PUMPKIN™
REAL-TIME SOFTWARE

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

Salvo Messages, Memory Models and Keil's Cx51 C 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.

The 8051 and its derivatives support a wide range of memory areas, including internal and external data memory. Through the use of language extensions called *memory types*, Keil's Cx51 compiler enables you as the programmer to place variables in these data memory areas. Salvo messages use *message pointers*, which can point to anywhere in RAM or ROM. Therefore to use messages, you must be comfortable with *pointers* and Cx51's memory types.

This Application Note explains how to use Salvo message pointers with Cx51's various memory models.

Default Memory Types

When building an application with Cx51, you must specify the *memory model* to be used. `SMALL`, `COMPACT` and `LARGE` are the choices.[1] The memory model affects the location of each declared variable, function argument and automatic variable unless its memory type (see *Cx51's Explicitly Declared Memory Types*, below) has been explicitly specified. Cx51's *default memory types* are listed in Table 1:

created by Andrew E. Kalman on Apr 24, 2002 updated on Jul 28, 2003

Memory Model	Default Memory Type	RAM Memory Area Used	Accessed via
SMALL	data	internal, 0x00-0x7F	direct
COMPACT	pdata	external, 0xXX00-0xXXFF[2]	MOVX @Rn[3]
LARGE	xdata	external	MOVX @DPTR

Table 1: Default Memory Types for Selected Memory Model

Cx51's Explicitly Declared Memory Types

As illustrated in Table 1, changing the memory model alters the default memory type, and thereby the location of any objects (declared variables, function arguments and/or automatic variables) lacking an explicitly declared memory type. This affects the use of pointers and their proper declaration.

In all of Cx51's memory models, the default memory type can be overridden on a per-object basis by explicitly declaring the object's memory type (and hence its location) using one of Cx51's `code`, `data`, `idata`, `bdata`, `xdata`, `far` or `pdata` memory types.

Simply Typed Objects

Below are some simple variable declarations in C. First, here's a `long int` located in directly accessible internal data memory:

```
long int data pos;
```

Here's an `int` in indirectly accessible internal data memory:

```
int idata mem;
```

Here's an array of `static chars` (a string) in external data memory:

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

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

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

```
typedef char data TYPE_DATA;
```

defines a type `TYPE_DATA` of `char` objects in directly accessible internal data memory. Declaring

```
TYPE_DATA temp1, temp2, temp3, temp4;
```

will place four `char` variables named `temp1-temp4` in the first 128 bytes of the 8051's internal RAM. You can now use `TYPE_DATA` throughout your code when declaring `char` variables in directly accessible internal data memory. If you choose to move all of those variables to another memory area, then changing the `data` memory type in the `typedef` is all that is necessary.

Pointers, Explicitly Typed Pointers and Pointers to Explicitly Typed Objects

Learning to use pointers with the various memory types may require additional study. Here's an `idata` pointer to a `char data`. Both the pointer and the char are located in internal memory:

```
char data * idata charP;
```

Here's an `xdata` pointer to a `data` char, i.e. the pointer is located in external RAM, but the `char` it points to is in internal RAM:

```
char data * xdata charP;
```

This is the same thing:

```
data char * xdata charP;
```

Here's a pointer to a `char`, both of which are in external RAM:

```
char xdata * xdata charP;
```

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

```
char data * idata charP * xdata charPP;
```

Salvo's Message Pointers

Suppose you're using a Salvo *message queue* to communicate between two tasks. Assume you are using the `SMALL` memory model. You have an array in external memory, e.g.:

```
char xdata myArray[6];
```

that contains one-character commands.

Note The explicit `xdata` in the declaration of `myArray[]` overrides the `SMALL` memory model's default memory type of `data` for the variable `myArray[]`.

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 external memory. The `(OStypeMsgP)` *typecast* is used in `OSSignalMsgQ()` to convert `&myArray[i]`, which is a pointer to a `char` in external memory, 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 Cx51 compiler to convert a message pointer to a `char` pointer (i.e. a pointer to a `char` in internal memory),[5] when what we really want is a `char xdata` pointer! Why? Because `myArray[]` is located in external memory, and we need the Cx51 compiler to treat `msgP` as if it's a pointer to a `char` object in external memory before dereferencing it. The correct line is:

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

We could have avoided this confusion by defining:

```
typedef char xdata myBank1Array;
```

by declaring:

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

and by writing:

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

when dereferencing the message pointer.

Effect of Selecting a Different Memory Model

In example above, had we declared `myArray[6]` as `char` (no `xdata`), and we used the `SMALL` memory model, then the simple message pointer dereferencing of `cmd = * (char *) msgP` would have worked properly.

If we then changed to the `LARGE` model, `myArray[]` would be located in external memory, and the dereferencing would not work properly at runtime. That's because the change in memory model (`SMALL` to `LARGE`) resulted in a change in default memory type (`data` to `xdata`), and `myArray[]` – lacking an explicit memory type in its declaration – follows the default memory type of the memory model selected. Yet the typecast – `(char *)` – remained unchanged, still assuming that `myArray[]` was of type `char`, not `char xdata`.

Therefore we recommend explicit memory types and typedefs when any sort of pointer dereferencing is required so as to avoid any problems when changing memory models.

Conclusion

Your application's RAM objects will be located in the 8051's internal and/or external memory space based on the Cx51 memory model you select and any explicit memory types you employ. If you use pointers to access those objects, or if you use Salvo's messaging services, 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. Failing to explicitly declare memory types may lead to problems when switching between memory

models.[6] Using `typedef` can help you avoid certain common mistakes.

Acknowledgements

Dan Henry, for issues surrounding the `COMPACT` memory model, and others.

References

Keil Elektronik GmbH. and Keil Software, Inc., *Cx51 Compiler User's Guide*, 5.2001.

---

1 `SMALL` is the default and is recommended for most applications.

2 `XX` represents the most significant 8 bits of the 16-bit external address and is asserted by `P2`.

3 Where `n` is 0 or 1, i.e. `R0` or `R1` contains the lower 8 bits of the 16-bit address.

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

5 Internal memory / `data` space because of the `SMALL` model.

6 From a performance standpoint, it's best to always explicitly declare the memory types of pointers in C51.