BaseLib2 Tutorial Series
Introduction and threading

André Neto

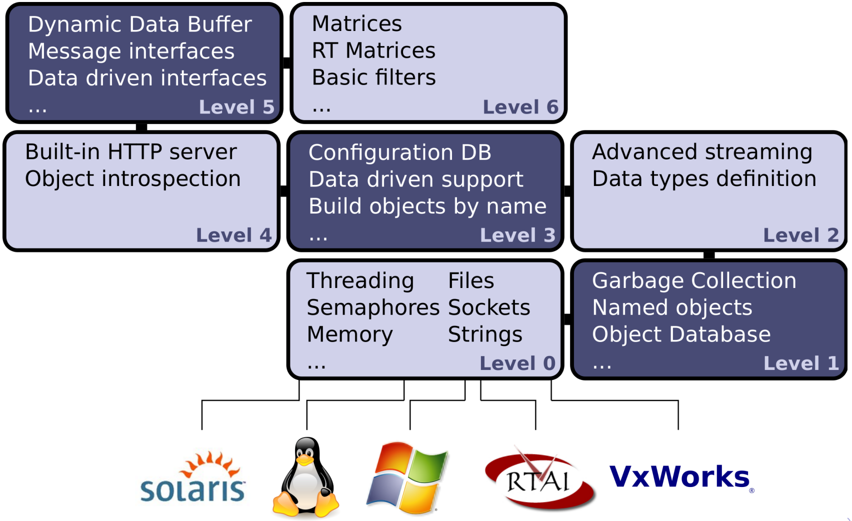
May 25, 2011

Outline

Design ideas

- Multi-platform C++ library, code once run everywhere approach
  - Very important when developing complex codes for real-time applications
- Designed to provide all the basic ingredients required to develop a real-time application
  - Does not ensure real-time by itself
    - Only when executing in a real-time OS...
    - ..and if the code was carefully written
  - ... but provides all the interfaces to quickly enable real-time features when the real-time OS is available

Layers


Dynamic Data Buffer
Message interfaces
Data driven interfaces
...
Level 5
Matrices
RT Matrices
Basic filters
...
Level 6
Built-in HTTP server
Object introspection
Level 4
Configuration DB
Data driven support
Build objects by name
...
Level 3
Advanced streaming
Data types definition
Level 2
Threading
Files
Semaphores
Sockets
Memory
Strings
...
Level 0
Garbage Collection
Named objects
Object Database
...
Level 1
solaris
RTAI
VxWorks

Note on other libraries

Dependencies

- BaseLib2 was designed to work without any dependencies
    - This also applies to the C++ stdlib and you should avoid mixing the two

Real-time codes

- be extremely careful when
    - you have to depend on libraries that you don't maintain (open-source or not)
    - link with libraries that were not designed to work in real-time operating systems (VxWorks)

Background

1. Knowledge of any C style-like syntax language
2. Knowledge of C++ (or at least object oriented)
   1. Thinking in C++ 2nd Edition by Bruce Eckel (free on the web)
   2. http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
3. If you know Java, you'll find similitudes in the style...

Environment

1. Any of the supported operating systems
    1. Linux, Mac OS X, MS Windows, Solaris, VxWorks
2. C++ compiler
    1. GNU Compiler Collection (gcc)
    2. For MS Windows use Cygwin (http://www.cygwin.com/) and visual studio compiler
3. CVS
4. Text editor
5. (The material for this tutorial should run in any jac machine)

Getting the source

CVS user

If you don't have a CVS username let me know

Instructions

```
export CVSROOT=:pserver:aneto@cvsppcc.jet.uk:2401/home/ppcc-dev/CVS_REPOSITORY
cvs login
cvs co MakeDefaults BaseLib2 MARTe
cvs logout
```

Makefile structure

- Each binary container (executable or library) has
  - Makefile.inc
    - lists all the objects that are to be compiled for the given target
    - contains what are the expected outputs (executable, static library, dynamic library)
  - Makefile for each operating system (also known as target)
    - name is always Makefile.osname (e.g. Makefile.linux Makefile.solaris)
- The MakeDefaults directory contains the compilation instructions for the different operating systems
  - You don't have to change these

Makefile.inc acronyms

Acronym	Meaning
CFLAGS	Extra compilation flags to be sent to the compiler
OBJSX	List of objects to be compiled
MAKEDEFAULTDIR	The location of the MakeDefault directory
TARGET	The operating system identifier (linux, macosx, rtai, v6x5100, v6x5500, vx5100, vx5500)
OBJS	Compiles all OBJSX
DLLEXT	Produce a dynamic library
LIBEXT	Produce a static library
EXEEXT	Produce an executable
LIBRARIES	Other libraries to link

There are others, but these are the most important...

Makefile.inc example

Makefile.inc

```
OBJSX=MessageTriggeringMask.x
MAKEDEFAULTDIR=../../../MakeDefaults
include $(MAKEDEFAULTDIR)/MakeStdLibDefs.$(TARGET)
CFLAGS+= -I.
CFLAGS+= -I../../../BaseLib2/Level0
CFLAGS+= -I../../../BaseLib2/Level1
CFLAGS+= -I../../../BaseLib2/Level2
CFLAGS+= -I../../../BaseLib2/Level3
CFLAGS+= -I../../../BaseLib2/Level4
CFLAGS+= -I../../../BaseLib2/Level5
CFLAGS+= -I../../../BaseLib2/Level6
CFLAGS+= -I../../../BaseLib2/LoggerService
CFLAGS+= -I../../MarteSupportLib
all:  $(OBJS) \
$(TARGET)/MessageTriggeringTimeService$(DLLEXT)
echo $(OBJS)
include depends.$(TARGET)
include $(MAKEDEFAULTDIR)/MakeStdLibRules.$(TARGET)
```

Makefile.os

- Usually only specifies the target and includes the Makefile.inc
    - when required, OS dependent libraries are also set here

Makefile.linux (example)

```
TARGET=linux
include Makefile.inc
LIBRARIES += -L../../../BaseLib2/$(TARGET) -lBaseLib2
-L../../MarteSupportLib/$(TARGET) -lMarteSupLib
OPTIM=
```

BaseLib2 type definitions (see GenDefs.h)

- Each type has the same meaning in all operating systems where your code might run
  - (float and double were not redefined)

Type	Meaning
int8	signed 8 bit integer
uint8	unsigned 8 bit integer
int16	signed 16 bit integer
uint16	unsigned 16 bit integer
int32	signed 32 bit integer
uint32	unsigned 32 bit integer
int64	signed 64 bit integer
uint64	unsigned 64 bit integer
float	IEEE-754 single precision floating point number
double	IEEE-754 double precision floating point number
intptr	large enough to store an integer pointer in any architecture

SystemXYZ.h

- Contains special definitions which are very OS dependent
    - Handles for files
    - Some low-level functions that might not be available in this OS
- Includes all the header files that are OS dependent
- e.g. see: SystemLinux.h, SystemVX5100.h, ...
- System.h includes all these files

Endianity.h

- Endianness defines in what order data is stored in the computer memory
  - little-endian (named Intel in this file): MSB in highest address
  - big-endian (named Motorola in this file): MSB in lowest address

Example

Decimal: 2864434397
Hex:0xAABBCCDD

Address	n	n+1	n+2	n+3
Little-endian	DD	CC	BB	AA
Big-endian	AA	BB	CC	DD

- Depends on the computer architecture and even in the programming language (Java is always big-endian)
- Very useful when you have to convert data from other data sources (I/O, network, ...) to the endianity of your environment

ErrorManagement.h

- Defines unified way of logging information
- Higher level code decides how this information should be displayed (file, network, ...)

CStaticAssertErrorCondition

CStaticAssertErrorCondition(EMFErrorType errorCode,const char *errorDescription,...)

errorCodes
Information
Warning
FatalError
InitialisationError

Example

CStaticAssertErrorCondition(Information,**"Hello %d times",10)**

Basic ideas

- Threads enable you to run code in *parallel*
- Threads are selfish by nature and try to run while there is some work to be completed
- Scheduler divides and arbitrates CPU time amongst threads
    - Several rules and scheduling schemes are available (Round-robin, FIFO, ...)
    - Real-time operating system are usually not fair and give more time to the threads with higher priority
- When threads have `nothing to do`, control should be voluntarily returned to the scheduler
    - Until something happens
        - wait on a semaphore
        - poll a resource (hum...)
    - Sleep for a defined amount of time

Creating threads

Threads::BeginThread

Threads::BeginThread(**ThreadFunctionType function,void *parameters,uint32 stacksize,const char *name,ExceptionHandlerBehaviour ehb,ProcessorType runOnCPUs)**

Parameter	Description	Default value
function	The function to be called by the thread	-
parameters	Parameter to be passed to the thread	NULL
stacksize	Stack size	Depends on arch.
name	Thread name	NULL
ehb	Not implemented	XH_NotHandled
runOnCPUs	CPU mask where thread can run	2 in multi-core, 1 otherwise

ThreadFunctionType

```
typedef void (*ThreadFunctionType)(void *parameters);
e.g. void MyThreadCallBack(void *myParameters){}
```

Sleep.h

- Voluntarily return control to scheduler
  - Real-time OS should provide better precision (in particular for **SleepNoMore**)
  - Other similar functions are available

Sleep

```
void SleepSec(double sec);
void SleepNoMore(double sec);
void SleepAtLeast(double sec);
```

Threads example (BaseLib2/Documentation/Tutorials/examples/ThreadExample1.cpp)

Source

```
//Shared variable to be incremented by the threads
static int32 sharedVariable = 0;
//Thread function call back
void IncrementDecrementFunction(void *threadID){
     ...
     sharedVariable++;
     ComplexAnalysis((thisThreadID + 1) * 1e-3);
     sharedVariable--;
     ...
}
int main(int argc, char *argv[]){
    //Output logging messages to the console
    LSSetUserAssembleErrorMessageFunction(NULL);
    ...
    for(i=0; i<numberOfThreads; i++){
        Threads::BeginThread(IncrementDecrementFunction, (int32 *)i, THREADS_DEFAULT_STACKSIZE,
NULL, XH_NotHandled, 0x1);
    }
     ...
    CStaticAssertErrorCondition(Information, "Value of sharedVariable = %d", sharedVariable);
```

- Several threads chaotically increment and decrement a shared variable

Concepts

- Extremely important when shared resources are accessed by different threads
- Usually protected using semaphores
    - Mutex types, guarantees that only a single thread can be accessing a shared resource at a time
    - Event types, collection of threads wait for an event to happen, after which are all allowed to interact with the resource
- Atomic types, guarantee the atomicity of some operations in variables
- All are important and to be used accordingly to the context

Atomic.h

- Enables integer variables to be increment, decremented or exchanged in a thread safe way
- Can be used if you need to test a given variable, without the burden of a semaphore
- Very good for spinlock-like implementations

Most important functions

```
void Increment(volatile *intXYZ)
void Decrement(volatile *intXYZ)
intXYZ Exchange(volatile *intXYZ, intXYZ newValue) //Returns the old value
bool TestAndSet(volatile *intXYZ) //Checks if a given variable is not zero
```

Atomic example (BaseLib2/Documentation/Tutorials/examples/AtomicExample1.cpp)

Source

```
//Shared lock
static int32 locked = 0;
....
int32 a = 3;
int32 b = 4;
//Exchange
if(b = Atomic::Exchange(&a, b)){
    CStaticAssertErrorCondition(FatalError, "Failed to exchange the contents of a and b!");
}
CStaticAssertErrorCondition(Information, "After exchanging a=%d b=%d", a, b);
//Atomic increment
Atomic::Increment(&a);
//Lock again CStaticAssertErrorCondition(Information, "locked should now be 1 locked = %d",
locked);
//Create thread to perform the unlock
CStaticAssertErrorCondition(Information, "Going to wait for thread to unlock");
Threads::BeginThread(UnlockWithTestAndSet);
//Spin lock
while(!Atomic::TestAndSet(&locked));
```

Mutex Semaphores

- Protects a shared resource
  - Only one thread can interact with it at the same time
  - Can be a variable
  - Can be a block of code
- Two types of Mutex available (both with timeout)
  - MutexSem (driven by the scheduler)
  - FastPollingMutexSem (spinlocks in a variable)

Most important in MutexSem.h

```
bool Create(bool locked = False)
bool Lock(TimeoutType msecTimeout = TTInfiniteWait) //notice and check against the timeout
bool UnLock()
bool Close()
```

Most important in FastPollingMutexSem.h

```
bool Create(bool locked = False)
bool FastLock(TimeoutType msecTimeout = TTInfiniteWait)
bool FastUnLock()
bool FastTryLock()
bool Close()
```

Event Semaphores

- Collection of threads wait for resource to be available
- Usually driven by an event
- Timeout can be specified

Most important functions for EventSem

```
bool Create()
bool Reset()
bool Wait(TimeoutType msecTimeout = TTInfiniteWait)
bool Post()
bool Close()
```

Semaphore example (BaseLib2/Documentation/Tutorials/examples/ThreadExample2.cpp)

Source

```
void IncrementDecrementFunction(void *threadID){
    int32 thisThreadID = (int32)threadID;
    CStaticAssertErrorCondition(Information, "Thread with id = %d waiting for event sem",
thisThreadID);
    if(!eventSem.Wait()){
        CStaticAssertErrorCondition(FatalError, "Thread with id = %d failed to wait in event sem
(timeout?)", thisThreadID);
    }
...
    //The mutex protects this region of code
    if(!mutexSem.Lock()){
        CStaticAssertErrorCondition(FatalError, "Thread with id = %d failed to wait in mutex sem
(timeout?)", thisThreadID);
    }
    sharedVariable++;
    ComplexAnalysis((thisThreadID + 1) * 1e-3);
    sharedVariable--;
    exitAfterCalls--;
    //Unprotect here
    if(!mutexSem.UnLock()){
        CStaticAssertErrorCondition(FatalError, "Thread with id = %d failed to unlock mutex
sem", thisThreadID);
    }
```

High Resolution Timer (HRT.h)

- Enables to have access to the CPU high resolution timer counter
    - Perfect tool to measure time elapsed between two points
        - Multiply number of elapsed ticks (HRTCounter) times the period (HRTPeriod)

Most important functions

```
int64 HRT::HRTCounter()
int64 HRT::HRTFrequency()
int64 HRT::HRTPeriod()
```

Example

```
int64 countsAtT1 = HRT::HRTCounter();
...
CStaticAssertErrorCondition(Information, "Time elapsed = %f", ((HRT::HRTCounter() - countsAtT1)
* HRT::HRTPeriod());
```

Training ideas

1. Create a program where a thread adjusts its Sleep time to be as precise as possible
    1. This means that you should measure how long the thread is actually sleeping and dynamically adjust the sleep value
2. Rewrite ThreadExample2 using FastPollingMutexSems
3. Rewrite ThreadExample2 using Atomic spinlocks