@implementation

MAKING MAB APPLICATIONS FROM THE COMMAND LINE

Q: I know how to make Multiple Architecture Binary (MAB) applications using Project Builder, but how do I perform a MAB **make** from the command line?

A: It depends on which version of NEXTSTEP you're using.

In Release 3.1, Project Builder uses the UNIX **make** facility to build projects. **make** is an extremely large and complex utility, but for our purposes it's fairly simple. Ordinarily, to **make** a debug version of a project, you would type **make debug**; to make an installed version, you'd type **make install**. To generate a MAB target, you just extend the **make** command to override some of the default values of symbols that your **Makefile** uses.

This **make** command overrides two NEXTSTEP makefile symbols to create a MAB target:

```
make debug RC_CFLAGS=" -arch m68k -arch i386" RC_ARCHS=" m68k i386"
```

Here's what the symbol-value assignments above mean:

```
RC_CFLAGS=" -arch m68k -arch i386"
```

This ensures that **make** uses the flags on the **cc** commands that it generates when compiling **.c** or **.m** source files into **.o** object files.

```
RC_ARCHS=" m68k i386"
```

This symbol is required so that the **obj** directory containing the machine code corresponding to each source file is put in **m68k_i386_debug_obj**. Without this flag, the MAB objects would

be stored in an **m68k_debug_obj** or **i386_debug_obj** directory, depending on what architecture you're running **make** on. This would be both incorrect and misleading.

Alternatively, you can add the following two lines to the end of the **Makefile.postamble** file in your project directory:

```
RC_CFLAGS = arch m68k -arch i386
RC_ARCHS = m68k i386
```

Then, you can run **make debug** to make a MAB application automatically. Note, however, that the options you set in Project Builder still dictate the type of build you perform thereÐeven if you make this modification to the **Makefile.postamble**, if you build thin from Project Builder, you get a single architecture ªthinº application.

In Release 3.2, the same scheme also works. However, to make things more convenient, the TARGET_ARCHS symbols have been added, and should be defined in much the same way as RC_ARCHS in the above example. In the 3.2 **Makefile** scheme, however, you needn't define RC_CFLAGS as well.

COMPILING A MAB ADAPTOR

Q: When I try to statically link an adaptor into my MAB Database Kit application in NEXTSTEP Release 3.1 or 3.2, I get the following errors at compilation time:

```
ld: for architecture i386
ld: warning /NextLibrary/Adaptors/SybaseAdaptor.adaptor/SybaseAdaptor
cputype (6, architecture m68k) does not match cputype (7) for specified -arch
flag: i386 (file not loaded)

ld: for architecture m68k
ld: warning /NextLibrary/Adaptors/SybaseAdaptor.adaptor/SybaseAdaptor cputype
(7, architecture i386) does not match cputype (6) for specified -arch flag: m68k
```

```
(file not loaded)
```

A: The adaptors provided with Release 3.1 or 3.2 are for a single architecture type only. For example, to build a MAB Sybase Adaptor file, you need to do the following:

1 Get the two versions of **SybaseAdaptor** from **Adaptors/SybaseAdaptor.adaptor** in **NextLibrary**. For convenience, we rename them respectively **SybaseAdaptor_m68k** and **SybaseAdaptor_i386** in this example.

2 Use the UNIX command **lipo**(1) to create a MAB adaptor file from these two input files:

```
myhost> lipo SybaseAdaptor_m68k SybaseAdaptor_i386 -create -output
~/Library/Adaptors/SybaseAdaptor_MAB
```

Now, you can hard link the adaptor into your MAB application.

DISTRIBUTED OBJECT CONNECTION FAILING

Q: I have a client application that's connecting to a server on another machine. When I specify the hostname, the client successfully connects to the server. But, when I specify the a*o wild card
as the hostname parameter to **connectToName: onHost:fromZone:**, the connection fails and **nil** is returned. What causes this?

A: When you explicitly specify the hostname, the Distributed Object (DO) request is efficiently routed using the usual TCP/IP routing mechanism to make the connection. However, when you use a wild card search for the host, each machine on the subnet must be queried. For this reason the wild card search limits itself to the immediate subnet. In other words, the wild card search fails if the two machines are not on the same subnet; the two nmserver processes are unable to find each other. To find out more, see the documentation for **connectToName:onHost:fromZone:**.

You can use two strategies to avoid using the wild card when you don't know the hostname:

- Hard code a machine name. Of course, it would be a friendly gesture to allow this machine name to be changed in the Preferences panel of the client application.
- Register the name with NetInfo. For example, you might want to create a new directory in the **locations** directory called **myAppNameServer**. The **myAppNameServer** directory would contain a property that lists the value of the hostname where the server is located.

(Valid for NEXTSTEP Releases 3.0, 3.1, and 3.2)

INTERNAL COMPILER ERROR

Q: When compiling the following snippet of code with the C++ compiler, I get the error message
Internal Compiler Error 89,º and the compiler crashes.

```
extern "C"
{
#include <bsd/libc.h>
#include <mach/cthreads.h>
}

class Test {
   struct condition aCondition;
   struct mutex aMutex;
public:
   void wait()    { condition_wait(&aCondition, &aMutex); }
};

main()
{
```

```
    Test  test;
    test.wait();
    exit(0);
}
```

A: This error is caused by a name clash of the symbol **wait** between the definition in the header file **/NextDeveloper/Headers/g++/sys/wait.h** and your member function **wait()**. To work around this bug, you can rename your **wait()** function to something else, such as **wait_on()**.

(Valid for NEXTSTEP Releases 3.1 and 3.2)

ARCHIVING AN IXSTORE OBJECT

Q: When I archive and unarchive an IXStore object, I'm able to write it to a typed stream, but reading it back gives me a memory protection failure with the following backtrace.

```
Program generated(1): Memory access exception on address 0xe
(protection failure).
0xa025f46 in -[IXStore read:] ()
(gdb) where
#0  0xa025f46 in -[IXStore read:] ()
#1  0x500c9be in InternalReadObject ()
#2  0x500f0f8 in NXReadObject ()
```

A: The memory smasher occurs in **InternalReadObject()** (the archiving code) when that method tries to send an **awake** message to the unarchived store object, which has been freed. The store was freed because transactions weren't enabled and so it was in a partially updated state.

To fix this problem, you can, for example, include **commitTransaction** in the **write:** method

of your store object to finish all outstanding transactions before archiving. Note that this is only necessary if transactions are disabled. If transactions are enabled, the store can be archived
with incomplete transactions pending; reading it back will only drop the uncommitted changes:

```
/* Make a new storage object with a brand new IXStore */
- init
{
   [super init];
   storage = [[IXStore alloc] init];
   return self;
}

/* Archiving myself */
- read:(NXTypedStream *)stream
{
   [super read:stream];
   storage = NXReadObject(stream);
   return self;
}

- write:(NXTypedStream *)stream
{
   [super write:stream];
      /* A convenient place to finish outstanding
      transactions. Not needed if transactions are
      enabled.
      */
   [storage commitTransaction];
   NXWriteObject(stream, storage);
   return self;
}
```

Please note that this program crasher has been fixed in Release 3.2, and an exception error is raised instead. However, you still need to follow the guideline to properly archive an IXStore object.

(Valid for NEXTSTEP Releases 3.1 and 3.2)

Next Article NeXTanswer #1643 **Info Panel**
Previous article NeXTanswer #1637 **Core Dump**
Table of contents http://www.next.com/HotNews/Journal/NXapp/Spring1994/ContentsSpring1994.html