smx++™

C++ Interface for smx

smx++ is designed to enable C++ experts and C++ novices, alike, to reap the benefits of object-oriented programming in embedded systems.

C++ experts will immediately see how to derive their own classes from those provided by smx++. Novices will be guided by the easy to follow smx++ manual and clear examples. In-plant training and consultation services are also available to help users quickly jump the gap to object oriented programming.

smx++ provides the following basic features needed to support C++ in a multitasking environment:

- C++ kernel service API.
- *this* pointer management.
- Frequent creation and deletion of C++ objects.
- Recycling C++ objects.
- Global object initialization and destruction.
- Interoperability between C and C++ code.

Classes

- smx_Object
- smx_BlockPool
- smx_EventCounter
- smx_EventFlags
- smx_Msg
- smx_MsgXchg
- smx_MsgXchgRsrc
- smx_Mutex
- smx_NewPool
- smx_Pipe
- smx_Sem
- smx_Task
- smx_Timer

smx++ API

The smx++ API has been designed to be modular and to provide good support for C++ object-oriented programming. smx++ is intended to provide a somewhat simpler and more protected API than the smx C API. It makes use of C++ features, such as inheritance, function overloading, and default parameters to achieve this. Some smx functionality which does not fit the C++ paradigm has been left in smx and will need to be utilized in C. The goal is to make it very easy to derive C++ classes, without needless traps and distractions.

Each object contains all of the methods required to manipulate it. Data members are generally declared as protected fields in order to prevent accidentally modifying sensitive data. The smx++ classes have been designed to be used as base classes for user classes and to make proper use of virtual and pure virtual methods, as appropriate, to ensure methods can or must be implemented in derived classes, respectively.

See the smx++ class summary at the end of this datasheet for more information.

this Pointer

C++ uses a hidden pointer, called the *this* pointer to point to C++ objects, each of which is a structure that provides information concerning the object. When an smx++ task is constructed, its *this* pointer is stored in the task's TCB. When the task is first started or restarted, its *this* pointer is restored and used to access the smx++ task instance. For C tasks, this field in the TCB is NULL, and the task's main function is called as a normal C function. Hence, smx plus smx++ permit a mixture of C and C++ tasks. Mixed C/C++ coding is discussed more fully below.

Frequent Creation and Deletion of C++ Objects

It is a characteristic of C++ that objects are frequently created (*constructed*) and deleted (*destructed*) as they go in and out of scope. This is unlikely to apply to tasks, but does apply to objects such as messages, blocks, and possibly semaphores, mutexes, etc., as well as to application objects derived from smx++ objects. The heap is normally used for C++ objects. In embedded systems, smx++ replaces the compiler heap with the smx heap to improve reliability. However, a heap is vulnerable to fragmentation, causing possible failure. This is because available memory in embedded systems is typically small, and embedded systems must run forever. Also, non-deterministic search times of large heaps are a problem for embedded systems, even if there is ample memory.

For this reason smx enables overloading the C++ *new* and *delete* operators with smx block pools. The blocks are large enough for any smx++ objects (approximately 12 bytes). Applications can define block pools having larger blocks for application objects. Block pools cannot be fragmented and they are fast and deterministic, so they solve the main problems. However, an application can run out of blocks. In this case, an error is reported, which can be easily fixed by enlarging the block pool.

Recycling Dynamic Objects

When an smx++ object, such as a task, is created, both a small task object instance is allocated and a standard smx Task Control Block (TCB) is allocated. The task object is linked to the TCB and most information resides in the TCB. smx is able to deal with dynamic C++ objects better than most kernels because it uses dynamic control blocks, which are allocated from control block pools. When an smx++ (or derived) object is deleted, both its object instance and its smx control block(s) are returned to their pools and are available for reuse.

This is not possible with other kernels, which use statically defined control blocks defined as:

```
TCB  taskA;
```

The control block is identified to kernel services, by its address &taskA. This is more than a little clumsy for recycling. For smx, a task is identified by its handle, taskA:

```
TCB_PTR  taskA;
```

For smx++, the handle is a field in the task object. Hence, when an smx++ task is deleted, both the handle and the TCB are recycled. This is true for all objects.

Global Object Initialization

For objects defined at the global level (which is necessary for objects shared between C and C++ code — see below) the compiler creates *initializers* which invoke constructors for the objects. These initializers are called from the startup code which runs before main(). smx automatically allocates control block pools and control blocks as needed for the objects being constructed.

Mixed C/C++ Coding

It is usually desirable to write low-level code such as Interrupt Service Routines (ISRs), Link Service Routines (LSRs), and certain core tasks (e.g. IdleMain()) in C (and sometimes partially in assembly). Also there may be legacy code, and C may seem more compatible with certain parts of a system or be more comfortable for some project members.

C++ is good for higher-level functions such as graphical user interfaces (e.g. using PEG), audio and video processing, complex data processing, database management and processing, communications, high-level protocols, user interactions, etc.

smx and smx++ permit mixed C and C++ systems. Tasks can be created either in smx or in smx++ and the scheduler will start each correctly. Objects such as exchanges and semaphores can be constructed in smx++ and accessed from C tasks. Here is an example of an smx++ object used from a C function:

```
class MyTask : smx_Task
{
      MyTask() : smx_Task(PRI_NORM, 0,
                          SMX_FL_NONE, "ATask")  {};

      virtual void Main(uint)
      {
            // do stuff here
      }

} ATask;

void func(void)   /* C function */
{
      ATask.Start();
}
```

Going the other way (i.e. using smx objects in smx++) is error-prone, and thus not permitted. Should this be necessary, create the object in smx++ and change the C code that uses it.

smx++ Classes

smx_Object is the base class for all smx++ objects, except smx_NewPool objects. All classes derived from smx_Object use the GlobalPool for dynamic memory allocation.

smx_BlockPool class provides a C++ interface to create and access an smx heap or DAR block pool. Heap and DAR block pools are discussed in the smx Reference Manual.

smx_EventCounter class is the wrapper for an smx event queue. It is commonly used to count tick events, as a delay mechanism for tasks.

smx_EventFlags class functions like the smx event flags group. It is used to allow Tasks to wait on logical combinations of events.

smx_Msg is a general class that can hold any type of message. Messages can be allocated from the heap or from a block pool.

smx_MsgXchg class provides both Pass Exchange and Normal Exchange capabilities. It is also the base class for the smx_MsgXchgRsrc class.

smx_MsgXchgRsrc class is used to create a resource exchange to store free messages for later use.

smx_Mutex class is used to create a mutex, possibly with priority inheritance and/or a ceiling priority enabled.

smx_NewPool class provides fast, efficient, non-fragmenting memory management for C++ objects via reentrant new and delete operators.

smx_Pipe class is used in the same manner as an smx pipe for communication with a hardware device or between tasks.

smx_Sem class is used to create a semaphore with a specified threshold and one or more task priority levels.

smx_Task class defines the API for working with tasks. This class incorporates integrated virtual methods for both the task main function and the hooked entry and exit functions called by the smx scheduler. This allows the user to derive a class from the smx_Task class that contains its own main and hook methods.

smx_Timer class functions like the smx timer object. It is used to create precise timing delays between events executed by LSRs.

In nearly all cases, the methods for the above classes are the same as smx services. See the smx datasheet for a list and description of smx services. For more detailed information on smx++ methods, see the smx++ Developers Guide.

References

1. smx datasheet
2. smx++ Developer's Guide
3. smx User's Guide
4. smx Reference Manual