C Support for Multiple Address Spaces in Embedded Systems

Walter Banks
Byte Craft Limited
421 King Street North
Waterloo, Ontario N2J 4E4

Voice : (519) 888 6911
Fax : (519) 746 6751
Email : walter@bytecraft.com

Introduction

The small-scale embedded systems implemented in a high-level language (most often a C dialect) are now being regularly employed in watches, microwave ovens, bathroom scales, children's toys, personal organizers, TV remote controls, and automotive applications. Embedded small-scale products have become very complex. Code quality from many compilers rivals that from the best hand coded assembly language. In other words, we as an industry can benefit from standardized features.

The traditional view of low-cost embedded microcomputers is that they have many different types of non-standard I/O hardware. By volume, 50% of embedded microcomputers are sold to volume manufacturers for less than $1 US. A study we did found remarkably few variations in function, only in silicon implementations. Features of embedded processors are so similar that as many as 90% of all low-end embedded system applications could be implemented on at least one member of the leading dozen or so microcomputers.

Applications that are developed for low-cost embedded systems are not particularly sensitive to development cost. Applications themselves are innovative and unique. The incentives for standardization are: time to market, reliable reusable code, product reliability and product liability. Standardization focuses the attention of the third-party software vendors to develop portable applications and libraries.

The typical small-scale embedded system is designed to be manufactured in large quantities, making production cost, reliability, code and system resource utilization important. The small-scale embedded system has between 0.5 and 256k of ROM, 32 to 32K bytes of RAM, often with a fast RAM component of up to 256 bytes. I/O is typically implemented in its own address space, or mapped on ROM or RAM address space. The

basic address spaces of most microcontrollers can be extended by using one of the serial bus protocols (I^2C, Microwire, SPI, USB, or CAN busses). It is worthy to note that in all of the embedded microcontrollers, there exist multiple address spaces that do not directly map in any single linear address map.

It is not a unique problem to use a C compiler to support application code on a processor where it is desirable to implement more than one size of pointer. The traditional implementation of this type of problem is to distinguish between different pointer types with tags (most commonly ***near*** and ***far***). Pointer size is tied to the tag, allowing application performance to be enhanced. References to ***near*** pointers carry with them implied address space information. Typically, pointers without tags are ***far*** pointers that can reach all of the address space.

A case for named address space.

Current practice for small-scale embedded systems is to create a large virtual address space, and map the actual processor on this space. This has been an ad hoc process whose functionally is similar (but implementation varies) between compiler vendors.

Reviewing for a moment the issues: small-scale embedded systems have multiple address spaces, often specifically related to the application. Traditional C compilers recognize that pointers may be either general or in some way context-specific. Unqualified pointers have application-wide scope. Extending this approach to small-scale embedded systems, we can create a virtual linear address space across the whole system, where each segment corresponds to a single memory area.

Segment	Local to Segment

Figure 1. Virtual address space created out of segments

A practical embedded system is likely to have multiple segments, organized along both logical and physical divisions.

ROM	ROM segment
FRAM	Fast Ram
RAM	RAM
IOspace	I/OSpace
XMEM	External Memory

Figure 2. Typical address spaces in small scale embedded system

Pointer declarations using named address space conventions can be declared by tying the segment name to the pointer, as the following example shows. Unnamed pointers are ***far*** and can be used to access any memory location.

```
int * FRAM ptr;        // Pointer to int in fast RAM
char * cptr;           // Global  pointer to char
```

Application developers can use named address space to control the memory area that variables can be allocated to. This feature of named address space is a particularly useful tool to group variables. The embedded systems application developer is given the freedom to choose the organization of the software.

```
char xmem ch;          // char placed in external memory
int dsp_x x[32];       // array of int's in DSP x memory
```

Named address space on execution space.

Traditional small embedded systems stored application executables in ROM, organized as a linear address space. The last ten years have seen slow changes that have eroded this simple linear address concept. It is useful to see how this has happened historically, and to try to predict the likely path that execution space is taking.

Initially, the application code was contained in a linear address space that could be completely addressed by the processor's program counter. Application system architecture generally separated the execution unit from the memory, resulting in consistent ROM access time. At this point, the path diverges into two supported architectures:

a) Single chip processors were developed, with ROM on board covering part of the allowable address space, and with provision for optional external ROM space.
b) Other processors had some form of memory management system grafted on the basic processor to accommodate applications that would no longer fit within the available address space. There are small differences in implementation; some processors allow direct switching from page to page, while others must switch by executing code from a common ROM area. From a language perspective, the switching mechanism is masked in the compiler implementation.

Many of the current small embedded systems have some component of executable space implemented in flash memory. Developers are seeing the advantage of using flash for field and feature upgrade capability for their products. Flash memory is also being used to maintain calibration information, as well as code storage.

Memory management in embedded systems is currently receiving a lot of attention, because it is one of the largest identifiable sources of application overhead (Our studies are showing that as much as 25% of the execution cycles in some systems are RAM and ROM memory management-related). It is a concern to some chip designers that flash memory access times are slower than the fastest RAM available. Some systems are being proposed and implemented that first load an execution RAM from flash on power-up, to improve system execution performance. The clear next step is to load an application from flash that is configured to support the specific requirements, and, if possible, avoid some of the ROM memory management overhead. This approach is being extended even further by having multiple processing units, each loaded with a part of the application code. Execution is still "one processor execution at a time" (common data RAM), but ROM memory management is reduced to calls between processors.

The apparent path is:

- single ROM only
- ROMs of differing access characteristics
- ROM memory management
- Flash first as storage media, then as update media, for either the entire application or selective parts
- Flash being loaded into execution RAM
- Flash being used as mass storage
- Single path execution on multiple processors

The future, I believe, will involve multiple path execution on multiple processors (which is part of some of the current engineering planning), and the use of flash as mass storage (like a file system on conventional processors but with characteristics more like the paged ROM in earlier embedded systems). Taking this path only slightly further into the 3- to 5-year range will see applications that, with functions running on multiple processors (perhaps distributed), are executing code from a single application source.

There still remain several language issues.

1) Pointer functions for a particular architecture have no need to access space beyond the normal natural address space of the underlying processor.
2) There may be some advantage to placing some functions in a specific address space. Execution speed differences for processors with internal and external memory implementation is one reason. The argument that this can be well handled by the linking phase of development is starting to change. The application needs to be part of the function placement. It is reasonable to expect that a function will reside in some address space that may not be part of the current processor's natural address space; how this call is achieved is implementation-specific. The application developer sees only that a function call is needed, and may, in some cases, need to control what memory space the function resides in.

Some final comments

It is important in this discussion to distinguish the difference between a implementation problem (arrays used in DSP-based filter software), and the capability of the language to support the needs of a broad base of users of the C language. Great care should be taken to not imply any specific implementation into C language definition. Adding named address space take nothing away from the current definition of the language. It adds support for the multiple address spaces that are a current reality, and it is flexible enough to support future embedded system architectures as they are developed.