2007

Software development of embedded systems on Macintosh

Phillip J. McKerrow

University of Wollongong, phillip@uow.edu.au

Publication Details

This conference paper was originally published as McKerrow, PJ, Lu, Q, Zhou, ZQ and Chen, L, Software development of embedded systems on Macintosh, in Proceedings of the AUC Conference, Gold Coast, 23-26 September 2007, 10.1-9.
Software development of embedded systems on Macintosh

Abstract
The development of embedded systems on Macintosh has been constrained by a lack of software tools. In this paper, we look at a number of tools for developing embedded software on a Macintosh host. Also, we examine the advantage of writing embedded applications in high-level, type-safe languages. In this context, we discuss programming PIC micro-controllers in C, Basic Stamps in PBasic and TINI micro-controller in Java.

Disciplines
Physical Sciences and Mathematics

Publication Details
This conference paper was originally published as McKerrow, PJ, Lu, Q, Zhou, ZQ and Chen, L, Software development of embedded systems on Macintosh, in Proceedings of the AUC Conference, Gold Coast, 23-26 September 2007, 10.1-9.

This conference paper is available at Research Online: http://ro.uow.edu.au/infopapers/700
Software development of embedded systems on Macintosh

Phillip M'Kerrow, Qinghua Lu, and Zhi Quan Zhou and Li Chen
School of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW, 2522. phillip@uow.edu.au

Abstract. The development of embedded systems on Macintosh has been constrained by a lack of software tools. In this paper, we look at a number of tools for developing embedded software on a Macintosh host. Also, we examine the advantage of writing embedded applications in high-level, type-safe languages. In this context, we discuss programming PIC micro-controllers in C, Basic Stamps in PBasic and TINI micro-controller in Java.

Keywords: embedded systems, safe programming languages, Java, PBasic, C, TINI microcontroller.

1 Introduction

When Macintosh sales slumped in the mid 1990s, Apple lost the real-time and embedded systems community. The developers who remained struggled to develop embedded systems, often having to develop or adapt tools for the Macintosh before developing their embedded systems. Many were forced to move off Macintosh onto Windows either by their employer or by lack of tools. Recently there has been a shift of embedded development from Windows to Linux.

With the growth of Mac OSX and the popularity of the iPod, interest in Apple products had increased, as shown by the performance of their stock (Fig. 1.). As more people are using Macintosh computers, there is renewed interest in embedded development on Macintosh computers. The fact that OSX is UNIX based means that open source tools developed for UNIX and Linux are regularly ported to Mac OSX.

![Apple share price chart](image)

Fig. 1. Apple share price in US$, 1985 to 2007 (Yahoo, 2007)

Some embedded systems are personal computers mounted on the machine that they control, such as the outdoor mobile robot Titan (Fig. 2.) Titan is controlled by programs written in LabVIEW running on a Mac Mini. The actuators and sensors are connected to the Mac Mini via USB input/output (i/o) cards.

Most embedded applications are developed on a host computer (Macintosh) and downloaded into a target computer (embedded processor). Usually, they are small applications to control actuators and read sensors in real-time. Many consist of a single processor card, an i/o card, a power source and a connection to a host. Traditionally the connection
to the host is an RS232C serial link, however, more modern systems connect to a network. Their user interface may be as simple as a few buttons and LEDs. They usually don’t have a keyboard and monitor, so a network connection to a remote desktop is desirable.

We have developed a web site to document our systems and to make our experience available to others (M’Kerrow, 2006). In the following sections, we look at the requirements of embedded software and the demands it makes on programming languages. Then we look at some of the systems that are available for development on the Macintosh.

![Image](image.png)

**Fig. 2.** Titan outdoor robot controlled by a Mac Mini. Monitor and keyboard are only used during development. A network connection to a remote desktop is used during operation.

2 Development of embedded systems

Embedded software has much more stringent requirements than personal computer software (Laplante, 1993). It must execute within strict time deadlines, it must be correct, and it must be robust. Your car has an embedded computer controlling its engine. You expect it to calculate the correct fuel/air mixture every time you press your foot on the accelerator. Also, you expect that computer to run for years without crashing or having to perform software upgrades to fix bugs.

The design of real-time systems involves the careful separation of tasks based on timing. For example, our ultrasonic sensors scan the environment every 100msec (M’Kerrow and Antoune, 2007). During each scan an analog-to-digital converter captures 1024 samples of the echo. At the end of the scan, the input program reads these samples from the hardware buffer and places them as a single echo on a queue. This input program is a hard real-time program because its execution is tied to the timing of the sensor. An analysis program reads the echoes from the queue and performs calculations to perceive objects in the environment. It is a soft real-time program because slight delays can be caught up by processing two echoes one after another. The producer/consumer queue provides a way of communication between hard and soft real-time process.

Software for embedded systems is developed on host desktop systems because the target systems don’t have any development tools or human interface. The software required on the host is an integrated development environment (IDE) for code editing, compilation and linking; a converter program to convert the output into a form suitable for downloading and a download program. The embedded computer must have a download program to load the object code into its memory and to start the program executing. Newer systems support download and booting over a network.
2.1 Testing embedded systems

Debugging and rigorous testing of embedded systems remains a difficult problem. A network connection facilitates the development of better tools than a serial link. Using a network, data collected on the embedded system can be analysed and displayed on the host. Also, the embedded system can be controlled from the host.

Performance monitoring and debugging take time to execute and consequently they impact on the timing performance of the processes running on the embedded processor. A network connection enables a hybrid approach where small, fast probes collect data and put it onto a queue. A soft real-time process takes the data from the queue and outputs it to the host over the network. All the calculation and analysis software runs on the host, moving most of the execution load to the host.

3 Languages for embedded systems

The requirements of embedded programs call for language features that are not found in many programming languages. These low-level features are often removed from languages because they are not safe. That is, when incorrectly used, they can crash other programs or the operating system. If the language doesn’t support low-level features then the language either has to be extended or it has to support calls to assembler routines. The latter is very unsafe. Rather than leaving these features out, Modula-2 placed them in a SYSTEM MODULE, so that the programmer would explicitly recognise that the instructions are unsafe and use them with care.

Low-level features include:

- access to specific memory locations, such as the address of a hardware input buffer;
- treating the contents of memory as different types, such as loading in bytes from a serial input and then using them as an array of pixels in an image;
- setting bits in a register, such as when changing the processor from user to system state;
- changing the return-from-interrupt address, such as an interrupt handler returning to a priority-pre-emptive scheduler which dispatches a different process to the one interrupted;
- saving and restoring system state including register contents before and after handling an interrupt;
- programming an interrupt handler, so that it is vectored to by the hardware and not called from software, and
- an interrupt handler being able to transfer data to a user process, such as the interrupt handler reading a value and then storing it in a variable known to the process.

These low-level constructs are machine dependent. The problem with them is that they lack the redundancy required by the compiler to check them for consistency with the rest of the program, and the compiler is not able to protect the programmer against errors. Also, the IDE must be able to add the appropriate header and create links to routines in the run-time support software in the embedded system, including its operating system.

Choosing a good programming language can significantly improve the quality of the embedded software. Reliability is the most important feature for real-time systems. Real-time programming languages should include run-time support to minimize run-time errors and to reduce the probability of the errors caused by programmer.

3.1 Low-level languages

Machine code defines the capabilities of a processor and is directly executed by it. Instructions are represented with binary numbers that have a one-to-one mapping to a hardware function. As people find lots of numbers difficult to remember, they program in assembly language, which uses mnemonics to represent the machine codes. However, assembly language code is difficult to read and it is easy to produce errors. Therefore, although it is common, the use of assembly language is not encouraged in real-time systems (Gritzalis, 1998).

Programming in assembler is tedious and time consuming even though the programmer has total control over the machine. Programmers soon realised that they were programming the same sequences of machine code over and over
again. By replacing these sequences with high-level language constructs they were able to program faster and their programs had less errors as well.

**PIC Microcontroller.** The PIC microcontroller is very popular for building simple embedded systems that consist of a PIC chip and some signal conditioning components. It is also very popular in University and TAFE subjects. A program is developed on a host and loaded (programmed) into the PIC’s flash memory. The PIC chip is physically removed from the programmer and plugged into the application circuit board. It is easy of use and has a very low price, making it a popular microcontroller.

However, only a few programming tools are available for Macintosh. Kevin Coble (2005) has released a beta version of MacroASM. It supports programming in assembler for a number of PIC microcontroller types and hardware programmers. He has used an XML approach to definitions for the microcontroller and the assembler formats, so that it is easy to extend to new PIC chips. Like many products developed as a hobby, further work depends on the interest of the user community.

MacTech published an extensive article on programming PICs on Mac OS X (Djadiningrat, 2004). The host development software consists of two parts: JAL and XWisp. JAL is a high-level language that generates a PIC compatible hex file. XWisp is a Python-based program that uploads the hex file from the Macintosh host via a Wisp628 programmer board (VOTI) to the PIC microcontroller. The article explains how to compile JAL for MacOSX, how to create a connection from the Mac via the Wisp628 programmer board to the target circuit containing the PIC, and how to upload a hex file using XWisp. The process is illustrated with the programming of a simple program to flash a LED.

### 3.2 High-level languages

High-level languages achieve three purposes:

1. They make the code easier for people to read.
2. They protect the program from dangerous constructs that human programmers produce both by accident and deliberately. This protection is achieved by the compiler taking over the function.
3. They increase programmer productivity. Programming is more enjoyable when you can focus on solving the problem and not be bogged down by run-time errors.

**The C programming language.** C achieves the above three goals to a limited extent. Firstly, the code is easier to read than assembler. Secondly, the compiler takes over the control of the register set so that the programmer can no longer select which register to use or explicitly change the content of a register. This protects a program against the programmer using one register for two different purposes. Also, in theory, it stops the programmer writing self-modifying code.

C is the language most commonly used in embedded programming. However, it has a number of serious problems that may result in a system crash, some of which are listed below:

- C code is very difficult to read and understand. It was designed prior to the research into human computer interface and its syntax is very poorly designed. Also, as it was developed before cut-and-paste editors, it has a cryptic syntax, making it easy to type but hard to read. Additionally, it has no concept of graphics, graphical user interfaces.
- Computer Science as a discipline often lacks rigor. C has defined a couple of functions differently to how they are defined in mathematics which causes confusion.
- It has week typing, which results in programs with numeric errors. By allowing statements that assign a float to an integer a program will truncate the value and give the wrong result.
- Pointer arithmetic allows the code to write anywhere and if the arithmetic is wrong the code will write over other code or data. A hardware solution (the memory management unit) was invented to protect against this software problem.
3.3 Safe Languages

The goal of a safe language is for the compiler to handle potentially unsafe operations rather than the programmer. Also, a safe language includes run-time support to catch and handle run-time errors. The features that make a language safe include:

- Minimizing the damage due to programmer error by getting the compiler to handle dangerous functionality. By catching more errors at compile time rather than at run time, it can also increase programmer productivity.
- A safe language is type safe. There is no mixing of types, so there are no errors due to numbers changing in value when assigned from a variable of one type to a variable of another type. Cast operations have to be explicit and justified.
- A safe language has asserts to check conformance to design. Asserts should not be used to debug code, rather they should be used to catch incorrect usage of functions. An assert performs a calculation on input values to confirm that they are in the desired range and type.
- There is no pointer arithmetic in a safe language. All address calculation, such as array indexing, are coded by the compiler. Programming with references rather than with pointer arithmetic stops a program scribbling outside a program’s memory area. As a consequence, it eliminates the need for memory management units as protection devices.
- A safe language includes overflow and underflow checking in its run time support, so that buffer writes can’t corrupt code. A common method of attacking the security of an operating system is to attempt to achieve a buffer overflow or underflow.
- A safe language has real-time garbage collection, i.e. automatic memory management to avoid memory leaks which may cause an operating system to run out of memory and crash.
- A safe language handles mathematical errors, such as divide by zero, that cause low-level hardware faults with exceptions.

4 A flying robot

A tethered robot (M'Kerrow, 2007) is a flying robot that swings on the end of a tether (Fig. 3.). It can be lowered from the top of a building to inspect the side of the building or to peer into windows. It can be lowered down a mine air-supply shaft to look for people trapped behind a rock fall. It can be lowered from a helicopter hovering over a village to reconnoiter between buildings.
The software for the robot consists of two parts: host and embedded. The host software provides a user interface to control and monitor the robot (Fig. 3.). It is written in LabVIEW and runs on a Macintosh host. It communicates with the robot over a serial radio link. The embedded software controls the motion of the robot in response to commands from the host and sends monitoring data to the host. It is written in PBasic and runs on a Parallax Basic Stamp (Parallax).

4.1 Parallax PBasic

The Basic Stamp is a low cost, embedded system for use in hobbies and education (Djajadiningrat, 2004). It runs a version of BASIC called PBasic and includes 24 i/o pins for connection to actuators and sensors. A program is written, compiled and downloaded from a Macintosh host over a USB to serial link using the macbs2 IDE (Konar) written by Murat Konar.

It is a lot of fun to program because it is in a high-level language with extensions to support i/o from serial and digital ports. However, it has several limitations that make real-time programming difficult. First, it only supports one process so everything has to be done in a single loop, which makes it difficult to read commands from the serial link while controlling the speed of a motor. It forces the programmer to think very carefully about the timing constraints of the task.

Second, the serial ports are not buffered, so if the program doesn’t read of an input character in time the next character overwrites it. Third, when a character fails to arrive the serial read instruction hogs the processor and until it times out. Also, the instruction for generating an output pulse on a digital line doesn’t return until the pulse is complete. These issues arise from not supporting multi-processing and make it difficult to program time constrained applications.
4.2 Connecting a Macintosh to an R/C model

A radio-controlled (R-C) model is manually controlled with a set of 4 joysticks mounted on a radio transmitter, with one joystick for each control channel. The position of the joysticks is represented with an analogue voltage. To transmit these voltages to the model, they are encoded as variable-width pulses in a pulse train. The receiver on the model decodes the pulse train back into voltages and sends them to the servos that control the model. Many R-C controllers have a trainer input, where the pulse train can be read from another controller, and transmitted to the model.

To control an R-C model from a Macintosh, we need to generate the pulse train and connect it to the trainer input. Using a Basic Stamp to convert a set of 4 values sent on a RS232C serial line to a pulse train to connect to the trainer input of an R-C transmitter (Fig. 4.) has all the programming issues mentioned in Section 4.1. The pulse train should be generated every 20 msec. If pulse trains start closer than 12 msec or further than 30 msec apart then some servo amplifiers ignore them. The Stamp can’t do anything else during the 5 to 9 msec it takes to transmit pulses for 4 radio channels.

So the serial input has to be read during the synchronisation pulse, which is 4 msec or longer. As the pulse time varies from 1000 to 2000 microseconds, with the mid (zero point) being 1500 microseconds, the command program running on a Macintosh transmits the difference to the mid point, reducing the number of characters from 4 to a maximum of 3. In addition it sends an alphabetic character to identify the start of the sequence and an alphabetic character after each channel value. The latter uses case to signal the sign of the difference.

Thus, the number of characters ranges from 9 to 17. As it takes 1 msec to receive each character at 9600 baud, the serial-read time varies from 9 to 17 msec. Consequently, the time between transmitting R/C pulse trains varies from 18 to 26 msec, which is inside the allowable range. The circuit diagram and software are available on our website (M’Kerrow, 2006).

5 Embedded programming in Java

Java was designed to be a safe language and meets the criteria in Section 3.3. The issues of developing real-time software in Java are discussed in detail in the accompanying paper (M’Kerrow, et. al., 2007). Java is designed to compile a program every time it is run. However, embedded systems are generally compiled once and run many times. This difference in underlying philosophy means that Java compilers normally are not optimised for producing code for embedded systems.
5.1 TINI

TINI (Tiny Internet Interface) is a microcontroller that runs a Java virtual machine (Fig. 5.). The TINI platform is a combination of the broad-based I/O, a full TCP/IP stack, and an extensible Java runtime environment that simplifies development of the network connected equipment. We develop the programs in XCode on a Macintosh and then download them into the TINI using the Slush shell that runs on it. Documentation on how to set up a system to develop applications for TINI on Mac OSX is available on our web site (McKerrow, 2006).

![Maxim TINI Java microcontroller from Dalas Semiconductor](image)

Fig. 5. Maxim TINI Java microcontroller from Dalas Semiconductor

The TINI OS schedules process with a round-robin scheduler every 8 msec and schedules threads every 2 msec. The JVM sits on top of TINI OS. In between there is a native interface layer that supports the execution of assembly code functions to solve low-level problems from Java applications. The i/o library uses the native interface layer to call functions written in assembler to read inputs and write outputs. Applications programs written in Java application sit on top of JVM.

We have been exploring the design of real-time operating systems in Java using the TINI system (McKerrow, et. al., 2007). We are interested in whether we can build an operating system in a safe language that a user process can’t crash. We want to move some of our perception and control software out of the Macintosh host and into Java microcontrollers mounted on our robots.

6 Conclusion

The eminent release of the iPhone, and its potential to run new applications, may result in many more people developing software on Mac OSX, with spin offs for the development of embedded software. The next few years promise to be an exciting time for the development of real-time and embedded systems on Macintosh computers.

While the number of tools for embedded development is small compared to other operating systems, the people who support and use them are very enthusiastic. We have found them to be very willing to help us to get their tools running and to fix bugs that we report to them. The more people who use their tools, the more incentive they have to update them.

Acknowledgements

The authors wish to thank the Apple University Consortium for supporting this research through their “Innovation Grant” scheme.
References


Konar, M. macBS2 ide for Parallax PBASIC. <http://www.muratkonar.com/otherstuff/macbs2/>


Yahoo Finance, Apple share price, May 29 2007. <http://uk.finance.yahoo.com/q/bc?s=AAPL&t=5y&l=on&z=m&q=q1&c=>

McKerrow 10-9