Jade Robot^{TM} C Programming Language Outline

Myke Predko

Last Updated: November 10, 2014

License and Warranty

Software Compatibility

The “Jade C” programming language described in this document was written to be supported by:
Jade Support 0.9.9.24 or later
Robot Software Release 35 or later
Robot Tokenizer Version 0.11.6 or later
_start.s defined as _Header4.script or later

Conventions, Options and Selections

Example code will be put in monospace font like:

```
A = B + C
```

In the language definition, there are a number of instances where there are optional parameters or multiple parameters for the same task. To make these situations more obvious, the following convention is used:

[] – Optional parameter
| - One parameter or another
... – Previous parameter can be repeated
<none> - indicates that nothing is a possible option

Overview

The primary method of programming the Jade Robot™ is the version of the C programming language described here. This language with Jade Support provides a much more capable and sophisticated programming environment than is available from other educational robots.

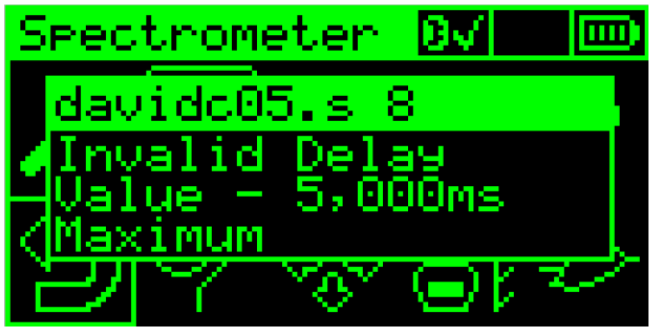
The Jade C language is a procedural, structured language that is syntactically similar to the Java programming language. Its design goals were to be easy to learn with minimal idiosyncrasies or platform dependencies that can be confusing for new programmers.

Errors and Warnings

It should be noted that Jade C has a very simple philosophy with regards to compilation and execution errors and warnings; the current operation stops on the first fail and there are no warnings. Error messages are text with the file the error is located and the line number of the statement.

The reason for this approach is avoid overwhelming a new programmer. Typically one error will result in many additional errors.

With respect to run time errors, if the Jade Robot stops or behaves unexpectedly, remember to check the display on the Jade Robot itself. All six of the green LEDs should flash, indicating that a run-time error has occurred as well as display a box on the display with an error message that provides you with failure information:



```
davidc05.s 8
Invalid Delay
Value - 5,000ms
Maximum
```

Debugging

The Jade C language and Jade Support Integrated Development Environment (IDE) provide the software developer with a source code level debug capability with the ability to:

- Execute at full speed, Single step, Single step over a function call or step out of a function call
- Halt at a specified breakpoint
- Examine Variables and change their values

Debugging and the capabilities listed above can be implemented with a Bluetooth or USB connection.

Data Types

Data types have been kept as simple as possible and Jade C is designed for only two types:

- "int" – signed 32 bit integers (ranging from +2,147,483,647 to -2,147,483,648).
- "str" – ASCII string. String length is part of the string, there is no terminating character.

The Jade Robot uses a 32 bit ARM Cortex M4, so there is no speed penalty defaulting to the 32 bit integer data type.

Program Structure

The mainline of the code consists of statements at the start of the .script file. There is no "main" function like what is found in C or Java.

Functions may follow the mainline code (or be placed in a .function file) but once a function has been placed in a .script file, the mainline code cannot be continued.

Here is an example Jade C program that flashes each of the LEDs on the Jade Robot:

```
//  Count  - Flash each of the LEDs
//         - Stop when button pressed
meta("icon", "LEDicon.b");
meta("description", "LED Flash:Flash LEDs");
//
int                         i;


    for (; "udlre" != syscall(getButton, "");)
    {  //  Wait for buttons to be released
    }

    for (i = 0; "udlre" != syscall(getButton, ""); i = (i + 1) % 6)
    {  //  Loop, incrementing "i" to 5 with no buttons pressed
        syscall(setLED, itos(i) + ":f");  //  Flash LED
        syscall(dlay, "100");
    }
```

Comments

Comments follow the C/C++/Java conventions:

```
/* ... Comments in between slash and splat ... */

//  Comments to the end of the current line
```

Comments can be embedded in a statement and will be ignored.

"meta" Data Statements

"meta" statements are used to embed specific information into the tokenized .script file - they cannot be used in .function files. They take the form:

```
meta(type, dataString);
```

There are two types of "meta" statements:

- "description" which is used to provide a description to the program. In the basic Jade Robot _start.s file, this string, up to a colon (":" – ASCII 0x3A) is displayed on the front panel for the user to provide a text description of the file. Text after the colon is ignored.
- "icon" specifies the bitmap used on the Jade Robot display to specify the program.

During the tokenization process additional meta data is stored automatically in the tokenized file including:

- Date and time the program was loaded into the Jade Robot
- Jade Support IDE used to develop the tokenized file (normally ".s")
- Variable names and associate variable index
- Function names and their offset in the tokenized file

Statements

Jade C follows the statement conventions set out in C and Java; that is to say that variable declarations, assignment statements as well as subroutine calls all end in a semicolon (";") and function entry points along with conditional execution statements end in an open curly brace ("{").

Statements are not line based – they start on the character after the previous statements end character and continue on to the next end character. This means that multiple statements can be placed on the same line, or a single statement can run over multiple lines.

Statement Formatting

To follow structured programming conventions and to make your program easier to read, it is recommended that statements are indented according to the following rules:

1. Major comments start in the first column.
2. Variable declarations start in the first column.
3. Function and subroutine declarations start in the first column.
4. Code starts in column 5 (four spaces from the first column).
5. Code inside conditional execution statements starts four columns to the right of the current code position.
6. Minimize the comments inside your code to just explain what isn't obvious.

Along with this, major blocks of code should be separated by blank lines (also known as "whitespace").

Null Statements

A “null statement” can be a single semicolon (“;”) without any other characters or where there are no statements in between the curly braces (“{“ and “}” – see “Curly Braces”) after a conditional execution statement or inside a function.

Null statements are valid in Jade C and can be used to set placeholders or indicate that nothing is done for a certain condition. They do not produce code or affect the operation of the program execution in any way – their only purpose is to illustrate a facet of the program to the reader.

Labels

Labels are used for variable, function and subroutine as well as macro names. Labels can be up to 30 characters in length. They follow normal conventions and can comprise of the following characters:

- “_” (Underscore), anywhere in the label
- “a”-“z” (Lower case alphabet characters), anywhere in the label
- “A”-“Z” (Upper case alphabet characters), anywhere in the label
- “0”-“9” (Numerics), anywhere in the label EXCEPT the first character

Labels cannot be used as reserved words (see “Reserved Words”).

Valid labels include:

- counter
- test01
- _userID

Invalid labels include:

- 7counter
- _@@
- LabelNameLongerThanThirtyCharactersLikeThisOne
- break

Reserved Words

The following list of words are used in Jade C language and cannot be used as labels (see “Labels”):

- abs
- break
- case
- continue
- default
- do
- else
- for
- if
- int
- itos
- len
- macro
- return
- stoi
- str
- switch
- syscall
- until
- while

Note that while all the reserved words are lower case, they can be used as labels with one or more of their letters set to upper case. For example “Return” is not the same as the reserved word “return”.

Care should be taken in creating labels that are the same reserved words to ensure that the final program does not become confusing to someone going through the program.

Variable Declarations and Assignments

Variables can be declared as of type "int" or "str" with optional initial values or as arrays.

```
int variableName0;                     //  Integer variable
int variableName1 = initialIntValue;   //  Initialized integer variable
int variableName2[5];                  //  Array integer variable
str variableName3;                     //  String variable
str variableName4 = initialStrValue;   //  Initialized string variable
str variableName5[6];                  //  Array string variable
```

Variable names cannot be repeated and an array cannot have an initial value. There can be up to 31 array single dimensional elements. Initial values are actually assignment statements and can include variables and functions.

File Types

As noted above, the mainline of a Jade C program is the ".script" file. This file is also the same name as the project in Jade Support and the associated files stored along with the .script file include:

- .function – functions and subroutines common to this and other projects (see "Functions")
- .panel – display definition file used to simply UI operations (see "Panel Operations" document)
- .bmp – bitmaps used to display images on the Jade Robot's OLED
- .wav – audio files used to output polyphonic sounds from the Jade Robot
- .txt – ASCII text files

Functions

The user can create functions (which return values) and "subroutines" that are accessed either within the currently executing .script or .function file or in other .function files located on the robot's file system. Functions can take an arbitrary number of arguments as either integers ("int") or strings ("str") and optionally return an integer or a string.

```
int sampleFunction(int i, int j, str inputString) {

sampleSubroutine(str inputString1, int count) {
```

See an example of how functions and subroutines are implemented here:

```
//  funcTest - Call a specific function


    func1(4);
```

```
int func2(int i, int j) {

    return i * j;

}


func1(int i) {

    i = i * 2;

}
```

Built in Functions

The only built in functions available to Jade C are:

- abs – return magnitude of signed integer (see "Built in Arithmetic Functions")
- len – return the integer length of a string (see "Built in String Functions")
- itos – convert integer to ASCII string representation (see "String/Integer Conversions")
- stoi – convert ASCII string with digits "0" through "9" to integer (see "String/Integer Conversions")
- syscall – access robot hardware (see "syscall API List")

"return" statement

The "return" statement is used to end a function and return a value to the caller. In functions (see "Functions") it was shown that a function can either return a value (integer or string) or not return a value (in which case it was called a "subroutine"). The return statement can be anywhere in the function or subroutine to provide an immediate return to the caller.

Functions must have a return statement with a value of the type specified in the function declaration, for example:

```
int increment(int i)
{

    return i + 1;

}
```

Subroutines may have a return statement but it cannot have a value associated with it.

Global & Local Variables

Variables declared in the mainline section of the .script file, before any functions are referred to as "Global" variables and are accessible in both the mainline code and function code of the .script file

UNLESS there are "Local" variables declared with the same label in the current function. Global variable labels cannot be repeated.

"Local" variables are declared inside functions and are deleted when the function ends. Local variable labels cannot be repeated within the function. If a Local variable is given the same label as a Global variable, then the local variable is accessed. If a Local variable is given the same label as another Local variable in a function that called the current function, then the Local variable is used in the function but the value of the Local variable of the calling function is not changed.

Assignment Statements

Jade C provides the user with a standard method for assigning values to variables:

```
variableName = Expression;
```

The "Expression" can be of either arithmetic (integer "int" – see "Arithmetic Expressions") or string ("str" – see "String Expressions") types.

The assignment statement can be embedded in another statement, as is done with C or Java programs. If it is not embedded in another statement and is on its own, then it needs to have a semicolon (";") at the end.

Array Variable Usage

Array variables in Jade C are similar to array variables in C or Java. To return the current value of an array variable or to change it, simply specify the variable name and the index:

```
arrayVariable[index]
```

The "index" is an integer value and can be an arithmetic expression (<Put in link to Arithmetic Expressions>). The index value is checked and must be in the range of 0 (zero) to the number of elements the array variable was declared with (up to 31).

Arithmetic Expressions

Arithmetic expressions are used to create integer values from the contents of variables, constants, function return values as well as strings (converted according to "String/Integer Conversions") and processing them using arithmetic operators (see "Arithmetic Operators") such as "+" and "-". They can take the traditional form and can be used in assignment statements or conditional execution statements:

```
value1 operator1 value2
```

Arithmetic Expressions can contain conditional operators (see "Conditional Operators") which are used to compare integer values and return true or false values about them.

Arithmetic Order of Operations

The order of operations (ie which operator is executed before others) is listed in “Arithmetic Operators” or can be overridden by the use of parenthesis (“(“ and “)”) indicates operations which take a higher priority (see “Arithmetic Operator Priority”).

```
value1 * value2 / (value3 + value4)
```

In the example expression above, “value3 + value4” will be evaluated first because it is included in parenthesis even though addition at a lower order of operations than multiplication or division.

The use of parenthesis is an excellent tool for improving the readability of an expression and should be used to indicate the high priority or necessary operations.

Arithmetic Operator Priority

Priority listed in the tables for arithmetic and string operators is given a numeric value with “1” being the highest priority and “9” being the lowest. As noted in Arithmetic Order of Operations (see “Arithmetic Order of Operations”) parenthesis should be used for setting the expected priority of the expression.

Two Parameter Arithmetic Operators

Operation	Operator	Priority	Execution
Addition	+	3	Add two integers together
Subtraction	-	3	Subtract the second integer from the first
Multiplication	*	2	Multiply two integers together
Division	/	2	Divide the second integer into the first
Modulus	%	2	Return the result of the Division operation

Two Parameter Binary Operators

Operation	Operator	Priority	Execution
Left shift	<<	4	Shift the first integer to the left second integer times
Right shift	>>	4	Shift the first integer to the right second integer times
Bitwise AND	&	5	AND the two integers together
Bitwise OR	|	5	Inclusive OR the two integers together
Bitwise XOR	^	5	Exclusive OR the two integers together

Single Parameter Arithmetic Operators

Operation	Operator	Priority	Execution
Negation	-	1	Negate the Integer value
Increment Variable	++	N/A	Increment the contents of the integer variable. If “++” before variable label then executes before other

			operators in expression, if after variable executes after other operators
Decrement Variable	--	N/A	Decrement the contents of the integer variable. If "++" before variable label then executes before other operators in expression, if after variable executes after other operators

Single Parameter Binary Operators

Inverse	~	8	Invert the bits of the integer

Shortcut Assignment Statements

Like C and Java, the Jade C allows for "shortcut" statements which take the form:

```
variable operator = value;
```

which replaces the statement:

```
variable = variable operator value;
```

Any of the two parameter arithmetic operators (see "Two Paramter Arithmetic Operators") can be used as part of a shortcut assignment statement.

Make sure you read the note on the dangers of overloading statements – see "Keep Statements Simple!".

Integer Increment/Decrement Operators/Statements

Integer variable increments and decrement operators are also available in Jade C. To increment a variable and decrement an array element:

```
++variable;  --array [index];  //  array[index] = array[index] – 1;
```

The "++" (increment) and "—" (decrement) can be placed inside a larger expression like:

arrayVariable[++index];

with the position of the operator indicating when the increment/decrement is going to be done (before the variable, before the rest of the statement, after the variable, after the execution of the statement).

Make sure you read the note on the dangers of overloading statements – see "Keep Statements Simple!".

Keep Statements Simple!

The ability to embed shortcut assignment along with increments and decrements is a powerful tool and not only speed up the entry of code by eliminating redundant keystrokes but make it more readable by combining operations into a single statement to make the overall function more clear.

But, "With great power, comes great responsibility."

Care must be taken in the use of shortcut statements and increments/decrements so that statements do not become complex or confusing to the point where even you can't read it.

A good rule of thumb is, no statement should have more than one shortcut or increment/decrement in it to avoid readability issues.

Built In Arithmetic Functions

Currently the only built in arithmetic function is :

```
abs(integerValue)
```

Which returns the absolute value for the integer.

String Expressions

String expressions are used to create string values from the contents of variables, constants, function return values as well as integers (converted according to "String/Integer Conversions") and processing them using string operators (see "String Operators") such as "+". They can take the traditional form and can be used in assignment statements or conditional execution statements:

```
value1 operator1 value2
```

String Expressions can contain conditional operators (see "Conditional Operators") which are used to compare integer values and return true or false values about them.

String Operators

There are only a couple of string operations and they're operation is quite simple:

Operation	Operator	Priority	Execution
Concatenation	+	3	Append "str2" to the end of "str1"
Get Character	.	2	Return the character in string at integer offset

The "Get Character" operator returns a single character which can be stored as an integer or a string as described in "String/Integer Conversions" (see "String/Integer Conversions").

Built In String Functions

Currently the only built in arithmetic function is:

```
len(string)
```

Which returns the length of the string value.

Note that there are some string handling syscall functions as listed in the “syscall API List” that are available for your use to simplify string operations.

String/Integer Conversions

There are two functions used for converting integer and string values:

```
string = itos(integerValue)  //  123 becomes “123”

integer = stoi(string)  //  “123” becomes 123
```

In both cases, the string value consists of the digits ‘0’ through ‘9’.

Conversions to ASCII characters is discussed in “ASCII/Integer Conversions”.

ASCII/Integer Conversions

Along with String/Integer conversions (see “String/Integer Conversions”), conversion between ASCII characters and their integer values can be made quite simply following these rules:

ASCII to Integer is accomplished with a single string character (either a string 1 byte long or a string with the “.” Operator (see “String Operators”).

Integer to ASCII is accomplished with an integer value which is between zero and 255 and can either be assigned directly to a string of 1 byte long or concatenated into a longer string as described in “Integer ASCII to String Concatenation”.

Integer ASCII to String Concatenation

Integer values, which could be ASCII characters (ie they are between 0 and 255 as discussed in “ASCII/Integer Conversions”) can be part of a string concatenation operation like:

```
str stringVariable = “Hello“ + 0x0D;    //  Add CR to end of String
```

Care must be taken when there are two ASCII integers that must be concatenated in a string – using the “+” operator will cause them to be added together into a larger integer value rather than concatenated together. In the case where two or more ASCII integers are to be concatenated into a string, null strings must be concatenated between them like:

```
str stringVariable = “Hello” + 0x0D + “” + 0x0A;
  //  Add CR & LF to end of string
```

Backslash Characters

The following backslash characters are supported in Jade C:

Character	Decimal	Hex	Character Function
\0	0	0x00	Null character
\b	8	0x08	Backspace
\t	9	0x09	Horizontal Tab
\n	10	0x0A	Line Feed
\r	13	0x0D	Carriage Return
\”	34	0x22	Double Quote
\’	39	0x27	Single Quote/Apostrophe

Conditional Execution

Conditional execution is based on the result of a value being “true” or “false” which are defined as:
- “true” for integers means a non-zero value and for strings means a non-zero length string
- “false” for integers means a zero value and for string means a zero length string

Values passed to the conditional statements can be generated using arithmetic (see “Two Parameter Arithmetic Operators”, “Single Parameter Arithmetic Operators”), string (“String Operators”) or conditional (“Conditional Operators”) operators.

The conditional execution statements are:
- “if” statement (see ”if-else Statements”)
- “switch” statement (see ”switch Statements”)
- “while” and “until” statements (see “while and until Loops”)
- “do”-“while” and “do”-“until” loops (see “do-while and do-until” Loops”)
- “for” loops (see ”for Loops”)

While the use of “curly braces” (see “Curly Braces”) is optional for all conditional execution statements except for “switch”, it is recommended that they are always used.

Conditional Operators

The conditional operators used in Jade C follows the conventions used in C and Java with comparison operators, logic operators and a logic “not” operator. All of these operators return the integer -1 for “true” and 0 (zero) for “false” and can be used with integer or string values (as well as combining them).

Comparison	Operator	Priority	Function
Equals	==	6	Return “true” if equals
Not Equals	!=	6	Return “true” if not equals
Less Than	<	6	Return “true” if left is less
Less Than or Equals	<=	6	Return “true” if left is less or equals
Greater Than	>	6	Return “true” if left is greater than
Greater Than or Equals	>=	6	Return “true” if left is greater than or equals

Operator	Operator	Priority	Function

AND	&&	7	Return “true” if both are “true”
OR	||	7	Return “true” if either one is “true”

Operator	Operator	Priority	Function
NOT	!	8	Return “true” if both are “true”

Yoda Syntax

A very common mistake for new programmers is to write an equals comparison as

```
variableA = constantB
```

which is in fact, an assignment statement and can be placed inside a conditional execution statement. This can be extremely difficult to debug or understand what is the problem with the statement, as the double equals (“==”) is not something that is normally intuitive as being “return true if both are equal”.

A simple way to avoid this problem is to place constants or computed values to the left of the equals comparison operator like

```
constantB == variable
```

This is often referred to as “Yoda syntax” as it reverses the flow of the code from what is natural flow but it will cause an error in the compiling step which is easy to fix and avoid the need to debug an embedded assignment when you mean to use an equals comparison.

Curly Braces

Link in C and Java, multiple statements can be conditionally executed if the conditional statement is followed by the open “{“ character. Statements will continually be executed as part of the conditional execution until the close “}” character. For example:

```
if (a == b)
{
    statement1;
    statement2;
      :
}
```

If there are no curly braces then, just the first statement following the conditional execution statement will be executed as in:

```
If (a == b)
    ++a;  //  Only statement executed if “a” equals “b”
```

NOTE: the use of curly braces is not optional for the “switch” (see “switch statement”), “do-while” and “do-until” (see “do-while and do-until Loops”) statements.

“if”-“else” Statements

The “if”-“else” statements follow the C/Java convention providing the program with the ability to execute code if a set condition is true or false. For example:

```
if (a == b)
{
    ++a;  //  This executes if “a” is equal to “b”
}
else
{
    ++b;  //  This executes if “a” is not equal to “b”
}
```

The “else” statement (and its conditional code) are optional.

“switch” Statements

The “switch” statement is used to select between different integer values without the need of multiple “if” statements. It takes the form:

```
switch (integerExpression)
{
case firstInteger:
    :
    break;
case secondInteger:
    :
    break;
:
default:
    :
}
```

The “case” statements are for immediate integer expressions – no variables are allowed. The “default” statement is used to capture execution for situations where the integer expression is not matched by any of the “case” statements.

The “break” will cause execution to leave the “switch” statement and are optional; if they are not used, then execution continues through the next “case” or “default” statements. “break” is discussed here (see “Break Statement”) as a means of leaving loop code which is analogous to the use here.

“while” and “until” Loops

The “while” and “until” loops follow the C/Java convention and provide the ability to loop while a condition is true or until a condition is not true, respectively, and take the form:

```
while (a < b)
{
    ++a;
}

until (a >= b)
{
    ++a;
}
```

Note that both these statements perform the exact same thing; they execute until variable "a" is greater than or equal to "b". By placing the condition statement at the start, if "a" was greater than or equal to "b" when the statements are first encountered, program execution will bypass the code inside the loops.

"do"-"while" and "do"-"until" Loops

The "do - while" and "do - until" loops follow the C/Java convention and provide the ability to loop with a condition check at the end of the loop, rather than at the start in traditional while/until loops (see "while and until Loops") and take the form:

```
do
{
    ++a;
} while (a < b);

do
{
    ++a;
} until (a >= b);
```

"for" Loops

"for" loops in Jade C follow the same conventions as "for" loops in C/Java. This means that multiple initializations or increment statements can be placed in the statement (or none at all). The basic form of the "for" loop is:

```
for (optionalInitialization; optionalCondition; optionalIncrement)
{
    //  Statements executing while "optionalCondition" is true
}
```

Some examples of for statements include:

```
for (i = 0; 25 > i; ++i)
{
    //  Execute code 25x
}

for (i = 0;; ++i)
{
    if (25 == i)
        break;  //  Exit loop after 25 iterations
}
```

Infinite and “do” Loops

There are many ways to implement an “infinite loop” (which is a loop that execution can never leave) such as:

```
while (1)
{
    //  statement(s) inside loop
}

until (0)
{
    //  statement(s) inside loop
}

do
{
    //  statement(s) inside loop
} while (1);

do
{
    //  statement(s) inside loop
} until (0);

for (;;)
{
    //  statement(s) inside loop
}
```

The “do” reserved word can used on its own create a very simple infinite loop:

```
do
{
    //  statement(s) inside loop
}
```

While they are called “infinite loops”, execution can break out of them using the “break” statement (see “break statement”).

“break” Statement

The “break” statement is used to leave loops (“for”- “for Loops”, “while”/“until” - “while and until Loops”, “do” – “Infinite and do Loops”, “do”–“while”/“do”–“until” - “do-while and do-until Loops”) as well as exit “switch” statements (“switch Statements”).

The “break” statement cannot be used for an “if” statement because it can be executed optionally.

“continue” Statement

The “continue” statement is used to return to the end of loops (“for” - “for Loops”, “while”/“until” - “while and until Loops”, “do” – “Infinite and do Loops”, “do”–“while”/“do”–“until” - “do-while and do-until Loops”) to have execution update the loop values (in the case of a “for” statement) followed by testing the condition statement of the loop.

Like the “break” statement (<Put in link to “break” Statement), “continue” can be executed conditionally after an “if” statement.

Keeping things Positive

When creating conditional code it is important to think about what the code looks like and how easily it can be understood by yourself (some time in the future) or by somebody else. One of the things that can make the program difficult to understand is when you put in “negative” logic and the statements needed to support them.

For example, you might want to execute in a loop until a flag variable is false – the loop statement would be:

```
until (!executionFlag)
{
    //  Execute until "executionFlag" is false
}
```

which is a bit convoluted. If you look at it from the “positive”, you would want to execute while the flag is true:

```
while (executionFlag)
{
  //  Execute while "executionFlag" is true
}
```

This avoids the need for the reader to first mentally set up what “until” does and then logically invert the value of “executionFlag” to understand what the code is doing.

This may be a bit of trite example, but there are generally very few cases where the use of “until” or “else” is more readable and efficient than “while” and “if”. The “for” statement executes as if it has a “while” statement built into it, which means that avoiding the use of “until” statement means that all loops perform the same way in your program.

Direct Execution Change

The “break” statement is used to perform an immediate exit to loops and switch code. An example of how break statements are implemented can be seen here:

```
//  brkTest5 - "break" Test
int i = 0;


    do {
        if (4 < i) {
            break;
        }
        ++i;
    } until (6 < i);
```

The “continue” statement is used to jump to a loop test condition. An example of how continue statements are implemented can be seen here:

```
//  conTest2 - "Continue" Test

//  Variables (already done)
int i;


    for (i = 0; 6 > i; ++i) {
        if (4 > i) {
            continue;
        }
    }
```

The “goto” operation is not part of Jade C. While being a generally frowned upon programming construct, adding it to the language would add considerable complexity and overhead, slowing down the execution of application code.

“syscall” APIs

The “syscall” API function is used to access hardware functions in the Jade Robot. The reason for providing the syscall API function is to simplify the hardware interfacing carried out by the user and prevent the user from putting the robot into a state where it cannot execute.

The form of the “syscall” API function is:

```
stringReturn = syscall(integerAPI, stringArgument);
```

Note that data passed to the specific API function as well as returned from it are strings. Integers can be converted to an argument or converted from an argument using the “itos” or “stoi” (see “ASCII String/Integer Conversions”), respectively.

The list of syscall APIs is quite extensive and can be found in the “syscall API List” document.

Panel Operations

The Jade Robot is provided with a "Panel" based user interface that can greatly simplify and speed up user interface operations.

The panel application generally consists of:

- A single ".script" and optional ".function" files which are written in the Jade C language described in this document. Access to the .panel files is implemented by "syscall" API function calls (see "syscall API List")
- A ".panel" file which defines the position of different "controls" on the Jade Robot display
- One or more ".bmp" files which provide a background or control images used in the panel

Input to the .panel file is provided through the button interface built into the robot or a remote control.

Macros

There is a basic macro capability built into Jade C which mimics the capabilities of the C "#define" statement. They can be implemented without any parameters like:

```
macro one                 1
```

or with multiple parameters (which are substituted into the macro):

```
macro addition(a, b)       (a + b)
```

Like C #define statements, macros can be written over multiple lines using the backslash ("\" ASCII 0x5C) character to indicate that the next line is to be concatenated onto the current one:

```
macro longAddition(a, b)   (a +  \
                           b)
```

Macros can be very useful programming constructs (they are used in this language to provide a label to integer conversion for the syscall APIs (see the "syscall API List"), but they should be considered to be an advanced programming concept only used in specific situations following specific rules:

1. There is no way to debug what kind of code is generated, which can be a problem for complex macros.
2. While there is a check for recursive macros by the Jade Support tool, there may be cases where they are missed.
3. Code from the start of the macro to the end of the line (or multiple lines if the "\" character is used) is copied into the source. This means that comments must use the "/* ... */" form to ensure that any following characters are not commented out when the macro text is substituted into the source.
4. It is highly recommended that if the substituted text is an arithmetic or string expression, the expressions are enclosed in parenthesis (as is illustrated in the "addition" macro above).
5. That the macro is only relevant to the current .script or .function file because the same macro cannot be used in multiple files (the same way that a .h file in C allows a single source for a #define that can be used in multiple source files).

Supporting Documents

Currently None

Glossary

ASCII – Standard 8 bit character set. See http://en.wikipedia.org/wiki/ASCII

Document Updates

Date	Changes	Author
2013.12.19	Initial Release	Myke Predko
2014.11.10	Updated to include “Jade Support” and to change language description to “Jade C”. Also changed references from the “Development Suite” to “Jade Support. Added information about runtime Error Messages displayed on the Jade Robot.	Myke Predko