


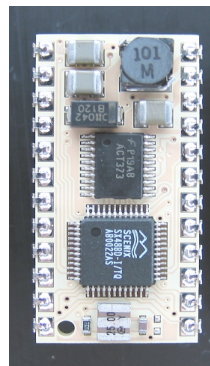
Javelin Stamp Library Submission Guide v1.0

PARALLAX 

599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office/Tech Support: (916) 624-8333
Fax: (916) 624-8003

Web Site: www.javelinstamp.com
Home Page: www.parallaxinc.com

General: info@parallaxinc.com
Sales: sales@parallaxinc.com
Technical: javelintech@parallaxinc.com
Library Submissions: javelinlibrary@parallaxinc.com



Contents

Why Submit a Library Class?	1
Requirements to Submit a Class	1
What is an Application Note?	1
I am not interested in developing an AppNote, can the public still have my code?	2
New to Programming?	2
Required Files for Submission	2
Class Files	2
Schematics	3
Parts	3
Programming Style	3
Formatting	3
Use Two Spaces For Indenting	3
Descriptive Constant, Variable, Method and Class names	4
Constants	4
Variables	4
Methods	4
Class	4
Comments	5
JavaDocs	6
Development Strategies	7
Memory Management	7

Javelin Stamp Library Submission Guide v1.0

Efficiency	8
Reducing Memory Size and Speed.....	8
Objects.....	9
Strings and StringBuffers	9
Variables.....	9
Abstract/Super Classes	10
Summary of Requirements.....	10

Javelin Stamp Library Submission Guide v1.0

Why Submit a Library Class?

There are many benefits for submitting a library class to Parallax, some of them are:

- ✓ \$100 credit towards Parallax manufactured products.
- ✓ The satisfaction that comes from having your code posted publicly to help others worldwide.
- ✓ Have your code professionally reviewed, to allow you to become a better programmer.
- ✓ Acquire the skills and knowledge to create programs with a professional look and feel.
- ✓ Watching your class grow and be used.

Most submitted classes for the Javelin interact with a component while others make coding easier, like a math class. You could also enhance or modify existing classes to make programming the Javelin easier or the code more efficient. If you created or enhanced a class because you had a need for it and it adds value to the Javelin's library, chances are others will also benefit from it, so submit your class to Parallax Inc. and share your class with the world.

Once submitted and accepted, the class will be fully documented by Parallax and released to the public as an Application Note on the Javelin Stamp's website www.javelinstamp.com.

This document will help guide you on the proper format and structure. For questions about submitting a class email us at: javelinlibrary@parallax.com

Requirements to Submit a Class

There are only *four* main requirements when submitting a class for review:

- 1) A commented library class
- 2) A test class
- 3) An application program
- 4) Schematic drawings
- 5) Parts list

For more information on these requirements, please refer to the section titled, "*Required Files for Submission*". The rest of this document explains a proper coding style used by Parallax. Seasoned programmers may wish to go directly to the "*Summary of Requirements*" section.

What is an Application Note?

An application note (*AppNote*) is a package that explains how to integrate the Javelin Stamp and another component. AppNotes are comprised of source code, schematics, parts list, and a detailed description of operation. They are useful working examples that can be easily integrated into your applications. For a complete list of published AppNotes go to the application note section at: www.javelinstamp.com.

Javelin Stamp Library Submission Guide v1.0

Many AppNotes are submitted to Parallax from our customers. These AppNotes have been reviewed, tested and documented by Parallax. They are fully endorsed by Parallax and technical help is available from the Javelin's discussion forum (www.javelinstamp.com discussion link) or email (javelintech@parallax.com).

I am not interested in developing an AppNote, can the public still have my code?

If you are not interested in submitting your class for an application note, but would like to make the class available for others, you can post your code in the Javelin's *discussion* forum, which is accessible from www.javelinstamp.com. You may request the code be uploaded to our web site so it can be downloaded as an individual file via a link by emailing us at: javelinlibrary@parallax.com

New to Programming?

Parallax is proud to be committed to education. We try our best to create an environment of learning through our documentation, website and technical support. As you develop your library class, we will help guide you during the development process. It's a win-win situation for both you and Parallax. The first step is to read the Javelin's user manual and examine the programs, line-by-line and try to understand them from a programmer's perspective. You should read this document and focus on style and form and acquire a good understanding of the basic fundamental aspects of programming the Javelin. You should then be ready to create your own library classes for yourself, Parallax and the Javelin community.

Required Files for Submission

You will need to submit circuit diagrams as well as class files.

Class Files

To develop your library class into an application note, you will also need to supply us with a test class, an application, and possibly a comprehensive library method example/guide.

The class you create will be a *library class*. A library class does not contain a main method, and it does not get tied down to a specific application. Because of this, you will need to have a main method within a separate class (*example code*) that will call the library for a specific application. This way you can have multiple applications interact with a single library class.

The library class developer is expected to produce the following classes:

- A fully developed library class.
 - Class should be as complete as possible (use a datasheet for all available commands and functions).
 - Class should be easily modified for other chips with similar functions. Use methods and constants instead of 'hard coding'.
 - Each method must be fully documented using the JavaDocs format.
- A test class, which is the simplest program needed to test the circuit and verify it is working properly.

Javelin Stamp Library Submission Guide v1.0

- An application, that performs a useful experiment for the circuit that you have constructed. Not all submissions, like a floating-point class, require a circuit.
- A fully commented comprehensive step-by-step class, written as a guide that demonstrates each method within the library.

For current examples of the classes above, please refer to the latest Application Notes.

Schematics

If your class requires a circuit, you must supply a schematic. Your schematic can be in a document (MS Word) or a graphic (BMP, JPG, etc.) file. We will create our own schematic from your drawing, if you are not a good computer drawer, you can fax the schematic to us at 916-624-8003.

Parts

A complete part list should be included. This part list should have the name of the part, a description of the part, the manufacture, and the part number.

Programming Style

There are many styles that programmers use when coding. Each programmer/company has its own style. Not to say one is better than another, but when a programmer/company decides on a particular style, it should be consistent across all programs. A great resource which adheres to industry standards is a small book titled "*The Elements of Java Style*", ISBN: 0 521 777682, which is available from Parallax's website (www.parallax.com).

Formatting

Formatting gives your program's appearance a particular look and feel. If your formatting varies from method to method it makes reading difficult and it looks unprofessional.

Use Two Spaces For Indenting

Some people use the TAB key for indenting, but the TAB key might be pre-set for a different size depending on the application used to create the program. Because of this it has been common to use two or three spaces for indentation. At Parallax we will use two spaces.

Javelin Stamp Library Submission Guide v1.0

Here is the an example using two spaces for each indentation:

```
import stamp.core.*;

public class ButtonLED {
    static boolean P0 = true;

    public static void main() {
        while(true) {
            if (CPU.readPin(CPU.pins[1]) == false) {
                P0 = !P0;
                CPU.writePin(CPU.pins[0],P0);
                CPU.delay(1000);
            }
            else {
                CPU.writePin(CPU.pins[0],true);
            }
        }
    }
}
```

Descriptive Constant, Variable, Method and Class names

Use descriptive, useful names like **keyboardMsg** or **getInt()**, instead of cryptic names like **km** or **get()**. Self explanatory names like this will require less comments and are easily identifiable.

Constants

Constants should always be capitalized. If the constant is more than one word, use the *underscore* for separation. Here are some examples of descriptive constants with proper formatting: **CLS**, **SERIAL_TX_PIN**, **LCD_PIN**

Variables

Variables should also have descriptive names, but should not be in all uppercase. A variable should begin with a lower case letter, and only use an uppercase letter when running two words together. Do not use an underscore. Here are some examples of descriptive variables with proper formatting: **sdaPin**, **sclPin**, **address**, **device**.

Methods

Methods use all of the variable's formatting rules; begin with lowercase letter, and use uppercase letters for multiple words.

Class

Classes use the same formatting rules that a variable uses except the class name must begin with a capital letter.

Javelin Stamp Library Submission Guide v1.0

Comments

Comments are a guide a programmer gives to other programmers; it is an explanation on what the code does and why. Many times as you program you may place comments for your own use. But once your program is completed you should go back through the program, line by line, and comment anywhere that might make it easier for someone reading your code.

Look for sections of code that perform a particular function, like a loop or an **if** statement. These sections of code should be commented by having a commented sentence above the section of code. A blank line could precede this comment, as well as having a blank line at the end of this section of code. Do this if you wish to separate this section of code from the rest of the program.

We don't expect you to comment every line of code of the entire class, but we do expect any confusing code to be commented. Parallax customers will want to completely understand why each piece of code was written the way it was. They will modify and enhance your code where you thought it was not possible. Having fully commented code will help educate them on your class. No need to comment on a print statement, or a break statement, but for incrementing counters, and many other commands, a quick comment telling the user what the line of code does would be very helpful.

The following is an example of a piece of code, which is correctly commented:

```
// This routine will scroll the entire message
for(int i=1;i<(s.length()-3);i++){          // Loop thru message
    strBuf.clear();                          // Clear StringBuffer
    strBuf.append("!LT0ASC");                // Append header code
    for(int j=i;j<i+4;j++) strBuf.append(s.charAt(j)); // Append message
    strBuf.append(zero);                     // Add LED & decimal codes
    strBuf.append(zero);                     // Add colon code
    strBuf=led(strBuf);                       // Add LEDs, colon & decimals
    strBuf=decimal(strBuf);                  // Add decimal codes from message
    txUart.sendString(strBuf.toString());    // Send to AppMod
    CPU.delay(wait);                          // delay
} //end for
```

The above code has a comment on every line. This might look like it's over commented, but how likely would it be that you would understand what **strBuf.append(zero);** did? This line of code does not convey what it does on its own. You can obviously tell it's appending zero, but why?

Javelin Stamp Library Submission Guide v1.0

Here is an example of commenting a whole section of code; this is acceptable since the section is short and uses many variables and methods that are all very descriptive:

```
// When a drag operation is detected, record the initial values and
// replace the asterisk cursor with the character being dragged.
if(mouse.leftDrag){
    xInitial = mouse.xDragStart/scale;
    yInitial = mouse.yDragStart/scale;
    c = display.index(xInitial,yInitial);
    display.placeChar(x,y,c);
}
```

Here is a list of rules for when you comment your code.

- ✓ A comment that precedes a section of code is lined up (vertically) with that section of code.
- ✓ A single space separates the actual comment text from the comment indicator (//).
- ✓ All comments, where applicable, are lined up with each other; normally we try to keep comments at column number 54. If some comments are longer, you will need to pick a smaller column number for that section of code. Of course, if a line of code is too long, adjustments will need to be made.
- ✓ Each closing curly brace has a quick comment on what the brace is for if the opening curly brace contains over 10 lines of code.
- ✓ Multiple closing curly braces should be commented.
- ✓ Every variable declaration must have a descriptive comment.
- ✓ Each method must have a descriptive comment (JavaDoc format preferred) on its function.

JavaDocs

JavaDocs are a special type of comment; they begin with `/**`, and end with a `*/`. A program is used which will read a java class file and extract the JavaDoc comments to create “help” files for programmers. The program that creates these JavaDoc help files requires the Java SDK environment to be installed, which is a very large program (approximately 100 megabytes). If the Java SDK is not installed on your system, do not install it just for the JavaDocs. Parallax will edit your JavaDoc comments for the correct format.

A JavaDoc comment should be included at the very beginning of each class, and for each method. It should introduce the class/method and answer the following questions:

- What’s the class/method for?
- What does the class do?
- Are any pins pre-assigned? Why? What are they?
- Any special circuit information a user will need to know?
- Any special requirements before calling the method?

Javelin Stamp Library Submission Guide v1.0

Here is an example of a simple JavaDoc comment for a method:

```
/** Method to read a single byte from the external eeprom at
 * the current address
 *
 * @param dev device number for external eeprom
 * @return read value
 */
```

Here is an example of a JavaDoc that contains detailed information that the programmer will need:

```
/**
 * Read value from A to D chip.<p>
 * This abstract method is used for a variety of A to D chips.
 * The channels are not implemented the same for each chip,
 * if your chip is not listed below, refer to the chip's datasheet as to
 * what 0, 1, 2 will be translated to.<p>
 *
 * ADC0831: Only has one channel CH0, use 0<br>
 * LTC1298: 0=CH0, 1=CH1, 2=CH0-CH1, 3=CH1-CH0<br>
 *
 * @param channel Specify 0,1,2,3 for specific ADC chip
 * @return raw value from chip
 */
```

Notice that JavaDocs use HTML tags, such <p> and
, the JavaDoc program will use these tags to create a HTML based file which can be viewed by using any web based browser. If you are not familiar with HTML tags, don't include them. We will make the modifications for you.

Development Strategies

The Javelin does not natively support 32-bit math. Be careful that large integer math does not overflow. Check the Javelin's website application page for new math classes. The discussion board may have some nice mathematical routines as well.

Memory Management

To stay true to timing issues of microcontrollers, garbage collection (GC) has been removed from the Javelin. To verify that your program is not losing memory you can take a snap shot of the available memory at the beginning of your class, then compare this with a current snap shot of the available memory within your class. To do this, place this line of code at the beginning of your main method:

```
int mem=Memory.freeMemory();
```

Javelin Stamp Library Submission Guide v1.0

Usually with microcontrollers you will have an endless loop. It is here, within this endless loop, that we will compare the value of `mem` with the current memory, such as with these two lines of code:

```
System.out.println(mem);
System.out.println(Memory.freeMemory());
```

This will print out the original value of `mem`, and then print the current memory available. As your program runs and objects get created you will be able to see the total amount of memory available. Objects get created when you use the `new` constructor, so be sure to only use this constructor outside of loops.

Efficiency

Microcontrollers all have limitations in memory, speed, and size; your code should be tailored with these factors in mind. The Javelin has 32 K, which is a lot for a microcontroller and these classes will be objects, and some programs can get quite large when including many other classes. Classes that perform data logging will also require a large amount of RAM. Writing efficient code is a skill, which takes time to develop. Below you will find some useful guidelines to help you write efficient code.

Reducing Memory Size and Speed

Java is an object-oriented language with garbage collection routines. The Javelin is object-oriented and does not use garbage collection. Once an object is made it cannot be destroyed, and the RAM will remain unclaimed. This really is not a problem when coding for microcontrollers, but be aware of this when creating objects.

Objects are not the only things that consume RAM, the code does as well. Each command uses memory, be observant of ways to use less commands for a particular segment of code. There's many ways to solve a problem, and even more ways to code the algorithm. Doing this not only will reduce the overall size of your class but will increase the speed of your program since each command does take time to execute.

For example, the following piece of code has 3 `if` statements with 5 comparisons:

```
for (int x=0;x<MAX;x++){
System.out.print(x);
    if ((x>90) && (x<MAX))
        System.out.print("A");
    if ((x>40) && (x<=90))
        System.out.print("B");
    if (x<=40)
        System.out.print("C");
}
```

Javelin Stamp Library Submission Guide v1.0

Here's another way to do the same thing as above with only 2 **if** statements with 2 comparisons:

```
for (int x=0;x<MAX;x++){
System.out.print(x);
  if (x>90)
    System.out.println("A");
  else if (x<=40)
    System.out.println("C");
  else
    System.out.println("B");
}
```

This method has 3 less compares, and 1 less **if**. We did add an **else**, but overall this routine will be faster, and take up less memory.

Objects

When you create a new object, be careful that its declaration is within a section of code that executes only once. A microcontroller's objective usually requires the code to be continually executing. This means the code will probably have an infinite loop, which is a loop that never exits. This would not be a good location for the **new** operator, since a new object will be created each time the loop is executed and eventually will use up the available RAM. A good rule to remember is always use the **new** operator outside of a loop. Ideally this should be at the top of the class or at the top of the **main()** method.

Strings and StringBuffer

Strings and StringBuffer are both objects, once created they cannot be destroyed. Because of this, steps have been taken to reduce the number of supporting objects when a String or StringBuffer has been created. Some methods within these objects behave slightly differently than traditional Java.

Variables

The Javelin does recycle variables without the use of garbage collection, but it is good programming practice to be aware when variables get created. There are a few tricks you can use to limit the maximum number of declared variables at any given time. These are discussed below.

It is recommended that all of your primary variables be declared at the top of the class or top of the **main()** method, right along side your objects. This is because your primary variables are used throughout the entire class, and they are easier for people to use when they are all declared in the same area.

Variables declared within a method are considered local variables for that method. They will remain alive if you call other methods from within this method. But, when you exit the method by using a **return** statement, all locally declared variables will be destroyed. This also applies to variables that are declared within a loop

Javelin Stamp Library Submission Guide v1.0

declaration. These variables will only stay alive as long as the loop is executing. Once the loop finishes and exists these locally declared variables will be destroyed.

Abstract/Super Classes

An abstract class is a non-specific class that contains methods that are common among a particular type of components, etc. A good example of this is our AtoD abstract class (*AppNote006*). This class contains methods common to the ADC0831 and LTC1298 analog to digital converters. This makes creating a library class for the ADC0831 (*AppNote007*) and LTC1298 (*AppNote008*) much easier. When constructing your library class, check our AppNote web page first (www.javelinstamp.com) to see if you can utilize one of our abstract classes.

If the component that you are creating a class for is from a family of components each with unique characteristics, and you believe an abstract class would be a benefit to us, please let us know. If we accept your offer, we can offer you an additional \$100 credit for the extra work involved on creating this abstract class.

Please note, abstract classes must be very generalized, great care must be taken in creating an abstract class that can be used with a family of classes. This type of endeavor is usually reserved for seasoned programmers with a high level of familiarity with the family of components which the abstract class is for.

Summary of Requirements

Below is a detailed summary of all the requirements needed, including formatting guidelines, for your library submission.

- A commented library class (*no main method*).
- A test class (*simple class to test circuit*).
- An Application program (*useful program that utilizes your library*).
- Demo class (*explains how to use each library method, if all the methods from your library were not used in your application*).
- Schematics (*if you are using components, a faxed hand drawing is acceptable*).
- Part lists (*part name, description, manufacture, and part number*).
- Formatted code (*Use two spaces to indent code*).
- Descriptive names (*for constants, variables, methods, and classes*).
- Constants are all uppercase, use underscore (`_`) to separate words (***CLS, LCD_PIN***).
- Variables are in lower case, capitalize first letter of adjacent words (***address, sdaPin***).
- Methods are in lower case, capitalize first letter of adjacent words.
- Classes are in lower case, 1st letter capitalized as well as adjacent words.
- Comment code by section is ok, as long as the meaning of the code can be easily understood, otherwise elaborate with in-line comments.
- Keep comments vertically aligned when possible with other comments.
- Comment all variable, constant and object declarations.

Javelin Stamp Library Submission Guide v1.0

- JavaDoc comments should be included for each class, and method.
- Do not use **new** within a loop, there is no garbage collection, objects cannot be destroyed and memory reclaimed. Test code for memory leaks.
- Write efficient code, try to minimize commands and maximize speed.
- Strings and StringBuffer are objects, there is no garbage collection, objects cannot be destroyed and memory reclaimed. Test code for memory leaks.
- If your component is from a family of unique components, you may wish to also create an abstract class for an additional \$100 credit.

Copyright © 2003 by Parallax, Inc. All rights reserved. Javelin, Stamp, and PBASIC are trademarks of Parallax, Inc., and BASIC Stamp is a registered trademark of Parallax, Inc. Windows is a registered trademark of Microsoft Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Other brand and product names are trademarks or registered trademarks of their respective holders.

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products.

