An Introduction & Guide to JALV2

An Introduction & Guide to JALV2

JAL is a high level language designed to hide the general nuisance of programming a MicroChip PIC processor. It is derived from the original JAL by Wouter van Ooijen (see http://www.voti.nl/jal/index.html), which is loosely base on Pascal.

JAL is not case sensitive.

Table of Contents

List of Tables

Chapter 1. Definitions and Conventions

1.1. Definitions

The following abbreviations are used throughout this guide:

bit

A bit within a bit, 0 <= *bit* <= 7

comment

Comments begin with either "--" or ";" and continue through the end of the line.

constant

A numeric constant.

expression

An expression is a sequence of values and operations. Expressions are subdivided into:

cexpr -- constant expression

An expression that can be fully evaluated at compile time. For example 1 + 2.

expr -- any expression.

An expression is anything that evaluates to a value, for example: b + c, x + 1, etc.

lexpr -- logical expression

A logical expression. This differs from an expression in that the result is 0 if the expression is zero, and 1 if the expression is anything other than 0.

identifier

Identifies a variable, constant procedure, function, label, etc. Must begin with a letter or '_' followed by any number of of letters (a-z), digits (0-9), or '_'. Note that identifiers beginning with '_' are reserved for the compiler.

program

A program is simply a sequence of *statement*s. Unlike other languages, in JAL, if the execution runs out of statements, the processor will be put to sleep.

scope

Scope is the 'visibility' of an identifier. Each statement_block creates a new scope. Anything declared within this scope will not be visible once the scope ends.

A variable can be redefined in a block as follows:

```
VAR BYTE x, z
...
IF (x) THEN
  VAR WORD x, y ; all references to x will refer
                ; to this definition
  ...
END IF
...
VAR WORD x ; this is illegal because x already exists
```

statement

A single assignment, definition, control (BLOCK, CASE, IF) or looping (FOR, FOREVER, REPEAT, WHILE).

statement_block

A sequence of statements. Variables, constants, procedures, and functions defined in a statement_block will not be visible outside of the statement_block.

token

The JAL compiler sees only a stream of tokens. An entire program can be written without any line breaks or extra spaces, except of course for comments which are terminated by and end of line.

var -- variable

1.2. Conventions

The following notational conventions are used throughout this guide:

{ a | b | c } -- one of

must be one of a,b,c

KEYWORD -- A JALv2 keyword

Upper case denotes a JALv2 keyword

'...' -- literal

Anything between the quotes must be typed exactly.

[...] -- optional

Anything between the brackets is optional.

Chapter 2. Variables, Constants, Aliases

2.1. Types

The following are the list of types understood by the JALv2 compiler.

Table 2-1. JALv2 Built-in Types

Type	Description	Range
BIT[1]	1 bit boolean value	0..1
SBIT[1]	1 bit signed value	-1..0
BYTE[1]	8 bit unsigned value	0..255
SBYTE[1]	8 bit signed value	-128..127
WORD	16 bit unsigned value	0..65,535
SWORD	16 bit signed value	-32,768..32,767
DWORD	32 bit unsigned value	0..4,294,967,296
SDWORD	32 bit signed value	-2,147,483,648..2,147,483,647
FLOAT[1]	floating point value	+/-10^-44..10^38

[1]base types

The larger types, [S]WORD, [S]DWORD are simply derived from the base types using the width specifier. For example, WORD is equivalent to BYTE*2, the later can be used interchangably with the former.

Floating point arithmetic is *very* expensive in terms of both code and data and should best be avoided. It is nominally based upon IEEE 754, though does not raise exceptions nor handle overflow or special numbers (+/-Infinity, +/-NaN, -0, etc). A floating point value is represented in 4 BYTEs.

A note needs to be made concerning the BIT type. In the original JAL language, the BIT type acted more like a boolean -- if assigned 0, the value stored would be zero, if assigned any non-zero value, the value stored would be one. This convention is still used in JALv2.

However, JALv2 also understands BIT types more like C bitfields. If, instead of BIT one uses the type BIT*1, the value assigned would be masked appropriately (in other words BIT*1 y = z translates internally to BIT*1 y = (z & 0x0001).

Even though the predefined larger types use standard widths (2 and 4), there is no such requirement imposed by the language. If you need a three byte value, use BYTE*3. The only upper limit is the requirement that any value fit within one data bank.

Finally, BIT and BYTE are distinct, so defining a value of BIT*24 is *not* the same as defining a value of BYTE*3!

2.2. Arrays

JAL allows one dimensional arrays of any non-bit type. These are defined during variable definition using the notation:

```
VAR type '[' cexpr ']' id
```

This defines id as type with *cexpr* elements. These are accessed using brackets. The elements are numbered from zero, so for 5 elements the accessors are 0 to 4.

Example:

```
VAR BYTE stuff[5], xx

xx = 2
stuff[0] = 1
stuff[xx] = 2
xx = stuff[xx]
```

Note: There is no error checking when an array is accessed with a variable. In the above example, if xx is 5 no error will be generated, but the results will not be as expected.

2.3. Records

Records are special types, composed of fields which are built-in types, arrays, and/or other records. These are defined with:

```
RECORD identifier IS
    type[*cexpr] id0 [ '[' cexpr ']' ]
    ...
END RECORD
```

Once defined, the RECORD identifier can be use anywhere a simple type can be used. Each individual field is accessed using '.'

Example:

```
RECORD eyeinfo IS
  BYTE left
  BYTE right
END RECORD

;
; a record can be initialized on definition as follows:
;
VAR eyeinfo eye = { 3, 4 }
;
; alternately, each field is accessed with the '.' operator:
;
eye.left = 1
eye.right = 2
;
; A more complex example. This sets eyes[0] to {1,2},
; eyes[1] to {3,4} and eyes[2] to {5,6}:
;
VAR eyeinfo eyes[5] = { { 1, 2 }, { 3, 4 }, { 5, 6 } }
;
; Finally, nested records and arrays are supported
;
RECORD face_r IS
  eyeinfo eyes
  BYTE    nose
  BYTE    freckels[5]
END RECORD

VAR face_r[5] = {
  { { 1,2 }, 3, {4, 5, 6, 7, 8} },
  { { 2,1 }, 3, {8, 7, 6, 5, 4} }
}
```

2.4. Variables

A variable is simply an identifier that holds a value. These identifiers have types associated which define how much space is required to hold the value. The following types are built-in:

The complete format for defining a variable is:

```
VAR [VOLATILE] [SHARED] type[*cexpr] identifier [ '[' [ cexpr ] ']' ]
  [ { AT cexpr [ ':' bit ] | var [ ':' bit ] | '{' cexpr1[',' cexpr2...] '}'
    | IS var }
  [ '=' cexpr | '{' cexpr1',' ... '}' | '"'...'"' ]
  [',' identifier2...]
```

This is, by far, the most complex construct in all of JAL, so I'll describe it one piece at a time below. Once variable definition is understood, everything else is easy!

VAR

Denotes the beginning of a variable definition.

VOLATILE

The VOLATILE keyword guarantees that a variable that is either used or assigned will not be optimized away, and the variable will be only read (or written) once when evaluating an expression.

Normally, if a variable is assigned a value that is never used, the assignment is removed and the variable is not allocated any space. If the assignment is an expression, the expression *will* be fully evaluated. If a variable is used, but never assigned, all instances of the variable will be replaced with the constant 0 (of the appropriate type) and the variable will not be allocated any space.

SHARED

Tells the compiler that this variable exists in shared memory, so there is no need to set bank bits (14 bit cores), or the BSR register (16 bit cores).

type[**cexpr*]

type is one of the predefined types (above). If type is BIT, BYTE, or SBYTE it can be extended using [*cexpr]. For BYTE and SBYTE, this means the variable will be defined as an integer using cexpr bytes, eg WORD is simply shorthand for BYTE*2.

If type is BIT, the definition changes. A BIT variable, as defined in JAL, is really of type boolean. When assigned any non-zero value, it takes on the value of 1. Using the [*cexpr], the definition changes to be more like a C bit field: assignment is masked. For example:

```
VAR BIT*2 cc
```

when assigning to cc, the assignment is:

```
cc = (value & 0x03)
```

identifier

Any valid JAL identifier

’[’ [*cexpr*] ’]’

Defines an array of *cexpr* elements. The array index starts at 0 and continues through (*cexpr* - 1). *cexpr* must be >= 1. An array *must* fit entirely within a single PIC data bank.

If *cexpr* is ommitted, the ’=’ term must exist and the size of the array will be set to the number of initializers present.

BIT arrays are *not* supported.

AT *cexpr* [’:’ *bit*]

Places the new variable at location *cexpr*. If it is a BIT variable, [’:’ *bit*] defines the bit offset with the location. Any location uses for explicit placement will not be allocated to another variable.

AT *var* [’:’ *bit*]

Places the new variable at the same location as an existing variable. Any location uses for explicit placement will not be allocated to another variable.

AT ’{’ *cexpr*1[’,’ *cexpr*2...] ’}’

Places the new variable at multiple locations. On the PIC, many of the special purpose registers are mirrored in two or more data banks. Telling the compiler which locations hold the variable allows it to optimize the data access bits.

IS *var*

Tells the compiler that this identifier is simply an alias for another. This has been deprecated, use "ALIAS *identifier* IS *identifier*1" instead.

’=’ *expr*

Shorthand assignment. The variable will be assigned *expr*.

’=’ ’{’ *expr*1 [’,’ *expr*2...] ’}’

For an array variable, the elements will be assigned *expr*1, *expr*2, ...

’=’ ’"’ ... ’"’

For a variable array, this assigns each ASCII value between ’"’ and ’"’ to one element of the constant array. Unlike C, there is no terminating NUL.

’=’ "..."

For an array variable, the elements will be assigned one the ASCII values inside the quotes.

= "abc" is equivalent to = {"a", "b", "c"}

’,’ *identifier*2...

Allows defining multiple variables with the same attributes:

```
VAR BYTE a,b,c
```

2.5. Constants

2.5.1. Unnamed Constants

An unnamed numeric constant has the type UNIVERSAL, which is a 32-bit signed value. When a value of type UNIVERSAL is used in an operation, it is converted to the type of the other operand.

An exception to above is floating point constants have type FLOAT.

Numeric constants have the following formats:

12 -- decimal
0x12 -- hexadecimal
0b01 -- binary
0q01 -- octal
"a" -- ASCII
"1.23" -- FLOAT

An ASCII constant evaluates to the first character except when used to initialize a constant or variable array in which case each character is used as one entry.

The full format of a floating point constant is:

```
[+|-]###.[###[e[+|-]###]
```

For example:

```
VAR BYTE ch = "123"    ’ ch is set to ’1’
VAR BYTE str[] = "123" ’ str[0] is set to ’1’
                       ’ str[1] is set to ’2’
                       ’ str[2] is set to ’3’
```

An ASCII constant allows the C language escaping rules as follows:

Table 2-2. ASCII Constant Escaping

Sequence	Value
"\ooo"	octal constant
"\a"	bell
"\b"	backspace
"\f"	formfeed
"\n"	line feed
"\qooo"	octal constant
"\r"	carriage return
"\t"	horizontal tab
"\v"	vertical tab
"\xdd"	hexidecimal constant
"\zbbb"	"binary constant"
"\\"	A single '\'

constants other than ASCII constants may also contain any number of underscores ("_") which are ignored, but are useful for grouping. For example: 0b0000_1111

2.5.2. Named Constants

The complete format for defining a named constant is:

```
CONST [type[*cexpr]] identifier [ '[' [ cexpr ] ']' ]
  '=' { cexpr | '{' cexpr1[',' cexpr2...]'}' | '"'...'"'}
  [ ',' identifier2...]
```

CONST

CONST denotes the beginning of a constant definition clause.

type[*cexpr*]

Defines the type of the constant. If none is given, the constant becomes universal type which is 32 bit signed.

'[' [*cexpr*] ']'

Defines a constant array (see array variable types). A constant array will not take any space unless it is indexed at least once with a non-constant subscript. On the PIC, constant arrays consume *code* space, not *data* space, and are limited to 255 elements.

If *cexpr* is ommitted, the size of the array will be determined by the number of initializers used.

'=' *cexpr*

For non-array constants this assigns the value to the constant

'=' '{' *cexpr*[',' *cexpr*2...] '}'

For arrays of constants this assigns the value to each element. There must be the same number of *cexpr*s as there are elements defined.

'=' '"' ... '"'

For an array of constants, this assigns each ASCII value between '"' and '"' to one element of the constant array. Unlike C, there is no terminating NUL.

2.5.3. String Literals (introduced with JALv2.4p)

String literals are enclosed in quotation markes '"'...'"' and can be used where ever an array of characters is allowed. The ASCII constant escaping noted under, 'Unnamed Constants,' applies to each characterw within the string literal.

Note that a string literal terminates with the first NUL characters (0x00).

2.6. Aliases

Aliases allow a multiple identifiers (variables, named constants, sub-programs) to refer to the same object.

The foramt for defining an alias is:

```
ALIAS identifier IS identifier2
```

Often it is useful to allow a variable or constant be refered to by multiple names. For example, if on a certain project pin_a1 is a red LED, you might prefer to refer to it as RED_LED. That way if, on a different project pin_a2 is the red LED, you'd need only change the alias and everything else would continue to work fine.

Chapter 3. Operators, Casting, Expressions, Casting

3.1. Operators

Table 3-1. JALv2 Operators

Operator	Operation	Result
COUNT	returns the number of elements in an array	UNIVERSAL
WHEREIS	return the location of an identifier	UNIVERSAL
DEFINED	determines if an identifier exists	BIT
'(' *expr* ')'	Grouping	Result of evaluating *expr*
'-'[3]	Unary - (negation)	Same as operand
'+'[3]	Unary + (no-op)	Same as operand
'!'	1's complement	Same as operand
'!!'[3]	Logical. If the following value is 0, the result is 0, otherwise the result is 1.	BIT
'*'[35]	Multiplication	Promotion[2]
'/'[35]	Division	Promotion[2]
'%'[5]	Modulus division (remainder)	Promotion[2]
'+'[3]	Addition	Promotion[2]
'-'[3]	Subtraction	Promotion[2]
'<<'	Shift left	Promotion[2]
'>>'[1]	Shift right	Promotion[2]
'<'[3]	Strictly less than	BIT
'<='[3]	Less or equal	BIT
'=='[4]	Equality	BIT
'!='[4]	Unequal	BIT
'>='[3]	Greater or equal	BIT
'>'[3]	Strictly greater than	BIT
'&'	Binary AND	Promotion[2]
'|'	Binary OR	Promotion[2]
'^'	Binary exclusive OR	Promotion[2]

[1]shift right: If the left operand is signed, the shift is arithmetic (sign preserving). If unsigned, it is a

simple binary shift.

[2]promotion: The promotion rules are tricky, here are the cases:

If either operand is FLOAT, the result is FLOAT.
If one of the operands is UNIVERSAL and the other is not, the result is the same as the non-UNIVERSAL operand.
If both operands have the same signedness and width, the result is that of the operands.
If both operands have the same width, and one is unsigned, the result is unsigned.
If one operand is wider than the other, the other operand will be promoted to the wider type.

[3]These operators allow FLOAT types.

[4]Floating point numbers should never be compared for equality due to the imprecise way in which they are stored. Attempting to do so will result in a warning from the compiler. Two different operations which should yield an identical mathmatical result may compare unequal. The correct way to compare two floating point numbers, say A and B, is 'abs((A - B)/B) < 1e-6' (floating point values have a nominal precision of 6 - 9 digits).

[5]Keep in mind that multiplication and division, even between integer types are very expensive operations in both code size and data size (see Chapter 11: Build-in Function).

3.2. Casting

Casting is the operation of changing the type of a value. This can be necessary for a number of reasons: when assigning a larger value to a smaller one, say a WORD to a BYTE, the compiler will issue a warning. An explicit cast will eliminate that warning:

```
VAR WORD xx
VAR BYTE yy
;
; the following assignment will issue:
; warning: assignment to smaller type; truncation possible
;
yy = xx
;
; no warning will be generated below
;
yy = BYTE(xx)
```

In the first case, the compiler wants you to know there might be an issue (a rather common one). In the second case, you've explicitly told the compiler you know these types are different, but that is OK.

Another case where casting is necessary is to guarantee correct promotion during an operation. Take the following:

```
VAR WORD xx
VAR BYTE yy
;
; this is not likely to do what you expect
;
xx = yy * yy
;
; this will generate the correct result
;
xx = WORD(yy) * WORD(yy)
```

Remember that an operator only sees its two operands, it has no other context. Say the value of yy is 255. In the first case xx will be assigned a value of 1: the lower eight bits of the result. In the second case, the value of yy is promoted to a WORD, so xx will be assigned 65025 which is more likely what you would expect.

3.3. Expressions

An expression is simply values (variable or constant) and operators. For example:

```
y = x
y = x + y
y = -x - y
y = (5 + (3 - 2x)) / z
```

Please take time to look at the operator and casting sections, as many bug reports have been generated by a misunderstanding.

Like C, but unlike Pascal, variables of different types can be mixed freely in an expression. In this case, the promotion rules listed under "operators" are in effect.

Chapter 4. Flow Control

4.1. BLOCK

Syntax:

```
BLOCK
  statement_block
END BLOCK
```

Creates a new block. Any variables defined in this block go out of scope at the block. Mainly useful with the CASE statement (below).

4.2. CASE

Syntax:

```
CASE expr OF
  cexpr1[',' cexpr1a...] ':' statement
  [ cexpr2[',' cexpr2a...] ':' statement ]
  [ OTHERWISE statement ]
END CASE
```

expr is evaluated and compared against each *cexpr* listed. If a match occurs, the *statement* to the right of the matching *cexpr* is executed. If no match occurs, the *statement* after OTHERWISE is executed. If there is no OTHERWISE, control continues after END CASE. Unlike Pascal, the behavior is completely defined if there is no matching expression.

Unlike C (but like Pascal) there is no explicit break. After a statement is processed, control proceeds past the END CASE.

Each *cexpr* must evaluate to a unique value.

Example:

```
CASE xx OF
```

```
  1:     yy = 3
  2,5,7: yy = 4
  10:    BLOCK
           yy = 5
           zz = 6
         END BLOCK
  OTHERWISE zz = 0
END CASE
```

Note that only one *statement* is allowed in each case, thus the reason for BLOCK as BLOCK...END BLOCK is considered a single statement.

4.3. FOR

Syntax:

```
FOR expr [ USING var ] LOOP
  statement_block
  [ EXIT LOOP ]
END LOOP
```

statement_block is executed *expr* times. If USING *var* is defined, the index is kept in *var*, beginning with zero and incrementing towards *expr*. If *var* is not large enough to hold *expr*, a warning is generated. If [EXIT LOOP] is used, the loop is immediately exited.

Note: *expr* is evaluated once on entry to the FOR statement.

On normal exit, *var* is equal to *expr*. After, 'EXIT LOOP,' *var* holds whatever value it had at the beginning of the loop.

November 2010 -- a minor enhancement has been made at the request of the users. If *expr* is a *cexpr* and is one larger than *var* can hold, the loop will be exited when *var* rolls over to zero. In this case, on exit *var* will be zero.

Example:

```
VAR BYTE n

FOR 256 USING n LOOP
  ...
END LOOP
```

On exit, n will be zero.

```
xx = 0
FOR 10 LOOP
  xx = xx + 1
  IF (xx = 5) THEN
    EXIT LOOP
  END IF
END LOOP
```

4.4. FOREVER

Syntax:

```
FOREVER LOOP
  statement_block
  [ EXIT LOOP ]
END LOOP
```

statement_block is executed forever unless [EXIT LOOP] is encountered, in which case the loop is immediately terminated. This is commonly used for the main loop in a program because an embedded program like this never ends.

Example:

```
xx = 5
yy = 6
FOREVER LOOP
  READ_ADC()
  CHANGE_SPEED()
  IF (speed = 5) THEN
    EXIT LOOP
  END IF
END LOOP
```

4.5. IF

Syntax:

```
IF lexpr THEN
  statement_block
[ ELSIF lexpr2 THEN
  statement_block ]
[ ELSE
  statement_block ]
END IF
```

This creates a test, or series of tests. The *statement_block* under the first *lexpr* that evaluates to 1 will be executed. Any number of ELSIF clauses are allowed. If no *lexpr* evaluates to true and the ELSE clause exists, the *statement_block* for the ELSE clause will be executed.

A special case of the IF statement is when any *lexpr* is a constant 0. In this case, the statement block is not parsed. This can be used for block comments.

```
IF 0

this is a dummy block that won't even be parsed!

END IF
```

Example:

```
IF x = 5 THEN
  y = 7
ELSIF x = 6 THEN
  y = 12
ELSE
  y = 0
END IF
```

4.6. REPEAT

Syntax:

```
REPEAT
  statement_block
  [ EXIT LOOP ]
UNTIL lexpr
```

statement_block will be executed until *lexpr* evaluates to 1, or until [EXIT LOOP] is encountered.

Example:

```
REPEAT
  xx = READ_ADC
UNTIL (xx < 5)
```

4.7. WHILE

Syntax:

```
WHILE lexpr LOOP
  statement_block
  [ EXIT LOOP ]
END LOOP
```

statement_block will be executed as long as *lexpr* evaluates to a 1, or until [EXIT LOOP] is encountered. This is similar to REPEAT above, the difference being the *statement_block* of REPEAT loop will always execute at least once, whereas that of a WHILE loop may never execute (because the test is done first).

Example:

```
WHILE no_button LOOP
  xx = xx + 1
  IF (xx = 10) THEN
    EXIT LOOP
  END IF
```

```
END LOOP
```

Chapter 5. Other Keywords

5.1. ASSERT

Format:

```
ASSERT expr
```

This is only useful if the "-emu" compiler option has been used, otherwise it is ignored. If *expr* results in a zero value, the emulator will stop at this point.

5.2. INCLUDE

Format:

```
INCLUDE filename
```

This instructs the compiler to stop parsing the current file, open and completely parse the include file, the return to this file on the next line. Note the included file must have an extension of '.jal' and the filename may not begin or end with a space.

Note that it is not possible to include the same file multiple times. Once a file is included, it will not be included again. Also be aware the the filename is taken literally -- no transform is done on it. This should be taken into consideration if you are writing a library as some filesystems are case-sensitive, and others are not, so "MYLIBRARY" and "mylibrary" might be two different files.

Example:

```
INCLUDE 16f877
```

5.3. Message generating

The following keywords generate a message, just as if it came directly from the compiler. Each is followed by a string which will be displayed as part of the message.

5.3.1. _DEBUG

Format:

```
_DEBUG '"' ... '"'
```

Generates a debug message. This will only be seen if the "-debug" compiler option has been used.

Example:

```
_DEBUG "this file is being deprecated"
```

5.3.2. _ERROR

Format:

```
_ERROR '"' ... '"'
```

Generates an error message.

Example:

```
_ERROR "this function should not be used"
```

5.3.3. _WARN

Format:

```
_WARN '"' ... '"'
```

Generates a warning message.

Example:

```
IF !DEFINED(foo) THEN
  _WARN "foo is not defined"
END IF
```

Chapter 6. Sub-programs: Procedures and Functions

Syntax:

```
PROCEDURE identifier [ '(' [VOLATILE] type { IN | OUT | IN OUT } identifier2 [',' ...
  statement_block
END PROCEDURE

FUNCTION identifier [ '(' [VOLATILE] type { IN | OUT | IN OUT } identifier2 [',' ...]
  statement_block
END FUNCTION
```

The only difference between a PROCEDURE and a FUNCTION, is the former does not return a value, while the later does. The procedure *identifier* exists in the block in which the procedure is defined. A new block is immediately opened, and all parameters exist in that block. A parameter marked IN will be assigned the value passed when called. A parameter marked OUT will assign the resulting value to parameter passed when called. While in a sub-program, a new keyword is introduced:

```
RETURN [ expr ]
```

When executed, the sub program immediately returns. If the sub program is a FUNCTION, *expr* is required. If it is a PROCEDURE, *expr* is forbidden.

A sub-program is executed simply by using its name. If parameters are specified in the sub-program definition, all parameters are required, otherwise none are allowed. A FUNCTION can be used anywhere a value is required (in expressions, as parameters to other sub-programs, etc). There is no limit to the number of parameters.

JALv2 is a pass by value language. Conceptually, an IN parameter is read once when the sub-program enters, and an OUT parameter is written once when the sub-program returns. This is not always desired. For example if a sub-program writes a string of characters to the serial port (passed as parameter), only the last character written will be sent. For this case we need VOLATILE parameters. These are either read each time used (IN) or written each time assigned (OUT). This is accomplished using pseudo variables (see below). If the value passed is not a pseudo-variable, a suitable one is created.

There are two ways to pass an array into a sub-program:

```
PROCEDURE string_write (BYTE IN str[5]) IS...
PROCEDURE string_write (BYTE IN str[]) IS...
```

The first follows the pass-by-value symantics noted above. An array variable of size 5, str, is allocated in the namespace of the procedure. Any callers must call with an array of exactly 5 bytes, which is copied into the local variable and used.

Alternately, the second version created a flexible array. This is pass-by-reference which means (1) the amount of data space used for str is only two or three bytes, and (2) any sized array can be passed in. This is generally far more useful, and far less wasteful. The operator COUNT can be used to determine the size of the array passed in.

Procedures and functions can be nested.

Example:

```
FUNCTION square_root (WORD IN n) RETURN WORD IS

  WORD result
  WORD ix

  ix = 1
  WHILE ix < n LOOP
    n = n - ix
    result = result + 1
    ix = ix + 2
  END WHILE
  RETURN result

END FUNCTION

xx = square_root(xx)
```

Recursion is fully supported but due to the overhead it is discouraged.

Chapter 7. Pseudo-variables

Syntax:

```
PROCEDURE identifier "'" PUT '(' type IN identifier2 ')' IS
  statement_block
END PROCEDURE

FUNCTION identifier "'" GET RETURN type IS
  statement_block
END FUNCTION
```

A pseudo-variable is a sub-program, or pair of sub-programs that work as if they are variables. If a 'PUT procedure is defined, any assignment to *identifier* is replaced by a call to the *identifier*'PUT procedure. Similarly, if a 'GET function is defined, any time the associated value is used is an implicit call to the function.

If both a 'GET and 'PUT sub-program are defined, the parameter type of the 'PUT must match the return type of the 'GET.

Example:

```
FUNCTION pin'GET() RETURN BIT IS
  return pin_shadow
END FUNCTION

PROCEDURE pin'PUT(BIT in xx) IS
  pin_shadow = xx
  port = port_shadow
END PROCEDURE

pin = 5
```

Chapter 8. Interrupts

Syntax:

```
PROCEDURE identifier IS
  PRAGMA INTERRUPT [FAST]
  statement_block
END PROCEDURE
```

PRAGMA INTERRUPT tells JAL that this procedure can only be called by the microcontroller's interrupt processing. Any number of procedures can be defined as an interrupt handler. When an interrupt occurs, first the microprocessor state is saved, then control passes to the first procedure marked as an interrupt handler. Control continues to pass to each interrupt handler until the last, then the microprocessor state is restored and the interrupt ended. The programmer is responsible for clearing whatever bits caused the interrupt to happen. A procedure marked as an interrupt handler cannot be called directly from elsewhere in the program. Beyond that, an interrupt handler can do anything any other procedure can do. The order the interrupt handlers are called is undefined, the only guarantee is each handler will be called at each interrupt, and will only be called once.

If an interrupt handler executes a sub-program that is also executed by the main body of the program, that sub-program will be marked recursive and incur the recursion overhead each time it is called.

If FAST is declared, the interrupt handler will only save the minimum amount of state necessary. This must be used with great care -- although the microprocessor state is saved, state used internally by the compiler is not. As such, only a completely assembly sub-program should be used. Any JAL statements might invalidate the internal state of the compiler. If any interrupt handler is marked FAST then only one interrupt handler is allowed.

Chapter 9. Tasks

Syntax:

```
TASK identifier [ '(' parameter list ')' ] IS
  statement_block
END TASK
```

JALv2 introduces the concept of TASKs which are a form of co-operative multi-tasking. Unlike preemptive multi-tasking, where control passes from one task to another automatically, control will only pass when a task specifically allows it. Due to the architecture of a PIC, true multi-tasking is very difficult. Tasks can only be started by the main program, or within another task. Tasks are started with:

```
START identifier [ '(' parameter list ')' ]
```

When a task is ready to allow another to run, it executes:

```
SUSPEND
```

To end the task, simply RETURN or allow the control to pass to the end of the task. If tasks are used, the compiler must be passed the argument, "-task n," where n is the number of concurrent running tasks. Remember that the main program itself is a task, so if you plan to run the main program plus two tasks, you'll need to pass in, "-task 3".

Finally, only one copy of the body of a task should be run at a time. The following would be an error because it attempts to run two copies of task1 at the same time:

```
START task1
START task2

FOREVER LOOP
  SUSPEND
END LOOP
```

Chapter 10. Assembly

When all else fails, one can resort to inline assembly. This can be in the form of a single statement:

```
ASM ...
```

or an entire block:

```
ASSEMBLER
  statements
END ASSEMBLER
```

Using assembly should be a last resort -- it is needed only when either a feature is not possible using JALv2 (for example, the TRIS and OPTION codes), or when speed is of the essence. JALv2 includes the entire assembly language set in the PIC16F87x data sheet, several instructions from earlier micro controllers, and several common macros. There is some support for the 16 bit keywords.

To guarantee the correct data bank is selected when accessing a file register, use one of the following:

```
BANK opcode ...
```

or

```
BANK f
```

The former takes the file register from the command, the later takes it directly.

Similarly, to guarantee the correct page bits are set (for GOTO or CALL), use one of the following:

```
PAGE opcode ...
```

or

```
PAGE lbl
```

Again, the former takes the label from the command, the later takes it directly.

Normally, the codes to set or clear the bank or page bits are only generated when necessary. If the bits are already in the correct states, no further commands are generated. If you need to guarantee the codes are always generated, use the following pragmas:

```
PRAGMA KEEP PAGE
PRAGMA KEEP BANK
```

The former will keep any page bits, the later and bank bits. These affect the entire sub-program in which they are declared.

To declare a local label for use in CALLs and/or GOTOs:

```
LOCAL identifier[',' identifier2...]
```

Once declared, a label is inserted into the assembly block by making it the first part of a statement, followed by a ':':

```
identifier: opcode...
```

The available opcodes are listed below. For a full description see the appropriate data sheet.

Note that when using inline assembly you should not modify the bank or page registers, FSR, or BSR. If these are modified, it is the programmers responsibility to return them to their original states.

10.1. Available Op-codes

The following abbreviations are used:

b -- bit number, 0 <= b <= 7
d -- destination, 'f' or 'w'
f -- file register or variable
n -- literal value, 0 <= n <= 255 unless otherwise noted
k -- label or constant

Note that not all opcodes are available on all devices. Check the datasheet for a complete description.

addwf f,d
addwfc f,d
andwf f,d
clrf f
clrw
comf f,d
decf f,d
decfsz f,d
incf f,d
incfsz f,d
iorwf f,d
movf f,d
movwf f
nop
rlf f,d
rlcf f,d
rlncf f,d

rrf f,d
rrcf f,d
rrncf f,d
subwf f,d
swapf f,d
xorwf f,d
bcf f,b
bsf f,b
btfsc f,b
btfss f,b
addlw n
andlw n
call k
clrwdt
goto k
iorlw n
movlw n
retfie
retlw n
return
sleep
sublw n
xorlw n
tblrd { * | *+ | *- | +* }
tblwt { * | *+ | *- | +* }
reset
option
tris n (5 <= n <= 9)

10.2. Common Macros

addcf f,d
adddcf f,d
b k
bc k
bdc k
bnc k
bndc k
bnz k
bz k
clrc
clrdc
clrz
lcall k
lgoto k
movfw f
negf f
setc
setdc
setz
skpc
skpdc
skpnc
skpndc
skpnz
skpz
subcf f,d
subdcf f,d
tstf f

10.3. Data Directives

The following allow data to be directly inserted into the code area. Retrieving these data is chip-specific. Also, as the data go directly into the program memory, the amount of space actually used is chip specific.

Below, the term *list* is a comma separated list of constants or strings.

db *list*

Inserts a list of bytes, one per program word.

dw *list*

Inserts a list of words. On 12 & 14 bit cores each word can be 14 bits (0..8191), whereas on 16 bit cores each word can be 16 bits (0..65535).

ds *list*

Pack two 7-bit values into a program word. Not necessary on the 16 bit cores.

Chapter 11. Built-in Functions

JALv2 attempts to be a minimal language with most complex operations done with sub-programs, however some functions simply cannot be efficiently supported externally.

11.1. Multiplication, Division, Modulus Division

Multiplication, Division, and Modulus Division are internal mainly because there is no way to predetermine the size of the operands. Note that unlike the other operators which are done inline, these are function calls and require one stack entry when used!

A second reason for having these built in is the optimizer -- when a multiplication or division by 1 is done, the operation is ignored. When a multiplcation or division by a power of two is done, the resulting code is performed using shifts instead.

For both of these operations, the code generated will be that required for the largest operands unless -fastmath is passed to the compiler.. For example, if the operation occurs only between two BYTEs, the 8-bit routine will be generated. If it occurs between BYTEs and WORDs, the 16-bit routine will be generated.

If -fastmath is used, a different function will be generated for each argument type.

The compiler keeps track of the last operation, so if you find yourself needing both the division result and the remainder of, a certain operation, make sure to put the assignments close together, thus saving a function call:

```
n = x / 10
r = x % 10
```

will only result in one call to the division -- the assignment to r will be a simple assignment.

11.2. Floating Point Operations

Most floating point operations are done via function calls because of the size of the code generated, so at least one stack entry will be used per operation. For multipllcation and division, two stack entries are required because these rely on the integer routines.

Operations that do not require a function call include: multiplication or division by a power of 2, assignment from a constant.

11.3. _usec_delay(*cexpr*);

_usec_delay(*cexpr*) is useful when an exact delay is required. It generates code that is guaranteed to delay a given number of micro-seconds. This is done using loops with one, two, or three variables, and no-op instructions as necessary.

For _usec_delay to work correctly, interrupts must be disabled, and ‘PRAGMA TARGET CLOCK’ must be issued to set the system clock speed.

Note that _usec_delay() will generate delays up to 4,294.967295 seconds (or ~71.5 minutes), this isn’t really the best use of space. On a 20MHz 16f877 this required 1043 instructions.

This is typically used for delays of a few 10s or 100s of uSec.