# 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


**1. Definitions and Conventions ..... 1**

1.1. Definitions ..... 1

1.2. Conventions ..... 2

**2. Variables, Constants, Aliases ..... 3**

2.1. Types ..... 3

2.2. Arrays ..... 3

2.3. Records ..... 4

2.4. Variables ..... 5

2.5. Constants ..... 7

2.5.1. Unnamed Constants ..... 7

2.5.2. Named Constants ..... 8

2.6. Aliases ..... 9

**3. Operators, Casting, Expressions, Casting ..... 10**

3.1. Operators ..... 10

3.2. Casting ..... 11

3.3. Expressions ..... 12

**4. Flow Control ..... 13**

4.1. BLOCK ..... 13

4.2. CASE ..... 13

4.3. FOR ..... 14

4.4. FOREVER ..... 15

4.5. IF ..... 15

4.6. REPEAT ..... 16

4.7. WHILE ..... 17

**5. Other Keywords ..... 19**

5.1. ASSERT ..... 19

5.2. INCLUDE ..... 19

5.3. Message generating ..... 19

5.3.1. _DEBUG ..... 19

5.3.2. _ERROR ..... 20

5.3.3. _WARN ..... 20

**6. Sub-programs: Procedures and Functions ..... 22**

**7. Pseudo-variables ..... 24**

**8. Interrupts ..... 25**

**9. Tasks ..... 26**

**10. Assembly ..... 27**

10.1. Available Op-codes ..... 28

10.2. Common Macros ..... 29

10.3. Data Directives ..... 29

**11. Built-in Functions ..... 31**

11.1. Multiplication, Division, Modulus Division ..... 31

11.2. _usec_delay(*cexpr*); ..... 31

# List of Tables

2-1. JALv2 Built-in Types ..... 3
2-2. ASCII Constant Escaping ..... 8
3-1. JALv2 Operators ..... 10

# 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 |

[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.

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

VAR eyeinfo eye

eye.left = 1
eye.right = 2
```

# 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] 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 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.

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.

Numeric constants have the following formats:

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

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.

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 |
|---|---|
| "\0qqq" | octal constant |
| "\a" | bell |
| "\b" | backspace |
| "\f" | formfeed |
| "\n" | line feed |
| "\r" | carriage return |
| "\t" | horizontal tab |
| "\v" | vertical tab |
| "\xdd" | hexidecimal 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.

## 2.6. 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* |
| '-' | Unary - (negation) | Same as operand |
| '+' | Unary + (no-op) | Same as operand |
| '!' | 1's complement | Same as operand |
| '!!' | Logical. If the following value is 0, the result is 0, otherwise the result is 1. | BIT |
| '*' | Multiplication | Promotion[2] |
| '/' | Division | Promotion[2] |
| '%' | Modulus division (remainder) | Promotion[2] |
| '+' | Addition | Promotion[2] |
| '-' | Subtraction | Promotion[2] |
| '<<' | Shift left | Promotion[2] |
| '>>'[1] | Shift right | Promotion[2] |
| '<' | Strictly less than | BIT |
| '<=' | Less or equal | BIT |
| '==' | Equality | BIT |
| '!=' | Unequal | BIT |
| '>=' | Greater or equal | BIT |
| '>' | 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 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.

## 3.2. 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. 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.

There have been a few comments that only generating the largest routine can be wasteful. This is true, but I decided long ago to minimize the code size. If you're doing multiplication or division, it's going to be slow.

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. _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.

# JALv2 Compiler Options

**JALv2 Compiler Options**

There are many options that can be passed to the compiler to tell it, for example, where to find library files, or where to put the output files. These options are all described here.

See the JALv2 documentation for definitions and conventions. Any time multiple options are allowed, the default option is preceded with a '*'. An {empty} option is interpreted as the default option.

All available compiler options can be seen by passing the single options "--help" to the compiler. Use this command to also see the defaults for each option.

# Table of Contents

**1. File Options ..... 1**
1.1. [no-]asm ..... 1
1.2. [no-]codfile ..... 1
1.3. [no-]hex ..... 1
1.4. include ..... 2
1.5. include path ..... 2
1.6. [no-]log ..... 2
1.7. [no]-lst ..... 2
**2. Misc. ..... 4**
2.1. clear ..... 4
2.2. quiet ..... 4
2.3. task ..... 4
**3. Bootloader ..... 5**
3.1. bloader ..... 5
3.2. fuse ..... 5
3.3. long start ..... 5
3.4. rickpic ..... 5
3.5. loader18 ..... 6
**4. Warnings ..... 7**
4.1. all ..... 7
4.2. codegen ..... 7
4.3. conversion ..... 7
4.4. directives ..... 7
4.5. misc ..... 7
4.6. range ..... 8
4.7. stack overflow ..... 8
4.8. truncate ..... 8
**5. Optimizations ..... 9**
5.1. cexpr reduction ..... 9
5.2. const detect ..... 9
5.3. expr reduction ..... 9
5.4. expr simplify ..... 10
5.5. load reduce ..... 11
5.6. temp reduce ..... 11
5.7. variable frame ..... 11
5.8. variable reduce ..... 12
5.9. variable reuse ..... 12
**6. Compiler Debugging ..... 14**
6.1. codegen ..... 14
6.2. debug ..... 14
6.3. pcode ..... 14
6.4. emu ..... 14
6.5. deadcode ..... 14

# Chapter 1. File Options

## 1.1. [no-]asm

Format:

```
-asm name
-no-asm
```

Set the name of the generated assembly file to ‘name’. The default is the program name with ‘.jal’ replaced by ‘.asm’, or ‘.asm’ appended if the program name doesn’t end in ‘.jal’.

If ’-no-asm’ is specified, no ’.asm’ file will be generated.

The assembly file can be compiled by the MPASM and hopefully generate the same HEX file as JALv2.

## 1.2. [no-]codfile

Format:

```
-codfile name
-no-codfile
```

Set the name of the generated COD file to ‘name’. The default is the program name with ‘.jal’ replaced by ‘.cod’, or ‘.cod’ appended if the program name doesn’t end in ‘.jal’.

If ’-no-codfile’ is specified, no ’.cod’ file will be generated.

The COD file is a symbol file used by MPASM and probably other debugging tools. The format was created and is maintained by Byte Craft Limited and its sharing of this format, and general support is very much appreciated.

## 1.3. [no-]hex

Format:

```
-hex name
-no-hex
```

Set the name of the generated HEX file to ‘name’. The default is the program name with ‘.jal’ replaced by ‘.hex’, or ‘.hex’ appended if the program name doesn’t end in ‘.jal’.

If ’-no-hex’ is used, no HEX file will be generated.

The HEX file is used by PIC programmers and bootloaders to load the program onto the microcontroller.

## 1.4. include

Format:

```
-include filename [ ';' filename2... ]
```

include ’filename’ before parsing the file. Multiple files can be included when separated by ’;’ or when multiple ’-include’ directives are used.

## 1.5. include path

Format:

```
-s path [ ';' path1... ]
```

Set the include path, elements separated with ’;’. Multiple "-s" options append more path elements.

## 1.6. [no-]log

Format:

```
-log filename
-no-log
```

Generate a log file which will contain all messages emitted by the compiler. If absent, standard output is used.

## 1.7. [no]-lst

Format:

```
-lst filename
-no-lst
```

Set the name of the listing file to filename, or prevent the generation of a listing file. The default is no listing file. The listing file has never been correctly generated, so this option is useless.

# Chapter 2. Misc.

## 2.1. clear

Format:

```
-[no-]clear
```

Clear data area on program entry. This does not clear user placed or variables, or variables marked VOLATILE.

## 2.2. quiet

Format:

```
-[no-]quiet
```

Turns off the progress messages

## 2.3. task

Format:

```
-task cexpr
```

Sets the maximum number of concurrent tasks to *cexpr*. The default is 0. Note this value must be one more than the number of concurrent tasks as the main program counts as a task.

nb: It is better to use 'PRAGMA TASK' in your program than setting the value here. Doing so guarantees the value is correct even if you forget to pass it during compilation.

# Chapter 3. Bootloader

Booloaders are tiny programs that allow a chip to be reprogrammed over the serial or USB ports, eliminating the need for extra programming hardware. There are several variants, and each has slightly different requirements of the program it hosts. These are the ones currently defined.

## 3.1. bloader

Format:

```
-bloader
```

Using the screamer/bloader PIC loader. See "PRAGMA BOOTLOADER BLOADER".

## 3.2. fuse

Format:

```
-[no-]fuse
```

Put the '__config' line in the assembly or HEX files

## 3.3. long start

Format:

```
-long-start
```

Force the first generated instruction to be a long jump. See "PRAGMA BOOTLOADER LONG_START".

## 3.4. rickpic

Format:

```
-rickpic
```

Assumes the target PIC is using Rick Farmer's PIC bootloader. See "PRAGMA BOOTLOADER RICKPIC".

## 3.5. loader18

Format:

```
-loader18 [ cexpr ]
```

See "PRAGMA BOOTLOADER LOADER18"

# Chapter 4. Warnings

## 4.1. all

Format:

```
-W[no-]all
```

enable/disable all warnings

## 4.2. codegen

Format:

```
-W[no-]codegen
```

enable/disable code generation warnings

## 4.3. conversion

Format:

```
-W[no-]conversion
```

enable/disable signed/unsigned conversion warning

## 4.4. directives

Format:

```
-W[no-]directives
```

enable/disable warning when a compiler directive is found

## 4.5. misc

Format:

```
-W[no-]misc
```

enable/disable uncategorized warnings

## 4.6. range

Format:

```
-W[no-]range
```

enable/disable value out of range warnings

## 4.7. stack overflow

Format:

```
-W[no-]stack-overflow
```

issue a warning on hardware stack overflow instead of an error

## 4.8. truncate

Format:

```
-W[no-]truncate
```

enable/disable possible truncation in assignment warning

# Chapter 5. Optimizations

Optimizations that are *enabled* by default are considered safe -- they have been well tested and shown to not cause any problems. Opzimizations that are *disabled* by default have not been as well tested. Using them might be unsafe. Before reporting any problems with the compiler, please make sure to retest with only the default optimizations and let me know the outcome.

## 5.1. cexpr reduction

Format:

```
-[no-]cexpr-reduction
```

enable/disable constant expression reduction

Default: enabled

Purpose: Detect expressions or subexpressions composed completely of constants, and reduce these to a single value.

## 5.2. const detect

Format:

```
-[no-]const-detect
```

enable/disable constant detection

Default: disabled

Purpose: Look for variables that are only assigned a single, constant value and replace them with that constant value.

## 5.3. expr reduction

Format:

```
-[no-]expr-reduction
```

enable/disable expression reduction

Default: enabled

Purpose: eliminate or refactor many simple operations and identities:

- x +/- 0 --> x
- x - 0 --> x
- 0 - x --> -x
- x - x --> 0
- x * 0 --> 0
- x * 1 --> x
- x * (-1) --> -x
- x / 1 --> x
- 0 / x --> 0
- x / (-1) --> -x
- x % 0 --> 0
- x % 1 --> 0
- x < 0, x unsigned --> 0
- x <= 0, x unsigned --> x == 0
- x >=0, x unsigned --> 1
- x > 0, x unsigned --> x != 0
- x & 0 --> 0
- x & x --> x
- x | 0 --> x
- x | x --> x
- x ^ 0 --> x
- x ^ x --> 0
- -x, x is single bit --> x
- 0 << x --> 0
- x << 0 --> x
- x << C, where C is >= bit size of x --> x

## 5.4. expr simplify

Format:

```
-[no-]expr-simplify
```

enable/disable expression simplification

Default: disabled

Purpose: Combine common subexpressions.

## 5.5. load reduce

Format:

```
-[no-]load-reduce
```

enable/disable redundant load of W removal

Default: disabled

Purpose: Look for redundant loads into the working register (W) and remove them

## 5.6. temp reduce

Format:

```
-[no-]temp-reduce
```

enable/disable temporary reduction

Default: disabled

Purpose: Look for assignments to a temporary which is then immediately assigned to another variable and never again used, and remove it.

## 5.7. variable frame

Format:

```
-[no-]variable-frame
```

allocate variables into a full frame

Default: disabled

Purpose: Place all variables in a function into a single block instead of using the first available space. This can minimize the number of bank switching operations, but can also lead to less efficient use of the data areas.

## 5.8. variable reduce

Format:

```
-[no-]variable-reduce
```

enable/disable unused or unassigned variables removal

Default: enabled

Purpose: If a variable is assigned a value but never used in an expression, it is removed. Likewise if a variable is never assigned a value, but is used in an expression it is replaced with the constant 0.

## 5.9. variable reuse

Format:

```
-[no-]variable-reuse
```

enable/disable reusing variable space

Default: enabled

Purpose: If two variables, say X and Y, are never, 'live,' at the same time they can share the same data location. This leads to far more efficient use of the data area, but currently can lead to extreme compile times (on the order of hours).

# Chapter 6. Compiler Debugging

These options are most useful for debugging the compiler itself.

## 6.1. codegen

Format:

```
-[no-]codegen
```

do not generate any assembly code

## 6.2. debug

Format:

```
-[no-]debug
```

show debug information

## 6.3. pcode

Format:

```
-[no-]pcode
```

show pcode in the asm file

## 6.4. emu

Format:

```
-[no-]emu
```

Run the emulator after compiling.

## 6.5. deadcode

Format:

```
-[no-]deadcode
```

enable dead code elimination

# JALv2 PRAGMAs

**JALv2 PRAGMAs**

There are many extra things the compiler either needs to know to do its job, or modify its behavior to suit a particular need. This information is passed to the compiler with something called a PRAGMA. This file describes every PRAGMA the compiler understands.

See the JALv2 documentation for definitions and conventions. Any time multiple options are allowed, the default option is preceded with a '*'. An {empty} option is interpreted as the default option.

# Table of Contents


**1. Chip Configuration ..... 1**

1.1. TARGET CHIP ..... 1
1.2. TARGET CLOCK ..... 1
1.3. TARGET FUSES ..... 1
1.4. TARGET opt tags ..... 2

**2. Compiler Configuration ..... 3**

2.1. BOOTLOADER ..... 3
2.1.1. BLOADER ..... 3
2.1.2. LOADER18 [ *cexpr* ] ..... 3
2.1.3. LONG_START ..... 3
2.1.4. RICKPIC ..... 4
2.2. CLEAR ..... 4
2.3. EEDATA ..... 4
2.4. FUSE ..... 4
2.5. IDDATA ..... 5
2.6. TASK ..... 5

**3. Global Configuration ..... 6**

3.1. ERROR ..... 6
3.2. NAME ..... 6
3.3. SIZE ..... 6
3.4. SPEED ..... 6

**4. PROCEDURE/FUNCTION Configuration ..... 8**

4.1. FRAME ..... 8
4.2. INLINE ..... 8
4.3. INTERRUPT ..... 8
4.3.1. * NORMAL ..... 9
4.3.2. FAST ..... 9
4.3.3. RAW ..... 9
4.4. JUMP_TABLE ..... 9
4.5. KEEP ..... 9
4.6. NOSTACK ..... 10

**5. Optimization PRAGMAs ..... 11**

5.1. EXPR_REDUCE ..... 11
5.2. CEXPR_REDUCE ..... 11
5.3. CONST_DETECT ..... 11
5.4. LOAD_REDUCE ..... 12
5.5. TEMP_REDUCE ..... 12
5.6. VARIABLE_FRAME ..... 13
5.7. VARIABLE_REUSE ..... 13

**6. Warning ..... 14**

6.1. ALL ..... 14
6.2. BACKEND ..... 14
6.3. CONVERSION ..... 14
6.4. DIRECTIVES ..... 14

6.5. MISC ..... 15
6.6. RANGE ..... 15
6.7. STACK_OVERFLOW ..... 15
6.8. TRUNCATE ..... 16
**7. Chip Specification ..... 17**
7.1. CODE ..... 17
7.2. DATA ..... 17
7.3. EEPROM ..... 17
7.4. FUSE_DEF ..... 17
7.5. ID ..... 18
7.6. STACK ..... 18
7.7. TARGET CPU ..... 18
7.8. TARGET BANK ..... 19
7.9. TARGET PAGE ..... 19
**8. Debugging ..... 20**
8.1. CODEGEN ..... 20
8.2. PCODE ..... 20

# Chapter 1. Chip Configuration

Select various attributes of a chip.

## 1.1. TARGET CHIP

Syntax:

```
PRAGMA TARGET CHIP chipname
```

Analogous to:

```
CONST _target_chip = cexpr
```

The compiler will look for a constant named 'PIC_chipname' and assign it to _target_chip. This might be used by some libraries to modify their behavior based on the chip in use. The compiler itself does not use this information.

## 1.2. TARGET CLOCK

Format:

```
PRAGMA TARGET CLOCK cexpr
```

Analogous to:

```
CONST _target_clock = cexpr
```

Set the target clock rate to *cexpr* in Hz. The compiler only needs this if the _usec_delay statement is used.

## 1.3. TARGET FUSES

Format:

```
PRAGMA TARGET FUSES [ cexpr0 ] cexpr
```

Analogous to:

```
CONST _config = cexpr,
```

```
or CONST _config[cexpr0] = cexpr
```

*cexpr*0 is only used when multiple config words exist in which case 0 is the first config word, 1 the second, and so on.

The configuration words define how some parts of the destination chip are used. While it is possible to set the fuses directly, it is generally better to use the "TARGET opt tags" construct below.

# 1.4. TARGET opt tags

Format:

```
PRAGMA TARGET opt tags
```

This accesses the mneumonic symbols defined with PRAGMA FUSE_DEF. This is preferable to setting the fuse values directly, because (1) mneumonics are more easily readable than numeric values, and (2) the same mneumonic can be setup differently on different chips.

# Chapter 2. Compiler Configuration

Configure compiler code generation policies.

## 2.1. BOOTLOADER

Format:

```
PRAGMA BOOTLOADER { BLOADER | LONG_START | LOADER18 [ cexpr ] | RICKPIC}
```

Set the user code preamble as follows:

### 2.1.1. BLOADER

Pre-amble:

```
ORG 0x0003
GOTO _pic_pre_user
```

### 2.1.2. LOADER18 [ *cexpr* ]

Pre-amble:

```
ORG cexpr -- (or 0x0800 if cexpr is not present)
```

The interrupt vector, if used, is put at *cexpr* + 8

### 2.1.3. LONG_START

Pre-amble:

```
ORG 0x0000
BSF/BCF _pclath, 4
BSF/BCF _pclath, 3
GOTO _pic_pre_user
NOP
```

### 2.1.4. RICKPIC

Pre-amble:

```
ORG  0x0003
GOTO _pic_pre_user
```

nb: if "PRAGMA INTERRUPT RAW" is used, the interrupt routine must not exceed one page (minus a few bytes).

## 2.2. CLEAR

Format:

```
PRAGMA CLEAR { YES | NO | }
```

YES -- Code is generated to set all user-data to 0
* NO -- No such code is generated

nb: volatile variables, and variables explicitly placed by the user are *not* set to 0

## 2.3. EEDATA

Format:

```
PRAGMA EEDATA expr;[','expr;1...]
```

Places data into the EEPROM (defined with PRAGMA EEPROM..). The first time this statement is executed, the data are placed into location 0 of the EEPROM. Each time after the data are placed in consecutive locations.

## 2.4. FUSE

Format:

```
PRAGMA FUSES { YES | NO | }
```

* YES -- The '__config' line is written to the assembly file
NO -- The '__config' line is not written to the assembly file

It is often convenient *not* to program the CONFIG word (for example, when using a boot loader). This suppresses that programming.

## 2.5. IDDATA

Format:

```
PRAGMA IDDATA expr;[','expr;1...]
```

Places data into the ID area (defined with PRAGMA ID...). The first time this statement is executed, the data are placed into location 0 of the ID. Each time after the data are placed in consecutive locations.

## 2.6. TASK

Format:

```
PRAGMA TASK cexpr
```

Set the maximum concurrent task count to *cexpr*. It is generally better to set this in your program, instead of doing so with the compiler option ("-tasks") as it is unlikely to change.

nb: When multiple tasks are used, the main task requires one task slot.

# Chapter 3. Global Configuration

These can all be used globally.

## 3.1. ERROR

Format:

```
PRAGMA ERROR
```

Causes an error to be generated. This has been superceded with the "_ERROR" command (see the JALv2 reference guide).

## 3.2. NAME

Format:

```
PRAGMA NAME str
```

Causes an error to be generated if the current source file name (excluding the .jal extension and path) doesn't match str.

## 3.3. SIZE

Format:

```
PRAGMA SIZE
```

Anything physically following this pragma will be optimized for size. See notes under 'PRAGMA SPEED'

## 3.4. SPEED

Format:

```
PRAGMA SPEED
```

Anything physically following this pragma will be optimized for speed. Currently this only affects the generation of the shift operators -- loops will be unrolled for SPEED, but used for SIZE. In the future this could affect other loop unrolling.

# Chapter 4. PROCEDURE/FUNCTION Configuration

These must be used with a procedure and/or function and are only in effect for the procedure/function in which they're used.

## 4.1. FRAME

Format:

```
PRAGMA FRAME
```

Used within a function or procedure, declares that all variables in the function or procedure will be allocated into a single 'frame'. This guarantees that all local variables will be allocated in the same data bank, so bank switching to access variables will be minimized. This can also result in 'out of data space' errors due to memory fragmentation when plenty of space is otherwise available.

Normally variables are allocated at the lowest address into which they will fit. This makes much better use of the memory, but can cause variables in the same function to be allocated in separate banks which results in bank switching overhead.

nb: Any re-entrant function, and any function called through a function pointer (aka, pseudo-variable function) will allocate per-frame regardless of this setting.

## 4.2. INLINE

Format:

```
PRAGMA INLINE
```

Used within a function or procedure, declares that this function or procedure will not get a seperate body, but rather will be copied directly into the calling code.

nb: If a procedure or function marked 'inline' is executed as volatile parameter, it will get a body.

# 4.3. INTERRUPT

Format:

```
PRAGMA INTERRUPT { FAST | RAW | NORMAL | }
```

This must be used within a procedure that takes no parameters. It defines the procedure as an interrupt entry point. Such a procedure is added to a chain of procedures executed whenever an interrupt hits. Once a procedure has been marked interrupt, it cannot be called directly by the program.

## 4.3.1. * NORMAL

W, STATUS, PCLATH, FSR, TBLPTR and _picstate are saved on ISR entry and restored on exit

## 4.3.2. FAST

_pic_state is *not* saved or restored. In this case, the interrupt procedure should be written entirely in assembly to avoid corrupting the pic_state area.

## 4.3.3. RAW

*None* of the normal pre-amble or cleanup code is generated. The user is entirely responsible for the interrupt. The code is guarenteed to placed at the interrupt vector.

nb: this feature isn't yet available

# 4.4. JUMP_TABLE

Format:

```
PRAGMA JUMP_TABLE
```

This generates a warning, but does nothing. It's here for backward compatability.

## 4.5. KEEP

Format:

```
PRAGMA KEEP { BANK | PAGE } [ "," { BANK | PAGE } ... ]
```

This guarentees the page and/or bank select bit manipulations will not be removed. Normally, they are removed if analysis shows them to be unnecessary. This is only useful to guarentee certain timings. This effects the entire procedure or function in which it is declared (not just from point of declaration).

## 4.6. NOSTACK

Format:

```
PRAGMA NOSTACK
```

Used within a procedure, the procedure will not be called using the normal call/return instructions. Instead, the return address will be stored in a procedure-local variable, '_return,' and the call will be executed by jumping to the start of the procedure. The return will be executed by jumping to the first statement after the call.

The overhead for this is two or three data bytes, four to six instructions for the return, and eight to ten instructions for the call. Currently re-entrant functions and functions called indirectly cannot use this pragma.

# Chapter 5. Optimization PRAGMAs

These all effect various optimizations done by the compiler. These cannot be used to turn on and off optimizations for specific parts of the code -- the last one parsed will be the one in effect.

For a complete description of each, including warnings and caveats, please refer to the Optimizations chapter of the Jalv2 Compiler Options document

## 5.1. EXPR_REDUCE

Format:

```
PRAGMA OPT EXPR_REDUCE { YES | NO | }
```

* YES --expression reduction is performed
NO -- expression reduction is not performed

Expression reduction looks for things like ‘x * 1’ and replaces with ‘x’. See ‘EXPRESSION REDUCTION’ in jalopts.txt for details.

## 5.2. CEXPR_REDUCE

Format:

```
PRAGMA OPT CEXPR_REDUCE { YES | NO | }
```

* YES -- constant expression reduction is performed
NO -- constant expression reduction is no performed

Constant expression reduction looks for operations on two constants that can be evaluated at compile time, saving both time and memory.

nb: disabling this will cause the backend code generators to fail, so only do so if ‘PRAGMA DEBUG CODEGEN OFF’ is specified.

## 5.3. CONST_DETECT

Format:

```
PRAGMA OPT CONST_DETECT { YES | NO | }
```

YES -- enable constant detection
* NO -- disable constant detection

Look for variables that are defined but are either only assigned once, or are always assigned the same value. When this happens, replace all occurances of the variable with the constant.

nb : PRAGMA CLEAR will prevent this option from having any effect unless the variable is only assigned the constant 0.

## 5.4. LOAD_REDUCE

Format:

```
PRAGMA OPT LOAD_REDUCE { YES | NO | }
```

YES -- Perform load reduction
* NO -- Do not perform load reduction

Load reduction tracks the value in W and attempts to remove any load into W of a value it already holds.

nb: this is still considered experimental!

## 5.5. TEMP_REDUCE

Format:

```
PRAGMA OPT TEMP_REDUCE { YES | NO | }
```

YES -- Perform temporary variable reduction
* NO -- Do not perform temporary variable reduction

Temporary reduction effects complex instructions. For example, without temporary reduction the expression:

```
a = b + c * d + e
```

will use three temporary variables. With reduction, it will only use one.

## 5.6. VARIABLE_FRAME

Format:

```
PRAGMA OPT VARIABLE_FRAME { YES | NO | }
```

YES -- allocate variables a frame at a time
* NO -- allocate variables individually

Normally, variables are allocated individually. This allows optimal use of data memory, but means that variables in a given procedure might be spread across multiple banks. Enabling this option will guarentee that all variables in a procedure will reside in a single bank.

nb: unlike ‘PRAGMA FRAME’ above, this affects the entire file

## 5.7. VARIABLE_REUSE

Format:

```
PRAGMA OPT VARIABLE_REDUCE { YES | NO | }
```

* YES -- Perform variable reduction
NO -- Do not perform variable reduction

Variable reduction looks for variables that are assigned, but not used, or used, but not assigned, or neither used nor assigned. In these cases normally the variable is removed (unless it is volatile). Turning this off leaves the variable around.

# Chapter 6. Warning

Warning pragmas are in effect until changed (they can be turned on and off at will).

## 6.1. ALL

Format:

```
PRAGMA WARN ALL { YES | NO }
```

* YES -- enable all warnings
NO -- disable all warnings

## 6.2. BACKEND

Format:

```
PRAGMA WARN BACKEND { YES | NO }
```

* YES -- enable all warnings
NO -- disable all warnings

This turns on debugging of the code generator (currently the translation from pcode --> PIC).

## 6.3. CONVERSION

Format:

```
PRAGMA WARN CONVERSION { YES | NO | }
```

* YES -- enable conversion warnings
NO -- disable conversion warnings

Conversion warnings occur when assigning unsigned to signed, or signed to unsigned.

## 6.4. DIRECTIVES

Format:

```
PRAGMA WARN DIRECTIVES { YES | NO | }
```

YES -- enable directive warnings
* NO -- disable directive warnings

The JAL language has a peculiar feature : the construct:

```
IF cexpr THEN ... END IF
```

is actually a compiler directive. If *cexpr* evaluates to 0, the compiler stops translating the code until it reaches the corresponding END IF. This warning will simply shows where this construct is used.

## 6.5. MISC

```
PRAGMA WARN MISC { YES | NO | }
```

* YES -- enable misc. warnings
NO -- disable misc. warning

There are some warnings that are not catagorized. This enables or disables them.

## 6.6. RANGE

Format:

```
PRAGMA WARN RANGE { YES | NO | }
```

* YES -- enable out of range warnings
NO -- disable out of range warnings

This enables or disabled ‘value out of range’ warnings.

## 6.7. STACK_OVERFLOW

Format:

```
PRAGMA WARN STACK_OVERFLOW { YES | NO | }
```

YES -- stack overflow results in a warning
* NO -- stack overflow results in an error

## 6.8. TRUNCATE

Format:

```
PRAGMA WARN TRUNCATE { YES | NO | }
```

* YES -- enable the truncation warning
NO -- disable the truncation warning

Truncation can occur when a larger sized value is assigned to smaller one.

# Chapter 7. Chip Specification

These are used in the chip definition files, not normally by the end user.

## 7.1. CODE

Format:

```
PRAGMA CODE cexpr
```

Defines the code size for a device -- used to detect code too large. For the 12 & 14 bit cores, this number is in WORDs, for the 16 bit cores, this number is in BYTEs. Blame MicroChip.

## 7.2. DATA

Format:

```
PRAGMA [DATA | SHARED] cexpr0['-'cexpr1[',' ...]
```

Defines a range allowed when allocating variables. DATA access is assumed to require whatever banking method is needed for the target, whereas SHARED is assumed to not require these bits.

## 7.3. EEPROM

Format:

```
PRAGMA EEPROM cexpr0 ',' cexpr1
```

Defines EEPROM available to the chip. *cexpr*0 is the ORG used when programming the device, size is the EEPROM *cexpr*1 in bytes.

## 7.4. FUSE_DEF

Format:

```
PRAGMA FUSE_DEF opt[':'cexpr0] cexprm '{'
  tag '=' cexprb
```

```
  ...
'}'
```

Defines symbolic fuse bits so the end user needn't twiddle them directly.

opt -- a string presented to the user
[:*cexpr*0] -- which config word stores this entry, starting with 0
*cexpr*m -- the fuse word is bit-wise ANDed with this before continuing
tag -- the sub-tag
*cexpr*b -- which bit to set

These are used by the end user with the 'PRAGMA TARGET opt tags' defined above. In this case the result is similar to:

```
_config = _config & cexprm | cexprb
```

## 7.5. ID

Format:

```
PRAGMA ID cexpr0 ',' cexpr1
```

Defines ID bytes available to the chip. *cexpr*0 is the ORG used when programming the device, size is the EEPROM *cexpr*1 in bytes.

## 7.6. STACK

Format:

```
PRAGMA STACK cexpr
```

Defines the hardware stack size for a device -- used to detect stack overflow.

## 7.7. TARGET CPU

Format:

```
PRAGMA TARGET CPU cexpr
```

Analagous to:

```
CONST _target_cpu = cexpr
```

Set the target CPU. *cexpr* should be one of the constants from come from chipdef.jal.

# 7.8. TARGET BANK

Format:

```
PRAGMA TARGET BANK cexpr
```

Analagous to:

```
CONST target_bank_size = cexpr
```

Set the target's data bank size. The default is 128 bytes.

# 7.9. TARGET PAGE

Format:

```
PRAGMA TARGET PAGE cexpr
```

Analgous to:

```
CONST _target_page_size = cexpr
```

Set the target's code page size. The default is 2048 words. This is not used in the 16 bit cores.

# Chapter 8. Debugging

The following are only useful when debugging compiler errors.

## 8.1. CODEGEN

Format:

```
PRAGMA DEBUG CODEGEN { YES | NO | }
```

* YES -- Enable the back_end code generation
NO -- Disable the back_end code generation

Allow the pcode to be generated without executing the PIC code generator.

## 8.2. PCODE

Format:

```
PRAGMA DEBUG PCODE { YES | NO | }
```

YES -- show the pcode in the asm file
* NO -- don't show the pcode in the asm file