PUMPKIN™
REAL-TIME SOFTWARE

AN-3
Application Note

750 Naples Street • San Francisco, CA 94112 • (415) 584-6360 • http://www.pumpkininc.com

Salvo, Banked Objects and the HI-TECH PICC Compiler

Introduction

One of the many attractions of Salvo, The RTOS that runs in tiny places™, is how easy it makes intertask communications, e.g. by enabling you to pass *messages* between tasks.

Instead of contiguous RAM, some processors have *banks* of RAM due to addressing restrictions or a small opcode size. Salvo messages use *message pointers*, which can point to anywhere in RAM or ROM.[1] To use messages, you need to be comfortable with *pointers* and *banked objects*.

This Application Note explains how to use Salvo message pointers with banked objects and the HI-TECH PICC compiler.

Note This Application Note references the Microchip PIC16C77 one-chip microcontroller. It has 4 banks of RAM, Bank 0 (0h-7Fh) through Bank 3 (180h-1FFh), each with up to[2] 80 bytes of RAM. Other banked processors may be similar.

PICC and the *bankn* Special Type Qualifier

With the exception of auto variables and parameters, you can place any object into any bank, assuming it will fit there. PICC's *special type qualifiers* for locating objects in a particular bank are of the form

```
bank1
bank2
bank3
```

for the desired RAM bank. Bank 0 is the default RAM bank, so to place something in Bank 0, no type qualifier is used.

created by Andrew E. Kalman on May 11, 2000 updated on May 20, 2001

Simple Banked Objects

Below are some simple declarations in C. First, here's a `long int` in Bank 0:

```
long int pos;
```

Here's an `int` in Bank 1:

```
bank1 int mem;
```

Here's an array of `static chars` (a string) in Bank 2:

```
static bank2 char strRc[SIZEOF_STR_RESP+1] = "\0";
```

And here are some `const chars` in ROM:

```
const char row30[6] = { 14, 13, 11, 7,  15, 15 };
```

These examples are easily understood, and once declared with the proper type qualifiers, you can access an object without worrying which RAM bank it's located in.

You can use C's `typedef` to make your code easier to read and more robust. For example,

```
typedef bank1 char bank1char;
```

defines a type `bank1char` of `char` objects in Bank 1. Declaring

```
bank1char temp1, temp2, temp3, temp4;
```

will place four `char` variables named `temp1-temp4` in Bank 1. You can now use `bank1char` throughout your code when declaring `char` variables in Bank 1. If you choose to move all of those variables to another bank, then changing the `bank1` type qualifier in the `typedef` is all that is necessary.

Pointers, Banked Pointers and Pointers to Banked Objects

Learning to use pointers with banked RAM may take a bit longer. Here's a banked pointer to a `char`, i.e. the pointer is located in Bank 2, but the `char` it points to is located in Bank 0:

```
char * bank2 charP;
```

Here's a (unbanked) pointer to a banked char, i.e. the pointer is located in Bank 0, but the `char` it points to is in Bank 1:

```
bank1 char * charP;
```

This is the same thing:

```
char bank1 * charP;
```

Here's a banked pointer to a banked char, i.e. the pointer is in Bank 1 and the `char` is in Bank 2:

```
bank2 char * bank1 charP;
```

Lastly, here's a pointer to a pointer to a `char`, all in separate banks:

```
bank2 char * bank1 charP * bank3 charPP;
```

Passing Banked Objects as Parameters

You can always pass a banked object *by value*, e.g.:

```
void MyFunction( int parm1 )
{
      parm1++;
}
```

and

```
MyFunction(mem);
```

where `mem` is as declared as a `bank1 int` will work correctly.[3] But if you want to pass the object *by reference*, and the object is banked, you must declare the pointer parameter with the proper special type qualifier, e.g.:

```
void MyFunction( bank1 int * parm1 )
{
      *parm1++;
}
```

and

```
MyFunction(&mem);
```

If you fail to declare the pointer parameter properly, your function will operate on an object with the same address (modulo 80h) but in a different bank – Bank 0 if no type qualifier is present. In the above example, if the linker places `mem` at B1h and `bank1` is left out of `MyFunction()`'s parameter declaration, then the function will increment the two-byte value starting at (B1h-80h) in RAM

Bank 0, or 31h. A mistake like this will cause unpredictable behavior in your program and must be avoided.

Salvo's Message Pointers

Suppose you're using a Salvo *message queue* to communicate between two tasks. You have an array in memory, e.g.:

```
bank1 char myArray[6];
```

that contains one-character commands. You pass those commands, one at a time, via a message queue, to another task:

```
OSSignalMsgQ(MSGQ1, (OStypeMsgP) &myArray[i]);
```

Each element of the message queue is a Salvo *message pointer* of type `OStypeMsgP`, usually predefined as `void *`, i.e. a pointer to anything. The power of using message pointers becomes apparent when you realize that there are no restrictions on what a message pointer can point to. It can point to a `char`, an `int`, a `const`, a structure, another pointer, a function, etc. As long as both parties agree on what a particular message points to, the information will pass correctly from sender to receiver.

In the example above, the messages in the message queue are pointers to an array of `char` in Bank 1. The `(OStypeMsgP)` *typecast* is used in `OSSignalMsgQ()` to convert `&myArray[i]`, which is a pointer to a `char` in Bank 1, into a message pointer. When another task receives the message, it will have to convert (via another typecast[4]) the pointer back to the appropriate type before *dereferencing* it:

```
void TaskRcv ( void )
{
      char cmd;
      OStypeMsgP msgP;


      for (;;)
      {
         OS_WaitMsgQ(MSGQ1, &msgP, TaskRcv2);
         cmd = * (char *) msgP; /* wrong! */
…
```

Sadly, the typecast above is not entirely correct. That's because we're asking the PICC compiler to convert a message pointer to a `char` pointer (i.e. a pointer to a `char` in Bank 0), when what we really want is a `bank1 char` pointer! The correct line is:

```
cmd = * (bank1 char *) msgP;
```

We could have avoided this confusion by defining:

```
typedef bank1 char myBank1Array;
```

by declaring:

```
myBank1Array myArray[6];
```

and by writing:

```
cmd = * (myBank1Array *) msgP;
```

Conclusion

If you run out of Bank 0 RAM in your application you'll need PICC's special type qualifiers to locate objects in other RAM banks. If you use pointers to access those objects, you need to pay close attention to declarations and typecasts to ensure that your pointers are pointing to what you think they're pointing to. Using `typedef` can help you avoid certain common mistakes.

---

1 On some processors a Salvo configuration option may need to be used for message pointers to point to RAM and ROM. On these processors, the default is to point to RAM only.

2 Some locations are dedicated file registers, mirrored in other banks or simply not available.

3 `bank1` does not appear in the parameter declaration because PICC places all parameters are in Bank 0.

4 Typecasting is a compile-time, not a real-time operation. Therefore it has no effect on run-time performance per se.