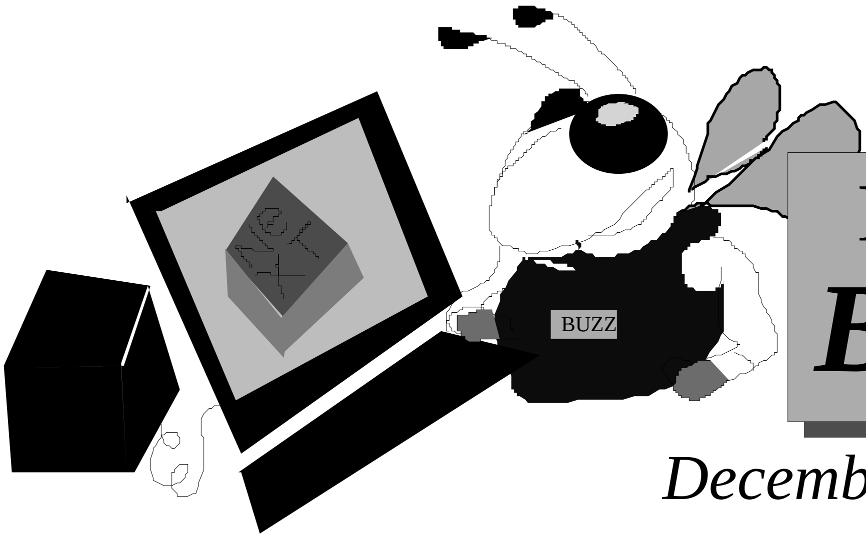
The NeXT Academic Project and 3rd Party Product Directories are out. Contact NeXT if you haven't received yours.



BuzzNUG *Buzzings*

December 1989 - Issue 1

WELCOME ONE AND ALL TO THE FIRST ISSUE OF THE BuzzNUG BUZZINGS!!!

Contents

The BuzzNUG General Council
Erica J. Liebman, Pres. erica@kong.gatech.edu
David Rosenbaum, V.P. daver@pyr.gatech.edu
David Samuel King, Tres/Sec dsking@pyr.gatech.edu
Tamora V. Sealey, Accounts tammy@cadnext2.gatech.edu

Contributers : Erica J. Liebman, Tamora V. Sealey, John T. Nelson, Herri Gunawan, Brian Skinner, Roger Rosner, and Jonathan Schwartz & the BuzzNUG membership.

Welcome!
Erica Liebman

Well here it is, the first issue (and hopefully the first of many) of the BuzzNUG Buzzings. Thank you everyone who has pitched in to help turn this wild idea into palpable reality. But don't relax right away. We still need more help. The newsletter and user-group reality needs to grow and mature and we are counting on a lot of you to help pitch in -- with ideas, with articles and with suggestions. We need you, the members, to help define what BuzzNUG is and what it will become.

BuzzNUG was originally created to fill some special needs that weren't being met by the (highly recommended) Internet "Comp.Sys.NeXT" discussion group. The *Buzzings* is going to focus on how-to articles, on reviews, on editorials and on technical discussions at a greater length and depth of focus than is really appropriate for an Internet newsgroup.

The key word in all this is **user**. We're not looking to serve the small computer community, or compare against Suns and Macs. Our focus is the NeXT machine. We're looking to help define, understand and aid the needs of the NeXT user/programmer. Many of us are NeXT developers. Several of us are Corporate programmers. And quite a lot are students -- most here at Georgia Tech. I myself fall into the first and third categories. What joins us together is our interest in a new and exciting machine, (with a really neat development environment -- I might add) and a wish to share knowledge and experience.

The possibilities for the Buzzings are pretty astonishing -- as NeXT users we immediately have access to a great desktop publishing system in WriteNow and Draw (even if we can't afford the $500 educational Pagemaker fee). We can include pictures and diagrams and even sound resources. *Our* interviews can include audio exerpts! (Try to do *that* on internet.)

To make this user group work, to make any venture work, is a team effort. We're glad you decided to join BuzzNUG and we're counting on each and every one of you for success. Send in those articles. Send in those editorials. Send in those NeXT challenges, puzzles and class-object requests for the "Object Bee-Line".

Oh yeah. The name. Buzz, is Georgia Tech's mascot. He is a yellow-jacket bee. You'll find a rather inept picture of him on the cover. (If any of you out there in User-land can come up with a better picture, mine is history.)

I hope you enjoy the rest of this first issue of BuzzNUG.

NeXT Availability for Students
Tamora Sealy

The College of Engineering CAE/CAD Laboratory announces the opening of six NeXT computers to be used by Georgia Tech qualified students and faculty. These computers are located in the A. French Bldg Room 102 and may be accessed via Georgia Tech's TCP/IP network.

For more information please contact:
Tammy Sealey 4-7774 French 209

My Love/Hate Relationship with The Cube
(disk drives and sharing software)
John T. Nelson

Steve Jobs' latest brainchild represents a wonderful synthesis of design concepts: increasing overall throughput by implementing a mainframe on custom VLSI, megapixel display with overlay bits for increased readability, floptical disk drive so that students and researchers can easily mount their "worlds" on a common machine, DSP chip for high speed crunching (particularly useful for music applications), Mach operating system, reasonably complete development environment and lots of value added software. If you're into music, and also want a Unix development machine (as I am) then NeXT would seem to be the ideal choice.

So if it's so good why haven't I purchased one yet? I guess I'm waiting to see what kind of software and hardware will be available for the machine in the near future. Like the Mac in its infancy, there exists almost nil software for music or other applications that I'm intersted in. I'm also waiting to see how NeXT responds to the user community's needs and criticisms.

The first complaint that most people have with the cube, is the speed of its floptical drive. The thing is just too damned slow to be used as a swap and mass storage device. As a mountable device for auxillary data and backups, it's just peachy keen. I can easily see musicians mounting huge wave tables of sounds and sequencer/synth patch data on the floptical drive for use with their music applications and synthesisers.

The good news is that NeXT seems to have recognized this deficiency and has added a 40 megabyte swap drive and at no charge to the customer! This, and the lowering of memory upgrade prices is a WONDERFUL step forward and shows NeXT's commitment to maintaining a quality product. Of course, access to the cube's file system on floptical will still be slow, but this patch helps overall throughput a lot.

One wonders, however, why a 330 megabyte hard disk wasn't made a part of the standard configuration instead of the floptical in the begining though. This is the question many people have. The answer (in my humble opinion) is that Steve Jobs wanted an inexpensive machine for education and thus the floptical seemed to be the least cost, high density media available that would also allow numerous users access to a single machine. It's encouraging to note that NeXT also offers a "network" version of the NeXT computer which does not include the floptical. From there one can presumably add the hard disk of his choice.

Speaking of disks, why doesn't NeXT offer a 1.4 megabyte Super floppy disk with the machine and driver software that will let it read and write floppy disks (particulary those that a Macintosh can also read and write!)? Perhaps NeXT is relying on 3rd party vendors to offer such peripherals. But why rely on third-parties? Why can't the user cobble together a driver and just plug in one of the off-the-shelf industry standard peripherals? Sounds like a good rainy-day project for some industrious Unix hacker. Drivers are easy to write and install in Unix systems when you understand the device completely (testing drivers is extremely difficult).

Source code seems to have been a sensitive issue recently. A lot of people are flaming NeXT because Jobs refuses to release source code. I can understand both sides to this view. To change the functionality of the kernel (not just add to it) you NEED source code. A lot of Universities maintain their own operating systems because they can not only correct bugs faster by doing it themselves, they can add functionality to the kernel. These changes may then be distributed to other users. A lot of institutions simply CANNOT wait for a vendor to deliver his sanctified patches... if he even gets around to delivering them at all.

I'm not suggesting that NeXT abandon software maintenance or distributions though. I am suggesting that a certain amount of course code should be distributed to the user's for modification if the user so desires. NeXT should and must remain the sole source for "official" distributions of software for NeXT computers. This centralized position should NOT exclude "do-it-yourselfers" who have the where-with-all to tinker with their machine's software however.

Now frankly, I don't know what source code is available or not ... I haven't heard the latest pronouncements from NeXT. I do know that it is MUCH easier to develop applications in an environment when you know EXACTLY how the software operates (not just what the specs say). Example driver code would be a very useful start.

So I'm suggesting that a base of software for the machine will grow if adequate technical specs and source code examples are provided. NeXT needn't be afraid of giving away it's considerable investment in software. On the contrary, hording software is an effective method for isolating yourself from your customer base. If NeXT wants to make software development easy and effecient on the cube then providing example source code and technical specs will facilitate a more powerful development environment than any automated software development tool they deliver.

In contrast to the cube, the Macintosh is rather nice in that you can change a lot of the functionality of the machine through the use of CDEVS, INITS and DAs and at a user level! The concept of a "resource" is also a useful one since that means you can extract modular pieces of "stuff" from one application and plug them into another (like window layouts, sounds, code etc). I'm not sure how one can "hook into" existing programs on the NeXT like this... if at all.

Something NeXT users REALLY REALLY need is an electronic network all their own. Imagine hundreds of NeXT computers connected across the country via their own 9600 baud modems. User's could exchange news, mail, source code and gateway to other networks across the country (Usenet and the Internet in particular). With such an informal user electronic network, user's could help themselves, and thus reduce the burden from NeXT. NeXT would also benefit since user bug fixes and suggestions could be forwarded to NeXT for inclusion into later releases.

Without an active and enthusiastic community of users AND programmers, the machine will most likely die.

Speaking of sharing software, those flopticals are a mite on the expensive side for distributing even $100 per unit software. The Super floppies would make a MUCH more economical distribution media. This would reduce costs for vendors but more importantly it would mean USERS could share software more readily. I'm certainly not going to exchange software on my $50 floptical and then risk trashing my disk drive mechanism because the floptical was damaged in transit.

---- To be continued ----

John T. Nelson
UUCP: sun!sundc!potomac!jtn
Advanced Decision Systems
Internet: jtn@potomac.ads.com
1500 Wilson Blvd #512; Arlington, VA 22209-2401
(703) 243-1611

I love music that sounds like a Conrail locomotive careening headlong into a truckload of Harpsichords.

FTP & BITNET on the NeXT

Herri Gunawan

I just joined BuzzNUG after learning about the NeXT Computer Newsletter. So I wrote this tiny article just as an additional note.

I would like to talk about the FTP (File Transfer Protocol) that allows us to send/get/put files among UNIX-based systems.

For example, say Michael created a file by vi editor in hydra, and now he wants to edit the file by NeXT WriteNow word processors. One important thing for file transfer is that the file must be in ASCII format for better results, otherwise the sent files can not be edited.

To use the FTP, click your terminal or Mach icon.

When the prompt % showed up, type *ftp destination_system*

(For instance: *ftp hydra*)

To list the files in that system, type *dir <enter>*

When you get the filenames you want to process, type:

- *get <filename>* to import the file from hydra to NeXT
- *put <filename>* to export a file from NeXT system to hydra

I have tried to import and export files among NVE1, NeXT, hydra, and Apple A/UX-based system, and all of them works beautifully if files are in ASCII modes.

Another note that I can give is about the nodes of BITNET networking system. Anybody who interested in BITNET nodes (complete listing), can link to my file at: gt0809d/network/bitnet.wn

To use this nodes, use a typical E-mail command but put an BITNET after the address, for instance:

%mail femgs0100@wismac1.BITNET

(femgs0100 is userid, wismac1 is a BITNET node)

Okay, here are some notes from me. In the next edition, I'll write about WriteNow, and other features. See ya!

Herri Gunawan, GT0809d@cadnext8.gatech.edu
*[NeXT lists 3 "official" FTP sites for common use : at Maryland, Purdue and Oregon. I have found that Purdue and Oregon have the best collections of NeXT resources -- what little there is out there. Hopefully with a lot of enthusiasm these sites will start to build up. They are : j.cc.purdue.edu, cs.orst.edu and umd5.umd.edu. Also note that ftp uvaarpa.acc.virginia.edu is starting to become active (more than Maryland, I'd say!). From experience, I've almost never had any difficulties with binary file corruption in ftp. Keep in mind, of course, what file format is being transfered. On matters similar: I like to say **prompt** early in the session to turn off interactive prompting and then use "**mget**"s in the background. Another hint is to use **ls flags filespec** to save the remote listings into a local file. Those things scroll by very quickly and the NeXT terminal program has no cut/paste or scrolling capacity. Boy, can't I wait for a really good Terminal program. -- E. Liebman. p.s. Remember to practice Download-Safety!!]*

Connecting a Hayes Compatible Modem to the NeXT
Erica Liebman

FRANCIS BEAUMONT 1584—1616 and JOHN FLETCHER 1579—1625
It would talk:
Lord how it talk't!

Getting that Hayes-like modem up and initially running on the NeXT is actually pretty simple . What you'll need is :

1. a standard Apple/Mac connection cable.
2. a Hayes compatible modem (of course)
3. a NeXT with super-user priviliges.

The instructions below will describe how to use tip - the terminal interface program. For more help on tip, try *man tip.* (Kermit is also available at several FTP sites and is much easier to use, although documentation is short. Several books about Kermit have been published and can be tracked down at your local library -- or you can call up Columbia

University where Kermit was originally written.)

Tip, as you'll find out is not that great a program -- but it comes free on the machine. With a win situation like that, who can say no. Warning : file transfers in tip are really awful and fail all the time -- even in pure ascii !.

All instructions are aimed at the competent and confident user

.

Here are the first three steps to using your modem :

Step 1 : Connect the cable to the modem and to the serial port "A" (if A is in use, connect it to B and substitute B for A in the instructions). Plug in the modem, turn it on.

Step 2 : become super user and edit /etc/remote. Copy the two lines which start a9600 to another part of the file. (If working with serial B, copy b9600). In THOSE COPIED TWO LINES change all occurences of 9600 to 1200, 2400 or 4800 depending on the speed of your modem. Save the file and leave super-user.

Step 3 : Open a terminal and type "*tip a1200 <enter>*" (or a2400, a4800, b1200, etc. You get the idea). You should now be able to type your AT commands. (Try AT and see if you get the OK response). Connect as usual.

leaving tip : to leave tip, use a "tilda followed by ^D" combination.

When things go wrong : generally things go wrong because there is a lock on the tty port (ttya or ttyb). Go super-user. Change directory to /usr/spool/uucp and look for a lock file that has the name "LCK..ttya" or "LCK..ttyb" in it and delete the lock file. (Remember that comment before about the confident and competent user!).

Making your life easier : You can set up your .tiprc file and /etc/remote files for automatic phone dialing. This is left as an exercise for the student.

Error Handling in Objective-C

by Brian Skinner, Roger Rosner, and Jonathan Schwartz

Introduction

What error drives our eyes and ears amiss?
—William "Rapmaster" Shakespeare, The Comedy of Errors

Granted, error handling in Objective-C programs has never been the most exciting topic. Nonetheless, efficient error handling can simplify the development and maintenance process for software engineers while clarifying error conditions for users. In this article, we'll try to elucidate the techniques we are using at Lighthouse Design in a medium sized (50,000 line—60 class) electrical schematic entry tool.

Obviously, the extent to which developers need to worry about error handling varies with the potential for error conditions in the final product. In small programs there's not much point in doing any sort of tricky error handling. If you need to let the user know about an error, you can often get away with simple **printf()** statements, such as:

```
if (file_error_condition)
    printf("I/O error.  Can't open file.\ n");
```

A slight variant of this embedded technique is the macro technique:

```
#define CANT_OPEN_FILE "I/O error.  Can't open file.\n"

if (file_error_condition)
    printf(CANT_OPEN_FILE);
```

On larger projects, where error messages can number in the thousands, this sort of error handling is obviously inadequate. There are many reasons why large projects require good error handling, some of which may be apparent in the following anecdote.

> I once worked on a 600,000 line system with several thousand error messages. Most of the error message text was simply embedded in the code, though some was in macros. The messages were short, and most of the associated explanatory text was available in an appendix to the printed manual. Every time we released a new version of the system (more than once a year), the printed documentation was updated to reflect changes to the code. This meant going through the code "by hand" and identifying all of the new messages, the changed messages, and, worst of all, the absence of the recently removed messages.

Clearly, using an error handling methodology would have saved a great deal of time and lead to more accurate documentation. Because of convenient encapsulation and code sharing, writing an elegant and convenient error handling system is relatively easy in Objective-C. We believe the techniques that follow will increase the maintainability of larger bodies of code.

Even in an object-oriented program on the NeXT, one is tempted to report errors using **NXRunAlertPanel()** "on the fly" (see Figure 1). The Draw program distributed with the NeXT makes consistent use of this technique.

```
#define Notify(title, msg) NXRunAlertPanel(title, msg, "OK", NULL, NULL)
- checkForFile {
   if (file_missing)
       Notify("Open Draw Document", "I/O error.  Can't open file.");
}
```

Figure 1. Use of conventional error handling on the NeXT.

In all fairness, the size of the Draw program doesn't warrant anything more sophisticated. Our schematic entry project, however, is considerably larger than Draw and deserves more involved error handling.

Purposes and Goals

In designing the error handling methodology for our project, we began with two purposes:

- To present warning and error messages to the user.

- To pass error conditions between methods.

Note that we did not intend this facility to be used as a general dialog panel management class, though such a concept might well have merit. This restriction allows us to limit the complexity of our user interaction. Furthermore, performance was given only minor consideration. For the most part, error handling is neither highly dependent on speed, nor given to taking large amounts of storage.

As a secondary consideration, we endeavored to make our error handler capable of presenting rich error messages—one-line explanations provided by simple dialog boxes are generally of little use and often lead to confusion.

With this in mind, we defined three goals for our error handler:

- To present a consistent reporting format to the user (and allow easy modification of the format).

- To encapsulate error messages with the classes that raise them.

- To make the error handling code as re-usable as possible.

To further preface our discussion, it is important to note that we did not endeavor to emulate any of the complex error handling behavior of languages like Ada, CLU, or Common Lisp. Our main purpose was to alert the user to the existence of problems encountered during program execution, rather than to provide any sort of program control transfer facility.

Solution

Appropriately enough, we began our solution by creating a class named **Error.** The methods of **Error** generally fall into one of two categories: registration and presentation.

Registration. Registration methods are used to inform the **Error** class of the existence of error messages. One very common usage is to put a series of registrations in the **+initialize** method of a class (see Figure 2). Note that the compiler-defined macro **__FILE__** expands to the file name in which the source is contained.

```
+ initialize
{
    [Error registerWithOwner:__FILE__ key:"cantOpenFile"
        message:"I/O error.  Can't open file %s.\n"
        help:"The file you've chosen is not available.\n"
             "Please choose another file name."
        sound:[Sound findSoundFor:"Cant_open_file"]
        header:"Error Writing To File"];
    [Error registerWithOwner:__FILE__ key:"printerOutOfPaper"
        message:"Your printer is out of paper."
        help:"The printer to which you are trying to print is out of paper.\n"
             "Follow these directions to load paper in the printer's \n"
             "paper cassette.\n"
             "  1. Remove the paper cassette from the printer. Pull firmly.\n"
             "  2. Remove the cover from the paper cassette.\n"
             "  3. Insert a stack of paper into the cassette, being careful\n"
             "     to put the tip of the stack under the paper overflow tab\n"
             "     in the corner of the cassette.\n"
             "  4. Replace the paper cassette cover.\n"
             "  5. Slide the cassette back into the printer."
        sound:[Sound findSoundFor:"PrinterOutOfPaper"]
        header:"Printer Error"];
    return self;
}
```

Figure 2. Registering errors in the **+initialize** method.

The **registerWithOwner:key:message:help:sound:header:** method passes a good deal of data to the **Error** class. The **owner** and **key** parameters are used later to retrieve and/or present an error instance, and uniquely identify it. We found it useful to follow the convention of implementing a class (**Foo**) in a single pair of files (**Foo.m** and **Foo.h**), and then use file names as our owners. Thus, errors with identical keys but different classes can be used in the same program. The **message** parameter is a **printf()** compatible string that appears in an alert panel, and the **help** parameter provides a rich explanation of the error to users requesting more information. Optionally, one might replace **help** with **helpLink**, a hypertext-style link to an explanation. The **sound** parameter offers yet another way to enrich the feedback with an audible report of the error. Lastly, the **header** parameter is used directly as the title parameter of **NXRunAlertPanel()**.

To register a large number of errors at once, it seemed more convenient to place the error descriptions in a separate file, and then register the file with the **Error** class. The error class then dynamically loads the errors, rather than immediately allocating the objects (see Figure 3). Needless to say, the file containing the error descriptions must be available at run-time.

```
+ initialize
{
   [Error registerWithOwner:__FILE__ fromFile:__FILE__ ".error"];
   return self;
}
```

Figure 3. Registering a file of errors in the **+initialize** method.

The file registration technique is somewhat similar to Macintosh resources.

Presentation. To display an error in an alert panel, the factory method **showErrorWithOwner:key:** (see Figure 4) can be used. As noted above, the **message** parameter can take **printf()** style formatting characters, with any necessary arguments listed at the end of the **showErrorWithOwner:key:** parameter list.

```
[Error showErrorWithOwner:__FILE__ key:"cantOpenFile", inputFileName];
```

Figure 4. Showing an alert box.

It is also possible to create an error instance without immediately displaying it, allowing their use as return values or parameters. Additionally, errors can be kept around for record keeping purposes, e.g., the most recent error. To allow for this, we created a similar factory method, **getErrorForOwner:key:**, that returns an instance of a specific error, without alerting the user to its existence. Repeated calls to this method with the same **owner** and **key** parameters return pointers to *the same* instance, thus minimizing memory requirements. To display an instance of an error, one sends it the message **showError** or, if it needs arguments, **showError:**.

The **done** method is used to inform an error instance that it is no longer needed by a section of code. The error instance can keep track of "live" references to itself, and, should that number ever reach zero, deallocate itself. In systems with large numbers of errors, this ability to remove error objects from memory can be an absolute necessity.

A plausible example of an **Error.h** file is given in Figure 5. The implementation of such a class is a relatively trivial exercise, though one should keep in mind that the use of "class variables" is necessary. Since Objective-C does not explicitly provide for class variables, they must be simulated with static variables in the class file. (Remember: unlike true class variables, these statics cannot be inherited.)

```
//*********************************************************************
// File Information
//*********************************************************************
// File: Error.h
#import <objc/Object.h>

@interface Error : Object
//*********************************************************************
// Instance Variables
//*********************************************************************
{
    const char *owner;   // Preferably the class name
    const char *key;     // Unique within an owner
    const char *message; // Can contain printf-style formatting
    const char *help;    // The help text
    Sound      *sound;   // A sound for the error
    const char *header;  // "Error", "Warning", "Message", etc.
}

//*********************************************************************
// Class Methods
//*********************************************************************
+ registerWithOwner:(const char *)theOwner key:(const char *)theKey
  message:(const char *)theMessage help:(const char *)theHelp
  sound:(Sound *)theSound header:(const char *)theHeader;
+ registerWithOwner:(const char *)theOwner fromFile:(const char *)theFile;
+ showErrorWithOwner:(const char *)theOwner key:(const char *)theKey, ...;
+ getErrorForOwner:(const char *)theOwner key:(const char *)theKey;

//*********************************************************************
// Instance Methods
//*********************************************************************
- showError:(int)argCount, ...;
- showError;
- done;

@end
```

Figure 5. The **Error.h** file.

Conclusions

Though the **Error** class technique is an effective and elegant alternative to embedded error text, it is certainly not the ultimate solution to the error handling problem. One deficiency results from the removal of error messages from the main body of the program: the code becomes less readable. Furthermore, adding error messages requires the use of a somewhat alien syntax. These factors can lead to slower programming or, worse yet, a broken train of thought. In systems with a truly enormous number of errors, the performance of the **Error** class may become a more serious problem—looking up an error in a table of thousands may be slow, and the table itself may consume a great deal of storage space. Yet another set of limitations stem from the **Error** class's lack of generality. It cannot present anything more than simple informational dialogs, nor can it return any complex information from the user.

Nonetheless, we find that the use of this technique has significantly cleaned up our code, facilitating its rapid evolution.

The authors of this article, Brian Skinner, Roger Rosner, and Jonathan Schwartz, are partners in Lighthouse Design, creators of electrical engineering CAD tools for the NextStep environment. They can be reached by electronic mail at lighthouse@lighthouse.com or ...!uunet!lighthouse!lighthouse, by physical mail at 6516 Western Avenue, Chevy Chase, MD 20815-3212, or by telephone at 301-907-4621.

Feedback from the Trenches

Erica Liebman & the entire BuzzNUG membership

Here's what you all said that you'd like to see or know.

...I'd be interested in hearing what info you find on this (*porting GNU to the NeXT)* If possible, I'd like to add any needed diffs/caveats to the NeXT archives to keep thousands of people from doing the same thing over and over. *Gerrit Huizenga NeXT Workstation Coordinator/ Maintainer of the Archives Purdue University Computing Center gerrit@cc.purdue.edu*

The trick is to define __STRICT_BSD__ (usually in the CFLAGS of the Makefile) This makes the declarations in header files BSD4.3 compatible (and less ANSI-C). Beyond that most things are trivial. I have also ported CNews, rn, and other Unix utilities. I will be attempting the g++ port in the next week; I suspect this will be more difficult as g++ is synchronized with gcc and gdb releases (and NeXT has their own version of gcc and gdb to which I will not have the sources for another month or two). Moreover, g++ will be of limited use since there are no NeXT appkit (nor Mach) libraries for it. The strict typing and static binding of g++ makes an interface with appkit a non-trivial task. *Edward Jung DTG uunet!dtgcube!ed* [I tried all this, but still couldn't compile any GNU stuff. Will keep on trying and keep you informed -- EJL]

...Any helpful hints that such a group could generate concerning the pitfalls or programming the NeXT would be greatly appreciated by those of us attempting to develop NeXT software. *Craig Henderson OCLC Office of Research*

...We have 4 machines. we'll do anything for software! *Peter Flur*

...We haven't really used the NeXT because it's a "NeXT" yet, simply because we don't have anyone that knows interface builder or objective-c *Peter Flur*

... Are there any publications that cover the NeXT? *Jeff Scott* [MacWeek occasionally will cover the NeXT -- ejl]

...One of the activities that should be promoted by this users group {should it ever get off of the ground :) } would be the dissemination of "objects" for IB. There are a few on the archives, but very few of the "gee whiz" type. The system admin stuff is boring, but nonetheless necessary and I see that as an additional area of emphasis. Other than that, I'd like to see to what extent the cubes are being used for in terms of its sound and graphics capabilities. Good luck with more responses. *Frank M. Guerra Lawrence Livermore National Laboratory guerra@lll-crg.llnl.gov*

...The idea "how-to" articles is good-- I'd like to see that. *Beth E. Binde binde@rutgers.edu*

...I'd like to see software archives full of good stuff. I'm afraid I wouldn't have much to contribute though, except maybe reviews of some of that good stuff. *Rob rsteele@XN.LL.MIT.EDU*

...Basically I'd use it to ask others how they've solved particular programming problems. Probably mostly questions of style. *William Shipley UW undergrad NeXT guru*

...Topics of current interest to me. 1) Hooking up a Mac-SE to the NeXT so that Write-Now files can be shared (and hopefully graphic files later). Interface-Builder hacking on the NeXT. *Solovary@math.berkeley.edu*

...I **think we'd be interested in reading just about anything technical. Example source is always welcome. Reviews of third party products would be nice. Information about what other people are doing (both academic and commercial) is good too. We're interested in details and not particularly in rumours. It would be a relief to read some information on the NeXT we don't already know...What are the newsletter's goals? Is it going to be very technical? How long will it be? What is your target audience? How much will it cost? (I'd really like to see something on the order of MacTutor.)***Roger & The Guys, rock@lighthouse.com*

...Please put me on your mailing list for the BuzzNUG news letter, also let me know if you get the "deck of cards" object. *Jeffery E. Piper jep@cl-next1.cl.msu.edu*

...The newsletter should be a source of the latest information on news-breaking NeXT topics (just like any newsletter would be). Maybe you could have different columns, like, Public Domain Programs, New Products/Peripherals, Bug Reports, SystemAdmin, Networking with NetInfo, etc. You could also try and solicit articles for your newsletter from other NeXT Experts. There are many more obvious other things you could do to make your newsletter exciting and popular. It would be nice to have a reliable and formal source of information on NeXT topics. It's really needed. I'm looking forward to seeing your first issue! *Salvatore Saieva Queens College, N.Y. Academic Computer Center*

...We're interested in a NeXT users group. Currently, we're trying to impliment speech synthesis on the DSP chip and we are finding it difficult with the lack of documentation. Basically, what we would like a user's group to do is to be an hub for information passing. Since we started work on the DSP we've found some problems, tricks, etc that we would be willing to pass on. Any bug reports or how-to reports would be great! How is the users group going so far? Has there been much response? *Craig Schock Man-Machine Lab, University of Calgary. schock@cpsc.UCalgary.CA*

...[>*Want to join BuzzNUG*?] Sure, as long as it doesn't cost anything. *Keith*

Object Bee-Line

I'm hoping that the **Object Bee-Line** is going to be a regular feature of the Buzzings, where people can post "Object Want-Ad"s. Since this is the first issue, this will be obviously abreviated.

Wanted : "Deck-Of-Cards" object that has the 52 icons, accepts the messages sort and deal. -- contact erica@kong.gatech.edu.

Wanted : "Scrolling Text-File Object" that will act as a window into a very large text file -- contact erica@kong.gatech.edu.

A Demo Program using external Sound Resources

Erica J. Liebman

I was inspired to write a demo that could show off the sound capabilities of the NeXT, yet be something every so slightly more than a tape-recorder or Max Headroom demo (cut/paste/cut/paste "H-h-hello there"). The requirements were such : something quick that I could put together and compile ahead of time, but which would allow me to change the sound resources quickly.

Guessing Program

What I came up with is a simple guessing program. (You know it -- pick a number between 1 and 100 and I'll tell you if it is too high or too low). The main advantages were simplicity, familiarity and encapsulation. First, the program was very easy to write. It took just a few lines of code beyond what Interface Builder created. Second, almost everyone has played a game like this at some point in their lives. By showing something familiar, that everyone could identify with, I was able to highlight the NeXT specific features. Third, it is very limited in what it attempts to do. The complete encapsulation of the guessing metaphor is sleek (I think) and doesn't contain a lot of distractors.

External Sound Resources

I decided to go with external sound resources because the file could be compiled and the sound resources recorded later at demand. The resultant compiled Application would be much smaller as well. The disadvantage here (of course) is that before running this program, you have to create four *.snd* files in the same directory as the application. I've found, though, that plugging in the microphone and asking somebody (or somebodies) to speak in the four phrases is found to be "enjoyable" especially when getting the immediate feedback of listening to the playback.

These are the four sound files required by the Guesser program and my suggestions for the phrasing.

instructions.snd : "I'm thinking of a number between 1 and 100. Can you guess what it is? I'll tell you if you are too high or too low."
too_high.snd : "Your number is...too high."
too_low.snd : "Your number is...too low."
just_right.snd : "You guessed the number correctly. Let's play again."

The Guesser Object

I took several steps in using these sound files. First, I created four instance variables for my Guesser object, each relating to a different message. I also added on a pointer to a textfield for input and an associated integer to keep track of the real value of the number to be guessed. The Guesser object accepts two special action messages of "guess" and create a "new game".

```
/* guesser.h -- object definition file
   for the Simple Sound Demo Program */

#import <objc/Object.h>
#import <appkit/appkit.h>
#import <soundkit/Sound.h>

@interface Guesser:Object
{
   int real_num;
   id  too_low;
   id  too_high;
   id  just_right;
   id  instructions;
   id  input_textfield;
}
+ new;
- setInput_textfield:anObject;
- newGame:sender;
- guess:sender;
@end
```

To read the sounds into my instance variables, I altered the +*new* method. Each sound i.v. was initialized using a *[Sound new]* call then read in by *[i.v. readSoundfile:"pathname"].* So long as the sound files conformed to the pathname, the actual data could be altered after compile.

Using the Textfield

I strongly object to having to press return entering data into a single data field. Similarly, I feel after data has been entered, it is up to the program, not the user, to highlight that data so the next data to be entered will overwrite the last value. Therefore after each data check, I included a *[input_textfield selectText:self];* message. I find this to be somewhat of a crusade with me. Sermon given, I shall move on.

Setting up Guesser

Getting Guesser running for yourself should be relatively simple. Do the following.

1. Copy the source into a fresh new directory. Name as Guesser.h and Guesser.m. (keep the capital G).

2. Start interface builder, create a new Application & save immediately into that fresh new directory.

3. Add a nice new project into the same fresh new directory.

4. Drag 2 buttons and 1 text into the "MyWindow". Rename the buttons to "Guess" and "New Game", get rid of the word "Text" from the textfield. Make the window look pretty.

5. Under Project, select the .[hm] files and *Add* Guesser.[hm]. (You should not get a copy message if you've done everything in the new directory with the source code.)

6. Click on My Window. In the Attributes inspector, change the name to Guesser and kill the resize, miniaturize and close options. Don't forget to press "OK".

7. Open the Classes (suitcase with .h) browser. Move all the way to the left and click on Object. Do a *subclass* operation. Rename subclass1 to Guesser. (Don't forget to capitalize **G**uesser).

8. Do an operations *parse*. After about 15-30 seconds, an Alert panel will appear saying *Parsed class has different actions or outlets*. Click on Replace.

9. Do an operations *instantiate* and a GuesserInstance will appear in the "suitcases" application window. Close up the class browser and click on the suitcases window.

10. Connect the GuesserInstance to the text field using a control-alt-drag. The inspector will automatically change to the connections inspector. Click on *input_textfield* and press the Connect button. A connection dot will appear to the right of input_textfield.

11. Starting with the Guess button, control-alt-drag *to* the GuesserInstance. Click on *guess:* in the connections inspector window and press the Connect button. The connection dot should appear.

12. Do the same for the New Game button.

13. Save and leave interface builder. Start a Mach session. Change directories to the new directory you've been working in.

14. *make* the file -- it should compile correctly and quickly.

15. From the browser, start up /NextDeveloper/Demos/SoundPlayer. Record the four sounds as described above. Save each with the **exact name** in the new directory you've been working in.

16. Run the program & play.

17. Problems or questions : write me at erica@cadnext5.gatech.edu **if desperate**.

Source for the *.m* file

```
/* Simple Sound Demo Program */
#include <time.h>
#include <stdlib.h>

#import "Guesser.h"

@implementation Guesser

+ new;
{
      self = [super new];

     /* init random number */
     srand(time(0));
     real_num = rand()&100;

     /* init sounds */
     too_low = [Sound new];
     too_high = [Sound new];
     just_right = [Sound new];
     instructions = [Sound new];

     /* load sounds from files */
     [too_low readSoundfile:"too_low.snd"];
     [too_high readSoundfile:"too_high.snd"];
     [just_right readSoundfile:"just_right.snd"];
     [instructions readSoundfile:"instructions.snd"];

     [instructions play:self];

     return self;
}

- setInput_textfield:anObject
{
    input_textfield = anObject;
    return self;
}
```

```
- newGame:sender
{
    real_num = rand()&100;
    [input_textfield selectText:self];
    [instructions play:self];
    return self;
}

- guess:sender
{
     int the_guess;
     the_guess = [input_textfield intValue];
     if (the_guess < real_num)
     {
         [too_low play:self];
         [input_textfield selectText:self];
     }
     else if (the_guess > real_num)
     {
         [too_high play:self];
         [input_textfield selectText:self];
     }
     else /* the player guessed correctly */
     {
         [just_right play:self];
         [input_textfield selectText:self];
     }

    return self;
}
@end
```

Scenes from the NeXT Issue

- I will write on creating and installing application icons. I'll be using the "Converter" lab as the demonstration.
- Dave Rosenbaum promises an article on how to open and adjust your monitor to a paper crisp delight to please even the most discerning eyes.
- An article by Andrew Stone, developer of TextArt about the development thereof -- *This was meant for this issue, but the NextAttachments line in Mail just wasn't working through other machines -- Suns just can't handle it. I got the text of the article, but the drawings were corrupted!*
- Review of "TextArt" (if I can get my copy from Andrew Stone in time).
- Open topic for comments/articles. : "Why is Object Oriented Programming such a good match for User Interface programming?"

The NeXT issue will be out approx 3rd week of February, so get in your articles by the 1st week of February or (preferably) the last week of January.

Buzz's Hint Corner

Buzz's Hint for December/January is this :
When in WriteNow, use the ***Alternative*** *button when clicking on the up and down arrows to move a page at a time.*

(Submissions for Buzz's Hint Corner are Welcome)