Embedded Java Controllers

Before you start piecing together your next robotics puzzle, consider what Jay has to say about the power of embedded Java controllers. In this article, he introduces you to Systronix Java controllers and the SimmStick bus they use. He concludes with information about the code and hardware that he used to create a simple motor controller for R/C servos.

I’ll admit it. I’m used to programming when I have lots of memory. Although the first computer that I built had 18 KB of RAM, which was large for the day, every year computers gain more memory. I wouldn’t even think of buying a laptop with less than 1 GB of RAM now—and perhaps more by the time this article sees print.

Later, I became interested in embedded systems. The rules changed. I stepped into a time machine, and suddenly I had RAM that could be measured in kilobytes again, and not many of those. Even the stamp-type controllers still had extremely limited memory. Also, most of these microcontrollers were programmed in assembly language, C, and BASIC. Yes, I can handle all of these languages, but I prefer Java and a bit more memory.

Then a small company called Systronix came out with the JStamp. This is an ajile aj-80 Java processor put into a 40-pin-wide DIP package with 512-KB RAM and either 512 KB or 2 MB of flash memory. And its native machine language is Java bytecodes! So, I ordered a JStamp+ (the one with 2-MB flash memory) and started developing. This controller is more powerful than any of the other Stamp processors I’ve found, and it is reasonably priced.

Systronix then came out with the JStik, which uses an aj-100 processor that benchmarks approximately four to five times faster than the aj-80 and has much more RAM and flash memory. The JStik also has built-in Ethernet, two COM ports, a high-speed I/O (HSIO) port, and the JTAG connector for programming.

My goal here is to introduce you to the Systronix Java controllers, the SimmStick bus used by them, and the differences between Java 2 Micro Edition (J2ME) and standard Java. Furthermore, I’ll explain the code and hardware needed to create a simple motor controller for R/C servos. I would also like to say up front that I use embedded controllers for robotics. However, pretty much everything in this article applies to other embedded systems as well.

JStamp/JStik

The Systronix controllers have lots of memory, plus everything else you’d expect from a microcontroller, except for A/D converters. Luckily, this is easily fixed by using one or more ADC chips such as Microchip's MCP 3208, which is an eight-channel, 12-bit ADC that communicates via SPI.

Both of these 3.3-V CMOS controllers can tolerate 5-V signals. This means they can communicate reliably with 5-V TTL and 3.3-V CMOS devices. Unfortunately, the signal definitions for 5-V CMOS are slightly out of range. It is best to use buffers when communicating with 5-V CMOS, unless those devices are friendly with 3.3-V CMOS. However, I’ve never had trouble interfacing PIC microcontrollers to the JStamp and JStik.

The JStamp is built much like any of the other Stamp processors. It has a 1” × 2" 40-pin DIP format. The JStamp’s brain is the aJ-80 microcontroller. The JStamp runs at a maximum of 70 MHz.

The JStik, which is a JSimm format board measuring 3” × 2.65”, has a few advantages over the JStamp. It may be bigger, but it supports two additional ports: an Ethernet port and the HSIO port. The JStik is roughly five times faster than the JStamp when running at the same speed, mainly because it has an aj-100 microcontroller. It runs at a maximum of 100 MHz.

JTAG INTERFACE

I’m used to using serial ports to program a controller, but the Systronix controllers use the JTAG interface. The JStamp has several pins that can be used to connect to the JTAG connector. (There is

---

**Figure 1**—The ajile architecture has all the bells and whistles associated with normal microcontrollers, except an ADC unit.
such a connector on the JStamp development station. The JStik has a JTAG connector onboard.

In order to program and debug the Systronix controllers, you need either the Systronix JTAG cable ($50) or Xilinx Parallel Cable III. I've always used the Systronix one that came with the JStamp development station; it connects to a computer's parallel port.

**SimmStick BUS**

The SimmStick bus, which was designed by an Estonian named Antti Lukats, is based on the old 30-pin SIMM modules. The bus has a decent number of signals and is quite useful. Dontronics is the largest distributor of the boards.

A SimmStick is designed to be an SBC, using a microcontroller as the main processor. Systronix refined the pin definitions for the SimmStick and have released this as the JSimm. The JStamp development station uses this bus, and the JStik is a JSimm board. The major differences are that Systronix defined some of the JSimm signals as JStamp/JStik I/O pins. Originally, I merely used the SimmStick bus to carry power and SPI signals, but I now use other signals.

One interesting thing: with the development tools you can turn off the drivers that you don't need. For example, if you don't need to use SPI, you can turn off the SPI driver and use all of I/O port C as general I/O. Not only can this shrink the final code, it allows you to use the pins you want for anything.

**aJile PROCESSORS**

In most respects, the aJile 80 and 100 processors work the same way. However, the aJ-100 has a wider data path, making it faster.

The processors run Java bytecodes as their native machine language. Because Java bytecodes are a high-level code than most machine languages, more can be done in a single instruction. This tends to make the aJile processors extremely efficient.

Like most controllers, these contain internal data memory. They also have an external memory controller to access the memory on the JStamp/JStik. There are three timers, an interrupt controller, two UARTs, and one SPI communication unit (see Figure 1).

**JStamp/JStik SPECIFICS**

The first Systronix product I bought was the JStamp+. Although it has the same footprint as the JStamp, it has 2-MB flash memory instead of 512-KB flash memory (see Photo 1).

The JStamp has two power-in pins: one is for regulated 3.3 V, and the other is for unregulated power. It also has an on-board switching regulator. Use only one of these pins, not both.

The JStamp has a pinout that fits in extremely well with the SimmStick form factor. The exact details are available on Systronix's web site.

Using the JStamp is easy. You can implement the JStamp development kit, or you can easily build a circuit around it. Because you can put 5 V into the unregulated power pin, the JStamp can be easily put into most common circuits. I'm planning on prototyping a board using the JStamp and a few other components to make a complete system.

The JStamp can take either regulated 3.3 V or an unregulated 5 to 14 V. At the maximum speed (74 MHz), the typical current consumption is 57 mA at 3.3 V, or a bit more at 5 V. If using unregulated power, the JStamp can supply 100 mA of 3.3-V regulated power for your applications.

The JStik is a bit of a change from the JStamp (see Photo 2). Although the latter is a 40-pin-wide DIP, the former is a SimmStick board. Not only does it include a faster processor, but also more memory, an Ethernet port, and an HSIO port. It is possible to use the JStik without a SimmStick backplane, either totally stand-alone or by using an alternate connector. Because the power connectors, the JTAG port, and the serial ports are built into the JStik, you can do a lot with the JStik alone. It is definitely an SBC.

The JStik is programmed just like the JStamp. Note that it has almost the same power requirements as the JStamp, except that at its maximum speed it requires 310 mA when running at 103 MHz. This low current consumption coupled with its power makes the JStik an excellent choice for robotics systems.

**PROGRAMMING TOOLS**

AJile provides a few applications to program and debug your JStamp/JStik. These come with the development versions of the hardware.

Jembuilder takes Java class files and converts them to a form that can be loaded into the controller. This is a type of linker. Charade loads this file and acts as an interactive terminal/debugger.

JSwat is a debugger. I haven't used it, so I can't comment on it. Although I admit to occasionally writing buggy code, I've learned other methods of debugging Java.

In addition, you need a Java compiler. Sun's Java works fine. A text editor to create the source files is useful, although some people prefer an IDE. All of the applications run in Windows.

**EMBEDDED JAVA**

If you're used to running Java on a larger machine, the connected limited device configuration (CLDC) version of Java is limited. Some things are left
out because they aren’t needed [e.g., the entire java.awt.* hierarchy is gone]. However, most of the core libraries remain, although in a somewhat smaller form. Many of the methods and classes not needed in an embedded environment have been removed. For instance, the main hierarchy is reduced to java.io, java.lang, java.util, and javax.microedition.io. Each of these packages is pruned to have only the classes needed for embedded systems. You can download the latest version of the CLDC standard from Sun.

The main thing you need to know is that the JStamp and JStik do not support normal preemptive threading. For all normal threads, you must use Thread.yield(), Thread.sleep(), or some sort of blocking I/O in order to switch threads. This makes for a much more defined real-time behavior when threading is used.

The other main difference is that the I/O packages are much more limited. There are no java.net.* packages. Replacing this is the javax.microedition.io package, which deals with I/O at a somewhat lower level than standard Java. However, the javax.comm package works fine when dealing with the serial ports.

Note that the CLDC is one of the configurations of Java 2 Micro Edition. Ajile has pledged to eventually provide the other configuration, but it isn’t available at this time.

JAVA PACKAGES

Because an ajile processor is a microcontroller, you need access to the low-level parts of the controller. Just as it is necessary to use low-level registers on standard microcontrollers, it is necessary to do the same with the ajile processors. Hence, ajile wrote a number of packages to deal with the controller at the device level.

The com.ajile.components package contains various hardware components, including timers, triggers, and debounced buttons. com.ajile.events includes events specific to the ajile processors. The com.ajile.io package adds BufferedInputStream, BufferedOutputStream, and BufferedReader to the CLDC java.io package.

com.ajile.jem is the lowest level package. By using the rawJEM class, you can deal with memory locations and registers. The package also contains the PeriodicThread and PianoRoll classes, which are used later in this article.

Because the CLDC doesn’t include the Float, Double, and Math classes, they are added in com.ajile.lang, along with some Error classes. com.ajile.drivers irq is the interrupt handler framework. The com.ajile.drivers.flash package contains the methods needed to read and write directly to the flash memory of the JStamp/JStik.

I find the com.ajile.drivers.gpio package to be the most useful. It is used to deal with low-level I/O ports and pins via high-level objects.

The com.ajile.drivers.gptc package is used to access the general-purpose timer/counters and the waveform module. The waveform module is a low-level part of the ajile chip, which can be used to generate arbitrary digital waveforms.

com.ajile.drivers.i2c has a driver to communicate via FC. com.ajile.drivers.rtc.DS1305 provides support for the internal real-time clock chip, while com.ajile.drivers.RX5C62 provides support for the calendar chip.

com.ajile.drivers.spi has a driver to communicate via SPI. The com.ajile.file system package supports the accessing of the file systems available to the ajile chips. (Currently, these are only flash memory file systems.)

com.ajile.bootloader is an interface that allows a program to be loaded into a separate JVM. com.ajile.util contains a class, PropUtils, which allows access to system properties.

com.ajile.microedition.io contains the MemoryEfficientConnector class. This is an implementation of the java.microedition.io connector interface. com.ajile.net.util is a package containing network utility classes. com.ajile.net.dns provides basic access to domain name service. And, finally, note that com.ajile.net.sntp is a package for dealing with the simple network time protocol.

In order to have the advantages of preemptive threading, ajile, which wrote the Java libraries and tools for these controllers, has provided the
PeriodicThread class to allow fully preemptive threading for time-critical items. The PeriodicThread can be a bit tricky to use until you get the hang of it. It uses the PianoRoll feature of the controller in order to run the PeriodicThread at intervals.

Listing 1—This code illustrates the use of the PeriodicThread. I removed the code normally used for stopping this thread safely because it doesn't concern the control of servos.

class ServoThread extends PeriodicThread {
    private static ServoThread servoThread = null;
    private boolean running = true;
    private boolean stopped = false;
    //There are 200 ticks in 20 ms.
    private int maxTicks = 199;
    //A place to store the active servos.
    private GpioServoMotor[] motors = {null, null, null, null};
    private int numMotors = 0;
    //Make sure that this class cannot be created by an outside class.
    private ServoThread() {
    }
    //This is the only way to create an instance of this class.
    public static ServoThread getInstance() {
        if (servoThread == null) {
            servoThread = new ServoThread();
            //The thread is called once every 0.1 ms, with no leading
            //wait, and with a high priority; the thread is then started.
            servoThread.makePeriodic(0, 100000, 0, 0, Thread.MAX_PRIORITY - 2, servoThread);
            servoThread.start();
        }
        return servoThread;
    }
    //Add servos here. At this time, I'm not concerned with removing
    //any servos from my list.
    public void addServo(GpioServoMotor servo) {
        motors[numMotors++] = servo;
    }
    //Just go in a continuous loop until somebody wants this to
    //stop, perhaps for powering down the robot.
    public void run() {
        int tick = 0;
        int i = 0;
        while (running) {
            //PeriodicThread.cycle() is similar to Thread.yield(), but it
            //causes the thread to start again at the specified interval.
            PeriodicThread.cycle();
            if (tick == 0) {
                //At the start of the cycle, turn on all pulses.
                for (i = 0; i < numMotors; i++) {
                    motors[i].getPin().setPinState(true);
                }
            } else {
                //Check for the end of a pulse for each motor.
                for (i = 0; i < numMotors; i++) {
                    if (motors[i].getTicksPerPulse() < tick) {
                        motors[i].getPin().setPinState(false);
                    }
                }
            }
            tick++;
            //At the end of a cycle, go back to zero.
            if (tick > maxTicks) {
                tick = 0;
            }
            stopped = true;
        }
    }
}
MOTOR CONTROLLER PROJECT

Now I’ll describe a simple motor controller that uses the I/O pins of the JStamp/JStik and controls four servo-motors. Because my application (my robot, Zeppo) uses servos converted for continuous rotation, I don’t need a lot of resolution in my motor control.

The full open-source robotics framework is posted at http://enerd.ws/robots/code/. The code for the motor controller is taken directly from this framework. I must warn you that the framework is constantly evolving, so the code on the site may be more recent than the code associated with this article.

An R/C servo motor (because I’m on a first-name basis, I call it just a “servo”) is a standard DC electric motor with a bit of electronics added. The electronics are designed to take pulses generated by an R/C receiver and convert them into a position. Normally, the pulses control the position of the motor to an angle of ±90°. However, I use servos that have been modified for continuous rotation (converted servos).

A standard servo will be at 0° or no rotation with a pulse of 1.5-ms duration. Positive rotation is achieved with a pulse duration of more than 1.5 ms. Negative rotation is achieved with a pulse duration of less than 1.5 ms.

The typical servo accepts pulses between 1 and 2 ms. I found that some converted servos will take pulses between 0.5 and 2.5 ms. For best results, the pulses should be repeated every 20 ms (see Figure 2).

Many Circuit Cellar authors talk about their hardware first. I’ve waited until now because the circuit is ridiculously simple (see Figure 3). I built it on a SimmStick prototyping board from Dontronics. It is a sea-of-holes board.

Because servos expect male headers in groups of three, I used three eight-pin male headers. Power goes along one row, ground the next, and the signals go on the third. I’m only using four signals at this time, even though I have eight headers on the board (see Photo 3).

The PianoRoll is a basic part of the processor’s waveform module. It starts a basic repetitive timer. The PeriodicThread uses this time signal.

I chose to use a PeriodicThread rather than a timer because, frankly, it was easier. Because I am controlling modified servos and electronic speed controllers, I don’t need the fine control that timers or direct control of the waveform module would give me. I know that I’ll do this at some point, just not today.

D. Jay Newman (the “D” is silent) works for the TLT group at Penn State University writing programs to aid the faculty in using classroom-based technology. In his spare time, Jay keeps busy constructing robots and writing. Jay is old enough to have built his first computer around a 6502 with 18 KB of RAM—back when that was a large amount. The JStik that is the brain of his current robot is much more powerful than his first homebuilt computer. Jay lives with his wife, dog, and an annoying parrot. You may contact him at jay@sprucegrove.com.

PROJECT FILES
To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2004/166.

SOURCES
aj-80/100 Java processors
ajile Systems, Inc.
(408) 557-0829
www.ajile.com

SimmStick Bus
Dontronics
www.dontronics.com

JStamp and JStik
Systronix
(801) 534-1017
www.systronix.com