade-off between size and speed. Forth is typically implemented as a threaded, interpreted language, but true compilers also exist. Forth's threaded interpretative nature allows programs to be up to 30% smaller than equivalent assembler programs [14]. This thrifty use of available resources is very useful in resource-constrained pervasive computing applications. Network support is also available for Forth, allowing it to serve as the basis for distributed applications.

2.2. Open Firmware

Open Firmware is an IEEE standard (recognized by ANSI) for the definition of firmware [9]. Firmware is read-only-memory (ROM)-based software that controls a computer between the time is turned on and the time the primary operating system takes control of the machine. Open Firmware offers the following features:

- A byte-coded machine-independent Forth-based language called FCode that allows the same device driver to execute on a wide

variety of CPUs. It is based on the ANSI Standard Forth [10].

- Plug-in device drivers written in FCode that are usually located in device ROM. They are loaded at boot-time into the Open Firmware system.
- A modular design that can be tailored to suit a specific system's needs.
- A programmable user interface that allows for the creation, modification, testing, debugging and execution of user programs in ANSI Forth, that shares a common execution environment with FCode.

In essence, Open Firmware is a small platform-neutral operating system that can load and execute programs from a variety of devices such as disks, tapes, and network interfaces.

2.3. FCode

The Open Firmware standard goes beyond Forth to define plug-in drivers that are written in a byte-coded, machine-independent, interpreted language called FCode, which is based on Forth semantics. Since FCode is machine-independent, the same device and driver can be used on machines with different CPU instruction sets. Each Open Firmware system ROM contains an FCode interpreter.

In interpreted Forth, a "word" (equivalent to a function in other languages) is implemented as a list of addresses for other more basic words. Each element in the list is called an execution token and is sometimes implemented as the address of a small structure defining the word to be invoked. The Forth interpreter traverses the list and invokes each word.

The first fundamental change offered by FCode is in the interpretation of these execution tokens. In an FCode system, the execution token is not an address but instead a single byte value (0-255 decimal) which is an index into a 256 element table. The Open Firmware standard defines the interpretation semantics associated with each element of this table. This follows the basic definition for "byte code". Other differences between FCode and Forth arise in the definition of branching, looping and the encoding of literals. FCode, like Forth, is an extensible language. In the byte-code environment, special codes are reserved as a prefix to longer 16-bit codes. These extended codes allow for developers to define their own words, temporarily assign them to byte codes, and act as extensions to the FCode environment while maintaining binary portability. The result is essentially a binary portability standard for Forth.

3. Flits

Our goal is to apply FCode to pervasive computing, to explore the possibility of a software environment powerful enough for the desktop but scalable enough to fit into the embedded world. Forth meets these requirements well. It defines a standard word as 16 bits or larger and is frequently implemented on 8-bit processors which can double up registers to handle 16-bit data and addresses.

We've introduced the concept of a "flit" ("Forth applet") that will serve as a unit of pervasive and mobile code used to integrate desktop and embedded computing. This is not a new technology but an extension of existing, mature technologies: Forth and FCode.

Flits will define small semantic units that require small implementations. They will be stored locally or transferred across the network as needed. Flit transport is modeled after well-known applet mechanisms. When downloaded from a remote location, they can be cached locally. An integrated version control system will help maintain consistency despite multiple available versions. Access to persistent local storage will be optional.

Flits will use FCode as a portable binary format that can be easily transported across a bus (as in Open Firmware) or a network. FCode applications are portable, processor-architecture neutral, and have low resource requirements.

Flits will execute on 8-, 16-, 32- or 64-bit processors and controllers.

Flits need not originate from Forth source code. They could be generated by language compilers or translators to accommodate programmers skilled with other languages. One potential source environment for flits is a subset of the Java language inspired by the Java Card™ effort [2].

Portable device drivers for embedded devices could be implemented as flits loaded in ROM. The size and complexity of the device would dictate the network protocols used. For example, X10 [15], a proprietary protocol for consumer products that transmits signals over existing 120 volt power lines, could be implemented as a flit.

We do not intend to integrate a graphical user interface. This would result in very large support libraries and applications. Flits are protocol machines by nature. We leverage this strength by stipulating that flits will generate XML streams when a graphical presentation is necessary. We will rely on graphical browsers, such as Netscape Communicator or Microsoft Internet Explorer, to use the XML output to create a graphical interface.

Flits are intended for a wide range of environments. Very lightweight environments appropriate for embedded systems lie at the low-end. The high-end consists of Internet-ready devices that must consider issues such as security and scheduling. We intend to focus first on the lightweight low-end devices.

Our near-term goals include the construction of a lightweight flit environment and associated tools. Specifically, we are focusing on:

- Building an FCode compiler and interpreter;
- Defining the flit architecture;
- Implementing a flit runtime support environment; and
- Integrating dynamic service discovery.

The next section describes the laboratory prototype in greater detail. We realize that concurrency, non-preemptive scheduling, and security are important. We will address these issues in a later phase of the project.

In an expanded environment, flit-related security issues can be addressed relying on certificates from trusted sources and enhanced, as needed by capability-based security mechanisms . The flit runtime environment will allow for the creation of "sandboxes" that prevent access to unauthorized disk and network access similar to those found in Java [16]. When necessary, certain Forth words can be implemented to prevent random access to unauthorized memory locations in a fashion similar to that used by current operating systems. Other mechanisms can be introduced to prevent a flit from redefining core Forth words. When running in user mode under a modern operating systems, an implementation can mitigate the effects of denial of service attacks by using user-level threads – as far as the underlying operating system is concerned, the Forth environment will appear as a single process executing in a single thread. At worst, a malicious flit will crash only the underlying Forth process.

4. Experimental Prototype

We are currently constructing a prototype environment to further develop our ideas. Our laboratory implementation consists of a core platform centered on a Forth interpreter. It is being extended with integrated network support, an FCode implementation, and the ability to boot and run in a stand-alone mode as well as under an operating system. We also plan to explore support for higher-level languages and begin research for pervasive computing software metrology and diagnostics.

4.1. Forth System

We have chosen a Forth interpreter, the Forth Inspired Command Language (FICL) [17], for experimentation. Like Tcl, FICL can be easily extended. It can be embedded into other

systems as a command language. A cell in Forth is the size of an item on the data stack. The ANSI Forth standard specifies 16 bits or larger, FICL uses a 32-bit cell size. We will ensure that our efforts are compatible with a variety of Forth implementations including Forth implementations that use a 16-bit cell.

4.2. Integrated Network Support

Interconnectivity is one of the key features of pervasive computing systems that distinguish it from at the lower-end from traditional embedded systems. We've implemented basic network functionality by creating new Forth words that wrap calls to underlying operating system networking APIs. Many operating systems, including those commonly used for personal computers, have the identical C-level networking APIs. This means that a wide range of platforms can use our extension with just a simple recompilation of the source code. In the past, network stacks have been implemented entirely in Forth. As our research progresses, we may revisit this issue.

4.3. FCode Implementation

Our FCode implementation is being written entirely in ANSI Standard Forth. The implementation uses a minimum of words beyond the core word set which should enable maximum portability. Beyond the core, it relies heavily on the optional String word set, which is unavoidable.

A Forth-based FCode implementation does bring a run-time performance penalty. Network-enabled embedded applications will probably not be effected since the benefits of cost-effective connectivity will outweigh CPU performance. Environments needing additional performance may rely on the many Forth compilers that generate native inline/non-threaded code.

The implementation consists of two components: a compiler and an interpreter. FCode source code is Forth source code compiled against the FCode dictionary. The output of the compiler is a stream of 8-bit values (byte-codes), each of which represent a single Forth word. The interpreter reads the FCode stream one byte at a time. Each byte is an index into a branch table pointing to word implementations. The word implementation is found and executed in sequence.

Most of FCode is implemented by constructing byte-code table entries that point directly to the underlying Forth implementation. A small number of words, such as literals, branching, looping, and new word definitions, must be re-implemented to support FCode.

4.4. Stand-alone operation via FLOSS

In some cases, particularly in deeply embedded systems, it is desirable to have the applications run without an underlying operating system to further reduce the resource requirements and complexity of the software. Forth has traditionally been used to run on "bare silicon" as a lightweight operating system. Because FICL is written in ANSI C and makes use of the C standard library, we have used the University of Utah's OSKit [18] to convert it into a bootable, stand-alone operating system kernel called FLOSS (Flit Operating System Services). OSKit provides a means to quickly link user-level C-source to kernel-level libraries that mimic standard library functionality. It also provides a wealth of Linux [19] and FreeBSD [20] device drivers wrapped in a Component Object Model (COM) glue code layer.

At present, FLOSS only offers the FICL interpreter. The network functions and the FCode interpreter are currently being integrated. FLOSS will allow the execute of flits on low-resource embedded PC platforms while maintaining many of the advantages of operation under a modern operating system.

4.5. High-level language support

Despite the advantages of using Forth-based technologies, its peculiar syntax and semantics will require a good deal of adjustment on the part of programmers already using high-level languages. Fortunately, we believe that language translators and retargeted compilers that generate Forth will allow programmers to continue using their existing languages and tools. For example, the Java Virtual Machine, like Forth, uses a stack-based architecture. We plan to explore the creation of a Java bytecode to Forth translator that will allow the use of existing Java tools to create flits. Java class file disassemblers are widely available. We believe that one can be used as the basis for a Java translator. As a result, our Forth-based solution can be viewed as an extension of existing technologies to low-end processors, rather than a competing technology.

4.6. Metrology and Diagnostics

A major goal of our research is to support the development of pervasive computing metrology and diagnostic techniques. Software measurements are greatly facilitated if the underlying execution environment can provide adequate "hooks" for measurement tools. This is one reason that we find the Java Virtual Machine so appealing. The Java Virtual Machine Debugger Interface (JVMDI) and the Java Virtual Machine Profiler Interface (JVMPI) offer a convenient set of measurement points for metrology tools.

The current state of pervasive computing complicates matters, since the mixture of software architectures makes it difficult to obtain a uniform measurement and diagnostic interface. Our Forth-based solution offers hope because of Forth's portable, open and extensible nature. We believe that it will allow us to easily and efficiently instrument code by redefining the necessary Forth words.

5. Summary

Forth makes a good foundation for pervasive computing. It is a recognized standard that has been successful in embedded applications by providing small solutions to difficult problems. The Forth runtime environment has a small memory footprint and does not required large support libraries for basic operation. It is also portable across a wide range of hardware from 8-bit controllers to 64-bit processors.

Forth has already been extended to applications that have similar requirements to those of pervasive computing. One example is the in the implementation of platform-neutral firmware via Open Firmware specification. Open Firmware achieves binary portability through the use of FCode, a standardized extensible Forth-based bytecode.

Our flits ("Forth applets") will adapt FCode for the network operation by providing an applet-like environment and a container for packaging multiple FCode/Forth components.

6. Conclusion

Current pervasive computing software technologies have addressed 32-bit desktop platforms. However, there are innumerable 8-, 16-, and 32-bit microcontroller and microprocessor-based embedded systems that do not support the resource requirements of these technologies.

We have described ongoing research to explore the feasibility of creating a portable Forth/FCode-based runtime environment capable of executing across a wide-variety of 8-, 16-, and 32-bit processors and offer mobile code and high-level language support. Our runtime environment will work under a variety of operating systems and will also run on bare hardware. It will also facilitate the creation of pervasive computing measurement and diagnostic tools.

We've introduced the notion of a flit, a unit of mobile executable code that can be hosted in processor and memory constrained systems.

The peculiarities of the Forth syntax can be hidden from the programmer by re-targeting high-level language compilers to generate Forth output or by creating translators that convert Java bytecode to Forth.

Note: The presence or absence of a particular trade name product does not imply criticism or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

7. References

[1] "The Aroma Project Web Site", http://www.nist.gov/aroma, National Institute of Standards and Technology, Gaithersburg, Maryland, 2000.

[2] "Java Card™ 2.1 Virtual Machine Specification", Final Revision 1.1, Sun Microsystems Inc., Palo Alto, California, June 7th, 1999.

[3] "Jini™ Connection Technology", http://www.sun.com/jini, Sun Microsystems Inc., Palo Alto, California, 1999.

[4] "www.jini.org Home Page", http://www.jini.org, The Jini Community, Sun Microsystems Inc., Aspen, Colorado, 2000.

[5] Edwards, W.K., "Core Jini", The Sun Microsystems Press Java Series, Prentice Hall, Upper Saddle River, New Jersey, 1999.

[6] Naughton, P., "The Java Handbook", Osborne McGraw-Hill, Berkley, California, 1996.

[7] Guttman, E., "Service Location Protocol Home Page", http://www.srvloc.org/, IETF SVRLOC Working Group, The Internet Engineering Taskforce, April 12th, 1998.

[8] "IETF Home Page", http://www.ietf.org/, The Internet Engineering Task Force, IETF Secretariat, Corporation for National Research Initiatives, Reston, Virginia, 2000.

[9] "IEEE Std 1275-1994: IEEE Standard for Boot Firmware (Initialization Configuration) Firmware: Core Requirements and Practices", Computer Society, Institute Electrical and Electronics Engineers, New York, NY, 1994.

[10] "American National Standard for Information Systems: Programming Languages: Forth", ANSI/X3.215-1994, American National Standards Institute, 1994.

[11] Carter, E.F., "Forth Interest Group Home Page", http://www.forth.org/, Carmel, California.

[12] Rash, J., "Space Related Applications of Forth", http://forth.gsfc.nasa.gov/, National Aeronautics and Space Administration, Goddard Space Flight Center Greenbelt, Maryland.

[13] Koopman, P.J., A Brief Introduction to Forth, Second History of Programming Languages Conference (HOPL-II), Boston MA. 1993, also http://www.cs.cmu.edu/~koopman/forth/hopl.html.

[14] Stiegler, M.D. and Hansen, R.H., Programming Languages: Featuring the IBM PC and Compatibles, Baen Books, New York City, New York, 1984.

[15] Holst, W., "Intelligent Homes", http://web.cs.ualberta.ca/~wade/HyperHome/, Department of Computing Science, University of Alberta, Edmonton, Canada, August 20th, 1998.

[16] Oaks, S., "Java Security", O'Reilly and Associates, Sebastopol, California, 1998.

[17] Sadler, J., "FICL 2.03 Release Notes", http://www.taygeta.com/ficl.html, May 20,1999.

[18] Lepreau, J., "The OSKit - Home Page", http://www.cs.utah.edu/flux/oskit/, The Flux Research Group, University of Utah, Salt Lake City, Utah, December 11th, 1999.

[19] "The Linux Home Page at Linux Online", http://www.linux.org, Linux Online Inc, Laurel, Maryland, January 28th , 2000.

[20] "The FreeBSD Project", http://www.freebsd.org, The FreeBSD Project, Gresham, Oregon, 2000.