Portability Techniques for Embedded Systems Data Management

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003
Phone: 425-888-8505

E-mail: info@mcobject.com
www.mcobject.com

Whether an embedded systems database is developed for a specific application or as a commercial product, portability matters. Most embedded data management code is still "homegrown," and when external forces drive an operating system or hardware change, data management code portability saves significant development time. This is especially important since increasingly, hardware's lifespan is shorter than firmware's. For database vendors, compatibility with the dozens of hardware designs, operating systems and compilers used in embedded systems provides a major marketing advantage.

For real-time embedded systems, database code portability means more than the ability to compile and execute on different platforms: portability strategies also tie into performance. Software developed for a specific OS, hardware platform and compiler often performs poorly when moved to a new environment, and optimizations to remedy this are very time-consuming. Truly portable embedded systems data management code carries its optimization with it, requiring the absolute minimum adaptation to deliver the best performance in new environments.

Using Standard C

Writing portable code traditionally begins with a commitment to use only ANSI C. But this is easier said than done. Even code written with the purest ANSI C intentions frequently makes assumptions about the target hardware and operating environment. In addition, programmers often tend to use available compiler extensions. Many of the extensions – prototypes, stronger type-checking, etc, – enhance portability, but others may add to platform dependencies.

Platform assumptions are often considered necessary for performance reasons. Embedded code is intended to run optimally on targets ranging from the low-end 8051 family, to 32-bit DSP processors, to high-end Pentium-based SMP machines. Therefore, after the software has been successfully built for the specific target, it is customary to have a "performance tuning" stage that concentrates on bringing out the best of the ported software on the particular platform. This process can be as straightforward as using compiler-specific flags and optimizations, but often becomes complex and time-consuming and involves patching the code with hardware-specific assembler. Even with C language patches, hardware-optimized code is often obscure and, more importantly, performs poorly on different machines.

Programmers also attempt to maintain portability through conditional code (#ifdef - #else) in a "master" version that is pre-processed to create platform-specific versions. Yet in practice, this method can create the customization and version-management headaches that portability is meant to eliminate. Another conditional code approach, implementing if-else conditions to select a processor-specific execution path at runtime, results in both unmanageable code and wasted CPU cycles.

All told, it's better to stick to ANSI C and to use truly platform-independent data structures and access methods as much as possible, to work around compiler- and platform-specific issues.

McObject developed the eXtremeDB in-memory embedded database adhering to ANSI C, with portability as one of the principal goals. Several techniques emerged, and are conveyed below, for

developers seeking to write highly portable, maintainable and efficient embedded code. Some of these apply to embedded systems portability generally, but are particularly important for data management. In many cases, an embedded application's database is its most complex component, and hence one where "getting it right the first time" (by implementing highly portable code) can literally save programmer-months down the road. Other techniques presented here, such as building light-weight database synchronization based on a user-mode spinlock, constitute specific key building blocks for portable embedded systems databases.

Word sizes

One proven technique is to avoid making assumptions about integer and pointer sizes. Defining the sizes of **all** base types used throughout the database engine code, and putting theses typedefs in a separate header file, makes it much easier to change them when moving the code from one platform to another or even using a different compiler for the same hardware platform:

```
#ifndef BASE_TYPES_DEFINED

typedef unsigned char   uint1;
typedef unsigned short  uint2;
typedef unsigned int    uint4;
typedef signed char     int1;
typedef short           int2;
typedef int             int4;

#endif
```

Defining a pointer size as a `sizeof(void*)` and using the definition to calculate memory layout offsets or using it in pointer arithmetic expressions avoids surprises when moving to a platform like ZiLOG eZ80 with three bytes pointers.

```
#define PTRSIZE sizeof(void *)
```

Note that the `void*` type is guaranteed to have enough bits to hold a pointer to any data object or to a function.

Data Alignment

Data alignment can be a portability killer. For instance, on various hardware architectures a 4-byte integer may start at any address, or start only at an even address, or start only at a multiple-of-four address. In particular, a structure could have its elements at different offsets on different architectures, even if the element is the same size. To compensate, the in-memory data layout used in McObject's eXtremeDB constantly requires allocating data objects to start from a given position, while handling the alignment of the element via platform independent macros that require no alteration when moving from one platform to another:

```
#define ALIGNEDPOS(pos, align) ( ((pos) + (align)-1) & ~((align)-1) )
```

```
pos = ALIGNEDPOS(pos, 4);
```

The code snippet above aligns the position of the data object (pos) at a 4-byte boundary.

Another alignment-related pitfall: on the some processors, such as SPARC, all data types must be aligned on their natural boundaries. Using standard C data types, integers are aligned as follows:

- short integers are aligned on 16-bit boundaries.
- int integers are aligned on 32-bit boundaries.
- long integers are aligned on either 32-bit boundaries or 64-bit boundaries, depending on whether the data model of the kernel is 64-bit or 32-bit.
- long long integers are aligned on 64-bit boundaries.

Usually, the compiler handles these alignment issues and aligns the variables automatically:

```
char c;
//(padding)
long  l;       - the address is aligned
```

But re-defining the way a variable or a structure element is accessed, while possible and sometimes desirable, can be risky. For example, consider the following declaration of an object handle (assuming the data object size being N bytes):

```
#define handle_size      N
typedef uint1 hobject    [handle_size ];
```

Such opaque handle declarations are commonly used to hide data object representation details from applications that access the data object with an interface function, using the handle merely as the object identifier:

```
typedef struct  appData_ { hobject h; } appData;

char c;
appData d;    - not aligned
```

Because `d` is a byte array the address is not memory aligned. The handle is further used as an identifier of the object to the library:

```
void* function (  appData *handle);
```

Furthermore, internally the library “knows” about the detail behind the handle and declares the object as a structure with the elements defined as short integers, long integers, references, etc:

```
typedef struct objhandle_t_
{
  ...
  obj_h         po;
  ...
  uint4         mo;
```

```
  uint2         code;
  ...
} objhandle_t;
```

Accessing object elements will lead to a bus error, since they are not correctly aligned. In order to fix the problem, we declare the object handle as an array of operands of the maximum size (as opposed to a byte array):

```
#define handle_size      N

#define handle_size_w
((( handle_size + (sizeof(void*) -1)) & ~(sizeof (void*) -1)) / sizeof(void*));

typedef void * hobject [handle_size_w ];
```

In this case, the compiler will automatically align the operands to their natural boundaries, preventing the bus error.

Word Endianness

Byte order is the way the processor stores multibyte numbers in memory. Big endian machines, such as Motorola 68k and SPARC, store the byte with the highest value digits at the lowest address while little endian machines (Intel 80x86) store it at the highest address. Furthermore, some CPUs can toggle between big and little endian by setting a processor register to the desired endian-architecture (IBM PowerPC, MIPS, and Intel Itanium offer this flexibility). Therefore, code that depends on a particular orientation of bits in a data object is inherently non-portable and should be avoided. Portable, endian-neutral code should make no assumptions of the underlying processor architecture, instead wrapping the access to data and memory structures with a set of interfaces implemented via processor-independent macros, which automatically compile the code for a particular architecture.

Furthermore, a few simple rules help keep the internal data access interfaces portable across different CPU architectures. First, access data types natively; for instance, read an `int` as an integer number as opposed to reading four bytes. Second, always read/write byte arrays as byte arrays instead of different data types. Third, bit fields defined across byte boundaries or smaller than 8 bits are non-portable. When necessary, to access a bit field that is not on byte boundaries, access the entire byte and use bit masks to obtain the desired bits. Fourth, pointer casts should be used with care. In endian-neural code, casting pointers that change the size of the pointed-to data must be avoided . For example, casting a pointer to a 32-bit value 0x12345678 to a byte pointer would point to 0x12 on a big-endian and to 0x78 on a little-endian machine.

Compiler Differences

Compiler differences often play a significant role in embedded systems portability. Although many embedded environments are said to conform to ANSI standards, it is well known that in practice, many do not. These non-conformance cases are politely called limitations. For example, although

required by the standard, some older compilers recognize `void`, but don't recognize `void*`. It is difficult to know in advance whether a compiler is in fact a strict ANSI C compiler, but it is very important for any portable code to follow the standard. Many compilers allow extensions; however even common extensions can lead to portability problems. In our development we have come across several issues worth mentioning, to avoid compiler-dependent problems.

When char types are used in expressions, some compilers will treat them as unsigned, while others treat them as signed. Therefore, portable code requires that char variables be explicitly cast when used in expressions:

```
#if defined( CFG_CHAR_CMP_SIGNED )
#define CMPCHARS(c1,c2) ((int)(signed char)(c1)-(int)(signed char)(c2) )
#elif defined( CFG_CHAR_CMP_UNSIGNED )
#define CMPCHARS(c1,c2) ((int)(unsigned char)(c1)-(int)(unsigned char)(c2) )
#else
#define CMPCHARS(c1,c2) ( (int)(char)(c1) - (int)(char)(c2) )
#endif
```

Some compilers cannot initialize auto aggregate types. For example the following may not be allowed by the compiler:

```
struct S { int i; int j; };
S s = {3,4};
```

The most portable solution is to add code that performs initialization:

```
struct S { int i; int j; };
S s;
s.i = 3; s.j = 4;
```

C-runtime library

Databases in non-embedded settings make extensive use of the C-runtime. However, embedded systems developers commonly avoid using the C-runtime, in order to reduce memory footprint. In addition, in some embedded environments, C-runtime functions, such as dynamic memory allocations/de-allocations (malloc() /free()), are implemented so poorly as to be virtually useless.

An alternative, implementing the necessary C-runtime functionality within the database runtime itself, reduces memory overhead and increases portability. For main memory databases, implementing dynamic memory management through the database runtime becomes vitally important, because these engines' functionality and performance are based on the efficiency of memory-oriented algorithms. eXtremeDB incorporates a number of portable embedded memory management components that neither rely on OS-specific low-level memory management primitives, nor make any fundamental assumptions about the underlying hardware architecture. Each of the memory managers employ their own algorithms, and are used by the database runtime to accomplish a specific task.

- A dynamic memory allocator provides functionality equivalent to the standard C runtime library functions malloc(), calloc(), free(), and realloc() according to the POSIX standard. The heap allocator is used extensively by the database runtime, but also can be used by applications.
- Another memory manager is a comprehensive data layout page manager that implements an allocation strategy adapted to the database runtime requirements. Special care is taken to avoid introducing unnecessary performance overhead associated with multi-threaded access to the managed memory pools.
- A simple and fast single-threaded memory manager is used while parsing SQL query statements at runtime, etc.

Synchronization

Databases must provide concurrent access across multiple simultaneously running tasks. Regardless of the database locking policies (optimistic or pessimistic, record-level or table-level, etc.) this mechanism is usually based on kernel synchronization objects, such as semaphores, provided by the underlying OS. While each operating system provides very similar basic synchronization objects, they do so with considerably different syntax and usage, making it non-trivial to write portable multithreaded synchronization code. In addition, an embedded systems database must strive to minimize the expense associated with acquiring kernel-level objects. Operating system semaphores and mutexes are usually too expensive to be used in embedded settings.

In our case the solution was to build up the database runtime synchronization mechanism based on a simple synchronization primitive – the test-and-set method – that is available on most hardware architectures. Foregoing the kernel for a hardware-based mechanism reduces overhead and increases portability. All we must do is port three functions to a specific target. This approach can also be used for ultra-low overhead embedded systems where no operating system is present (hence no kernel-based synchronization mechanism is available). Furthermore, the performance of the test-and-set “latch” illustrated below remains the same regardless of the target operating system and depends only on the actual target’s CPU speed.

```
/* this is the TAS (test-and-set) latch template*/
void sys_yield()
{
  /* relinquish control to another thread */
}
void sys_delay(int msec)
{
  /* sleep */
}
int  sys_testandset( /*volatile*/ long * p_spinlock)
{
  /* The spinlock size is up to long ;
   * This function performs the atomic swap (1, *p_spinlock) and returns
   * the previous spinlock value as an integer, which could be 1 or 0
   */
```

```
}
```

The code fragments below provide implementations for Win32, INTEGRITY OS and Sun SPARC platforms:

Win32-based synchronization:

```
#ifndef SYS_WIN32_H__
#define SYS_WIN32_H__

/* sys.h definitions for WIN32 */

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <process.h>


#define sys_yield() SleepEx(0,1) /*yield()*/
#define sys_delay(msec) SleepEx(msec,1)
#define sys_testandset(ptr) InterlockedExchange(ptr,1)

#endif  /* SYS_WIN32_H__ */
```

Sun SPARC based synchronization:

```
#ifndef SYS_MCOSOL_H__
#define SYS_MCOSOL_H__

/* sys.h definitions for Solaris */

#include <sys/time.h>
#include <unistd.h>
#include <sched.h>

int  sys_testandset( /*volatile*/ long * p_spinlock)
{
  register char result = 1;
  volatile char *spinlock = ( volatile char * ) p_spinlock;

      __asm__ __volatile__(
            "     ldstub      [%2], %0    \n"
:           "=r"(result), "=m"(*spinlock)
:           "r"(spinlock));
      return (int) result;

}
void sys_yield()
{
  sched_yield();
}
void sys_delay(int msec)
```

```
{ /* */ }
```

Green Hills INTEGRITY OS:

```
#ifndef SYS_GHSI_H__
#define SYS_GHSI_H__

/* sys.h definitions for Green Hills Integrity OS */
#include <INTEGRITY.h>

void sys_yield()
{
  Yield();
}
void sys_delay(int msec)
{
}
int  sys_testandset(long * p_spinlock)
{
  return ! ( Success == TestAndSet(p_spinlock, 0, 1) );
}
```

The concept of mutual exclusion is crucial in database development -- it provides a foundation for the ACID properties that guarantee safe sharing of data. The synchronization approach discussed above slants toward the assumption that for embedded systems databases, it is often more efficient to poll for the availability of a lock rather than allow fair preemption of the task accessing the shared database.

It is important to note that even though this approach is portable in the sense that it provides consistent performance of the synchronization mechanism over multiple operating systems and targets, it does not protect against the "starvation" of tasks waiting for, but not getting, access to the data. Also, provisions must be made for the database system to "clean itself up" if the task holding the lock unexpectedly dies, so that other tasks in line for the spinlock do not wait eternally. In any case, embedded data management is often built entirely in memory, generally requires a low number of simultaneous transactions, and the transactions themselves are short in duration. Therefore, the chances of a resource conflict are low and the task's wait to gain access to data is generally shorter than the time needed for a context switch.

Non-portable features

While replacing C runtime library functionality and memory managers and implementing custom synchronization primitives lead to greater data management code portability, sometimes it is not possible or practical to overload the database with functionality—such as network communications or file system operations—that belongs to the operating system. A solution is to not use these services directly, or re-create them in the database, but instead create an abstraction of them that is used throughout the database engine code. The actual implementation of the service is delegated to the application. This allows "hooking up" service implementations without changing the core engine code, which again contributes to portability.

For example, data management solutions often include on-line backup/restore features that, by their nature, require file system or network interaction. Creating an abstraction of stream-based "read" and "write" operations, and using this abstraction layer within the database runtime during backup, allows the database to implement the backup/restore logic while staying independent of the actual I/O implementation. At the same time, this approach allows a file-based, socket-based or other custom stream-based transport to be "plugged in" with no changes needed to the database runtime. The following segment illustrates such a "plug-in" interface:

```
/* abstraction of write and read stream interfaces;
 * a stream handle is a pointer to the implementation-specific data
 */
typedef int (*stream_write)
  ( void *stream_handle, const void * from, unsigned nbytes);
typedef int (*stream_read)
  ( void *stream_handle, /*OUT*/ void * to, unsigned max_nbytes);

/* backup the database content to the output stream */
RETCODE db_backup
  ( void * stream_handle, stream_write output_stream_writer, void * app_data);

/* restore the database from input stream */
RETCODE  db_load
   ( void * stream_handle, stream_read input_stream_reader, void *app_data);
```

The application needs to implement the actual read-from-stream / write-to-stream functionality. For example, if read/write to a file is sufficient:

```
int file_writer(void *stream_handle, const void * from, unsigned nbytes)
{
  FILE *f = (FILE*)stream_handle;
  int nbytes = fwrite(from,1,nbytes,f);
  return nbytes;
}

int file_reader(void *stream_handle, void * to, unsigned max_nbytes)
{
  FILE *f = (FILE*)stream_handle;
  int nbytes = fread(to,1,max_nbytes,f);
  return nbytes;
}
```

Another example of the database "outsourcing" services to the application involves network communications. Embedded databases often must provide a way to replicate the data between several databases over a network. Embedded settings always demand highly configurable and often deterministic communication that is achieved using a great variety of media access protocols and transports. Thus, as a practical matter, a database should be able to adopt the communication protocol used for any given embedded application, regardless of the underlying hardware or the operating system. Instead of communicating directly with the transport or a protocol, a database runtime goes through a thin abstraction layer that provides a notion of a "communication channel." Like the backup/restore interfaces, the network communication channel can also be implemented via a stream-based transport:

```
#define channel_h void*

typedef int (*xstream_write)(channel_h ch, const void * from,
                             unsigned nbytes, void * app_data);

typedef int (*xstream_read) (channel_h ch, void * to,
                             unsigned max_nbytes, void* app_data);


typedef struct {
  xstream_write fsend;
  xstream_read  frecv;
  ...
 } channel_t, *channel_h;
```

The database would use a set of API functions that provide the ability to initiate and close the channel, send and receive the data, and so on.

Conclusion

While the idea of commercially available, "off-the-shelf" databases is catching on for embedded systems, that market's great profusion of hardware designs, operating systems and compilers raises the probability that embedded systems developers will at some point develop data management from scratch – or at least customize an existing database management system. And some embedded systems developers *like* to "roll their own," taking the complexities of multi-user coordination, transaction management, lookup algorithms, hot backup and the like as a challenge. By following several general rules of developing portable code—such as using standard C and avoiding assumptions about hardware related parameters—developers can greatly simplify the re-use of their data management code in new embedded environments. And new approaches, such as those described above, to implementing standard database services can ensure that the old concept of a database delivers the portability, performance, and low resource consumption demanded for embedded systems.