MPLAB™ C30 Managed PSV Pointers

Beta support included with MPLAB C30 V3.00

Contents

1 Overview

The dsPIC30F/33F and PIC24F/H families of processors contain hardware support for accessing data from within program FLASH. Previous versions of MPLAB C30 only supported direct access via one single PSV page. As the 16-bit parts are getting larger program memories, customers will need to easily access many pages. This feature provides better support in this area.

The extensions presented here are beyond the scope of C89, though the method chosen to implement them is within the scope of C99.

1.1 Why Beta?

We hope to provide a feature that will benefit our customers; internal consultation has come up with a set of features described within this document. By marking the feature as Beta we acknowledge that this design may be incomplete, and leave room for further customer directed enhancement.

1.2 Other Sources of Reference

Other concepts used in this document can be found in manuals that reside in the *doc* folder that may have been installed with the product, or online at `http://www.microchip.com/c30`.

Doc Id	Title
DS51284	MPLAB C30 C compiler User's Guide
DS51317	MPLAB ASM30 MPLAB LINK30 and Utilities User's Guide

2 Current feature support

Today MPLAB C30 allows the programmer to easily create data in program memory using the following features:

- the compiler supports a variable attribute, `space()` which allows the user to nominate:

  `psv` to place the variable into program memory in suitable chunks for PSV access to be managed by the user

  `auto_psv` to place the variable into program memory in suitable chunks for PSV access, to be managed by the compiler (currently only 1 `PSVPAG` setting)

  `eedata` to place the variable into EEDATA memory, access is completely defined by the user

  `prog` to place the variable into unrestricted program memory, access is completely defined by the user

among other spaces. For a complete description of the `space` attribute, please refer to the MPLAB C30 User's Guide.

- the compiler will place (by default) all `const` qualified variables into the `auto_psv` space; this may be disabled by using the command line switch `-mconst-in-data`
- MPLAB C30 will enable and support the use of 1 PSV page

Using the current features, the programmer can create data and access data in several different ways:

- Variables stored in compiler managed PSV section

```
const char *foo = "a constant string";
int fibonacci[4]
   __attribute__((space(auto_psv))) = { 1, 1, 2, 3 };

void bar() {
  fprintf(stderr,"%s: %d %d\n", foo, fibonacci[1],
                                fibonacci[3]);
}
```

  When compiled without specifying the constants-in-data memory model both `foo` and `fibonacci` will be placed into the (same) compiler managed PSV section. If this section exceeds 32K in size, the linker will complain and the application will fail to link. The tool chain will arrange for the `PSVPAG` to be correctly set at program start-up.

  Function `bar()` requires no special treatment to access the variables defined in this way. The tool chain assumes that the `PSVPAG` has not changed.

- Variables stored in a user managed PSV section

```
char *foo
   __attribute__((space(psv))) = "a constant string";
int fibonacci[4]
   __attribute__((space(psv))) = { 1, 1, 2, 3 };

void bar() {
  int a,b;

  CORCONbits.PSV = 1;
  PSVPAG = __builtin_psvpage(fibonacci);
  a = fibonacci[1];
  b = fibonacci[3];
  PSVPAG = __builtin_psvpage(foo);
```

```
  fprintf(stderr,"%s: %d %d\n", foo, a, b);
}
```

When compiled **without** specifying the constants-in-data memory model this program will fail at run-time. That is because the literal string in the call to `fprintf` will occupy the `auto_psv` space and the `PSVPAG` setting has been changed. The other things to note are:

- the user must ensure that the PSV window is enabled
- only one `PSVPAG` is available at a time, so the user must cache some of the values before the call to `fprintf`
- it really isn't hard, just annoying

Note, that this example can apply to data stored in EEDATA too as the hardware allows the EEDATA memory to be accessed via the PSV window.

- Variables stored in a general program section

```
char *foo
   __attribute__((space(prog))) = "a constant string";
int fibonacci[4]
   __attribute__((space(prog))) = { 1, 1, 2, 3 };

void bar() {
  /* eek! */
}
```

Well, I have been slightly lazy and omitted the code for `bar` to show how to access this kind of memory. Mostly to prove a point. There are two choices, using PSV or using the `TBLRD`[1] machine instructions. In this case, both are just as hard because the tool chain has not ensured that either variable doesn't cross a PSV boundary and table operations are hard to perform in C.

3 Goals

Ideally a programmer should be able to make use of variables, where ever they are, without needing to configure hardware resources. In general this would require the language to support a new kind of pointer, one that represents a complete 24-bit address in program FLASH. In the ideal world, our user managed PSV example could be coded in the following manner:

[1] V3.00 contains built-in definitions to make this easier

```
char *foo
   __attribute__((space(psv))) = "this is a constant string";
int fibonacci[4]
   __attribute__((space(psv))) = { 1, 1, 2, 3 };

void foo() {
  fprintf(stderr,"%s: %d %d\n", foo, a, b);
}
```

I don't think this ideal will be reached (unfortunately) but we can come very close. Incidentally, the reason why I don't think this will happen is solely due to they way library functions like `fprintf` are implemented, but it shows a general problem that is likely to affect many users.

In this case, when `fprintf` processes the `%s` formatting character, it expects the string to be located in data memory. It is not, so the function will fail.

There are other, architectural, caveats that prevent us from reaching the ideal. In the following sections, I hope, the reader will observe that our extensions meet the spirit of this ideal goal. Functionality that we provide includes:

- ability to easily create data in program FLASH memory
- ability to access the data without having to think about the hardware limitations involved with the access

and is discussed in the following sections.

4 Managed PSV pointers - 24-bit pointers

Managed PSV pointers allow the compiler set keep track of the full 24-bit address of an object. When a managed PSV pointer is accessed, the compiler knows that the `PSVPAG` must be set before access. This operation is most certainly not atomic, so the compiler's view of what must be saved in an ISR has changed. ISR routines that use the `PSVPAG` resource will be required to set it first (an incompatibility with old objects).

We will support two kinds of managed pointer, a small pointer where the address is constrained to single PSV page and a large pointer where addresses may overlap the page boundary. The small pointer, `__psv__`, can be used to create a pointer to data stored in `space(psv)`. The large pointer, `__prog__`, is more general and should be used when the data *may be* stored in `space(prog)`. Large pointers are much more general.

A managed pointer is identified by a new syntax feature which resembles a *cv-qualifier* like `const` or `volatile`. Note that using this new pseudo-qualifier does not imply that the storage is in a PSV range. It is possible, for example, to define pointers to PSV objects where the pointer itself lives in normal data memory. Some examples:

`int foo __attribute__((space(psv)));`	`foo` lives in FLASH; it is not a managed variable - just like current syntax
`__psv__ int foo __attribute__((space(psv)));`	`foo` lives in FLASH; it is a managed variable, the programmer can accesses it as if it were a normal C variable
`int __psv__ *bar = &foo;`	`bar` lives in RAM; it is a managed pointer to an int (like other *cv-qualifiers*, read out from the pointer until the type is complete); assignments to `bar` should be addresses of variables in FLASH and can be dereferenced with normal C syntax - the compiler will do the hard work. Pointer arithmetic on a `__psv__` pointer will fail if the programmer tries to overflow the PSV page.
`int big_foo[256] __attribute__((space(prog))) = { ... };`	`big_foo` is an array that lives in FLASH, but it is not constrained by PSV address boundaries like a `space(psv)` would be.
`int __prog__ *big_bar = big_foo;`	like `bar`, `big_bar` is a managed pointer to an int. This pointer is not constrained by a PSV page boundary; any arithmetic performed will also update the page. A `__prog__` pointer can point to something placed into a `space(psv))` constrained section.

4.1 Actual concrete examples

Caveat: code-generation is supplied for illustrative purposes only.

4.1.1 Example 1: Managed Pointers as Function Arguments

In the following function:

```
int eek(const char *bar, __prog__ int *foo, __prog__ int *foo2) {

   int i = *foo2 + *foo;
   baz();
   return i + *foo;

}
```

`foo` and `foo2` are pointers to an int in program space. The compiler will generate a sequence of code that preserves the PSVPAG and determine the correct PSV page and offset to access each pointer. For example:

```
_eek:
        mov.d   w8,[w15++]
        mov     w10,[w15++]
        mov.d   w2,w8
        mov.d   w4,w2
        mov     _PSVPAG, w0     ; preserve current _PSVPAG
        mov     w5, _PSVPAG     ;   PSVPAG for foo
        rrnc    w2, w2          ;   create *real* offset for foo
        mov     [w2],w1         ;   dereference foo
        mov     w0, _PSVPAG     ; restore _PSVPAG
        mov.d   w8,w2
        mov     _PSVPAG, w4     ; preserve _PSVPAG
        mov     w9, _PSVPAG     ;   PSVPAG for foo2
        rrnc    w2, w2          ;   create *real* offset for foo2
        mov     [w2],w0         ;   dereference foo2
        mov     w4, _PSVPAG     ; restore _PSVPAG
        add     w1,w0,w10       ; i = *foo2 + *foo
        rcall   _baz
        mov.d   w8,w2
        mov     _PSVPAG, w4     ; preserve _PSVPAG
        mov     w9, _PSVPAG     ;   PSVPAG for foo
        rrnc    w2, w2          ;   create *real* offset for foo2
        mov     [w2],w0         ;   derefence foo2
        mov     w4, _PSVPAG     ; restore _PSVPAG
        add     w10,w0,w0       ; return i + *foo;
        mov     [--w15],w10
        mov.d   [--w15],w8
        return
```

One might call this function by:

```
int foo_value, foo2_value __attribute__((space(prog)));

main() {
  eek("test", &foo_value, &foo2_value);
}
```

The assembly for the function call looks like:

```
mov     #psvpage(_foo_value),w3
mov     #psvptr(_foo_value),w2
mov     #psvpage(_foo2_value),w5
mov     #psvptr(_foo2_value),w4
mov     #.LC0,w0
rcall   _eek
```

4.1.2 Example 2: Managed Pointer (to an Array) as Function Arguments

In the function below, `foo` points to a null-terminated buffer somewhere in FLASH.

```
int eek(const char *bar, __prog__ int *foo) {
   int sum = 0;

   while (*foo) sum += *foo++;
   return sum;

}

int foo_buffer[256] __attribute__((space(prog)));

main() {
   eek("test", foo_buffer);
}
```

4.1.3 Example 3: managed array of managed pointers

`ppi_array` is an array of pointers to ints; the pointers are themselves managed (24-bit) pointers. I used the `typedef` because it makes the definition easier to read!

```
typedef int __prog__ *prog_int;  /* pointer int in space(prog) */

/* ppi_array is an array of 16 managed pointers to ints;
   The array itself is in a psv space, and we would like the compiler
   to access the array using a managed access */
__prog__ prog_int ppi_array[16] __attribute__((space(psv)));

int main() {

  int sum = 0;
  int i;
  int __psv__ *p;

  for (i = 0; i < 15; i++) {
    p = ppi_array[i];
    sum += *p;
  }
  return sum;
}
```

4.1.4 Example 4: Accessing FLASH Variables Directly

Although we talk about this new feature as 24-bit pointers, it really extends beyond that. It is possible to identify a variable as being in FLASH, and accessing them without the need to use a pointer.

```
#include <stdio.h>

__psv__ int foo __attribute__((space(psv))) = 1;
__psv__ float bar __attribute__((space(psv))) = 3.14159;
__psv__ int eek[256] __attribute__((space(psv))) = {
    200,201,202
  };

main() {

  fprintf(stderr,"foo    (@ 0x%6.6lx) is: %d\n", &foo, foo);
  fprintf(stderr,"bar    (@ 0x%6.6lx) is: %f\n", &bar, bar);
  fprintf(stderr,"eek[2] (@ 0x%6.6lx) is: %d\n", &eek[2], eek[2]);

}
```

Running this under the simulator might yield the following to stdout:

```
foo    (@ 0x004459) is: 1
bar    (@ 0x004451) is: 3.141590
eek[2] (@ 0x004059) is: 202
```

The correct answers, but those addresses do look odd! In order to make certain operations easier, the compiler represents the PSV addresses in a non-standard way. If you'd like to decompose them, the format of a 24-bit pointer (consuming 32-bits) is:

bits 31 ... 24	23 ... 16	15 ... 1	0
padding 8 bits	PSV page 8 bits	PSV offset bits 0 to 14	PSV offset bit 15

5 Managed PSV Pointers - ISR Considerations

A data access using 24-bit pointers is *definitely* non-atomic. Furthermore an interrupt service routine never really knows what the current state of the `PSVPAG` register will be. Unfortunately the compiler is not really in any position to determine whether or not this is important in all cases. The compiler will make the simplifying assumption that the writer of the interrupt service routine will know whether or not the the auto, compiler managed, `PSVPAG` is required by the ISR (or any functions called by the ISR) and will request that the programmer specify whether or not it is necessary to assert the default setting of the the

PSVPAG register. This is achieved by adding a further *attribute* to the interrupt function definition:

auto_psv	the compiler will set the PSVPAG register to the correct value for auto_psv or const (by default) variables, and restore the PSVPAG before returning
no_auto_psv	the compiler will not set the *PSVPAG* register

For example:

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void) {
  IFS0bits.T1IF = 0;
}
```

Current code (that does not assert this) may not execute properly unless recompiled. When recompiled, if no indication is made, the compiler will generate a warning message and select the auto_psv model. The choice is provided so that programmers that are especially conscious of interrupt latency may select the best option for them. Saving and setting the PSVPAG will consume approximately 3 cycles at the entry to the function and one further cycle to restore the setting upon exist from the function. Note that boot or secure interrupt service routines will use a different setting of the PSVPAG register for their constant data from interrupt routines in a general FLASH area.

6 Support information

Questions, comments or suggestions can be sent to Microchip in the usual ways: via your FSE/FAE representive, directly through support@microchip.com.