JALv2 PRAGMAs

Contents

Chapter

1

Introduction

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 for JAL compiler version jalv25r4 or higher.

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.

Chapter

2

Chip Configuration

Select various attributes of a chip.

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.

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.

TARGET FUSES

Format:

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

Analogous to:

```
CONST _config = cexpr
```

Or:

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

cexpr0 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.

TARGET opt tags

Format:

```
        PRAGMA TARGET opt tags
```

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

Chapter

3

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:

BLOADER

Pre-amble:

```
ORG 0x0003
GOTO _pic_pre_user
```

LOADER18 [*cexpr*]

Pre-amble:

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

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

LONG_START

Pre-amble:

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

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).

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.

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.

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.

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.

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

4

Global Configuration

These can all be used globally.

ERROR

Format:

```
PRAGMA ERROR
```

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 This has been superseded with the "_ERROR" command (see the JALv2 reference guide).

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.

SIZE

Format:

```
PRAGMA SIZE
```

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

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

5

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.

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.

INLINE

Format:

```
        PRAGMA INLINE
```

Used within a function or procedure, declares that this function or procedure will not get a separate 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.

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.

* NORMAL

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

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.

RAW

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

nb: this feature isn't yet available

JUMP_TABLE

Format:

```
        PRAGMA JUMP_TABLE
```

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

KEEP

Format:

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

This guarantees 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 guarantee certain timings. This effects the entire procedure or function in which it is declared (not just from point of declaration).

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

6

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.

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.

CEXPR_REDUCE

Format:

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

* YES -- constant expression reduction is performed

NO -- constant expression reduction is not 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.

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 occurrences 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.

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!

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.

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 guarantee that all variables in a procedure will reside in a single bank.

nb: unlike `PRAGMA FRAME' above, this affects the entire file.

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

7

Warning

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

ALL

Format:

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

* YES -- enable all warnings

NO -- disable all warnings

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).

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.

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.

MISC

Format:

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

* YES -- enable misc. warnings

NO -- disable misc. warning

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

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.

STACK_OVERFLOW

Format:

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

YES -- stack overflow results in a warning

* NO -- stack overflow results in an error

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

8

Chip Specification

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

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.

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.

Note that automatic variable allocation only uses the area denoted by DATA, never the area denoted by SHARED, so these two can overlap.

EEPROM

Format:

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

Defines EEPROM available to the chip. *cexpr0* is the ORG used when programming the device, size is the EEPROM *cexpr1* in bytes.

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
>
> [:*cexpr0*] -- which config word stores this entry, starting with 0
>
> *cexprm* -- the fuse word is bit-wise ANDed with this before continuing
>
> tag -- the sub-tag
>
> *cexprb* -- 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
```

ID

Format:

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

Defines ID bytes available to the chip. *cexpr0* is the ORG used when programming the device, size is the ID *cexpr1* in bytes.

STACK

Format:

```
PRAGMA STACK cexpr
```

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

TARGET CPU

Format:

```
PRAGMA TARGET CPU cexpr
```

Analogous to:

```
CONST target_cpu = cexpr
```

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

TARGET BANK

Format:

```
PRAGMA TARGET BANK cexpr
```

Analogous to:

```
CONST target_bank_size = cexpr
```

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

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.

TARGET INST

Format:

```
        PRAGMA TARGET INST cexpr
```

Analogous to:

```
        CONST target_instruction_set = cexpr
```

Set the instruction set to be used by the compiler. This pragma is optional for most PICs but is needed for certain PICs with a variant on certain instruction sets used by MicroChip. This pragma will only be present in device files for which this is needed, otherwise it is omitted.

Chapter

9

Debugging

The following are only useful when debugging compiler errors.

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.

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