Monitoring Shared memory: A Toolset Prototype for the INFORMIX OnLine Administrator

Eric Gronholz

College of Saint Benedict/Saint John's University

Follow this and additional works at: https://digitalcommons.csbsju.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.csbsju.edu/honors_theses/343

Available by permission of the author. Reproduction or retransmission of this material in any form is prohibited without expressed written permission of the author.
Monitoring Shared Memory:  
A Toolset Prototype for the INFORMIX OnLine Administrator

A THESIS
The Honors Program
College of St. Benedict/St. John's University

In Partial Fulfillment
of the Requirements for the Distinction "All College Honors"
and the Degree of Bachelor of Arts
In the Department of Math/Computer Science

by

Eric B. Gronholz
May, 1993

Lynn R. Ziegler, Advisor
Monitoring Shared-Memory
A Toolset Prototype for the INFORMIX OnLine Administrator

Approved by: [Signature]
Lynn R. Ziegler, Associate Professor of Computer Science

Department Readers:

(1) ____________________________
Noreen L. Herzfeld, Associate Professor of Computer Science

(2) ____________________________
J. Andrew Holey, Associate Professor of Computer Science

(3) ____________________________
Michael R. Ross, Computer Science Department Chair

(4) ____________________________
Margaret L. Cook, Honors Program Director
# Table of Contents

Abstract 6  
Contents of Figures and Listings 7  
Acknowledgements 8  
Introduction 9  
The X Window System 10  
INFORMIX OnLine and \texttt{tbstat} 15  
"Monitor" Program 19  
Improvements 29  
Conclusion 30  
Bibliography 31  
Appendices 32
ABSTRACT

The "Monitor" Program was developed to assist system administrators for the INFORMIX OnLine Database Server with their evaluation of shared memory components used on their system. Using the Open Software Foundation (OSF)/Motif Widget set for Xt Intrinsics on the X Window System, this program creates scaled dials and pop up windows to display system information at regular intervals. The program is surprisingly simple to operate and is general enough that with just a few modifications, it can be used to monitor any system information, not just shared memory information for OnLine. This program marks a new way to look at systems analysis, and could be the building block of many powerful programs of the future.
Contents of Figures and Listings

Figure-1  The Client/Server Model  13
Figure-2  Complete X Window System  13
Figure-3  Widgets  15
Figure-4  Two Process Architecture  17
Figure-5  INFORMIX-OnLine Shared Memory  17
Listing-1  Three Processes of Program  20
Listing-2  tbstat Output  22
Figure-6  The "Monitor" Program  25
Figure-7  Creation of Child Process  27
Acknowledgements

I would like to thank Joe Foley and the rest of the people at Systems Migration, Inc. for the opportunity to create a product for them.

I would like to thank my friends and family who put up with some unnerving times throughout the course of this project.

I would like to thank the faculty members on my thesis committee for their support throughout this project.

Most of all, I would like to express my deepest thanks to Lynn Ziegler. Not only did he act as an advisor and instructor, but sometimes took on the role of counselor, and certainly became a friend.

Thank you to all of you, for without any of you this project never would have been completed.

Sincerely,

Eric B. Gronholz
May, 1993
Introduction

As the need for faster and more accurate access to information increases, so does the responsibility of system administrators to keep computer systems and database packages operating efficiently. This responsibility presents administrators with many challenges, but one in particular is their accessibility to information concerning certain shared memory structures. Often the information they need to evaluate how well the system is utilizing its resources is hidden in long and cluttered files. These files can be confusing and extremely time consuming to read. What is needed then, is a program which will show system administrators this information quickly and in an easy to follow format.

"Monitor" is a unique computer application that combines the C programming language with the X Window System to produce a graphical representation of specific shared memory structures utilized by the INFORMIX OnLine Database Server. Through the use of pop-up windows and scaled dials, system information is displayed in a format which makes it easily available to the system administrator. Because structures are independently monitored and displayed, the administrator can isolate potential or current shared memory problems more quickly and easily than normal monitoring allows, thereby facilitating database maintenance and tuning. In short, this tool is useful for evaluating the
system's efficiency and for identifying the kind of activity most commonly executed on the system.

The information displayed by "Monitor" is actually output generated by the INFORMIX OnLine utility `tbstat`. When run at the command line, `tbstat` reads the shared memory structures and reports their contents to the screen at the instant the utility is run. It should be noted that "Monitor" was developed on an SGI (Silicon Graphics) Indigo R3000 machine running the Unix operating system. This system did not have INFORMIX OnLine installed, however, so the `tbstat` utility was not available. To counter the problem of needing this utility's output to develop and demonstrate "Monitor", I ran the utility on a system supporting OnLine and redirected the output to a file. I then wrote my own `tbstat` program which simply lists the file containing actual `tbstat` output to the screen. Hence the program runs as if it were on a system which supports the utility, allowing for demonstrations of the program on any X supported system. A more technical explanation of my `tbstat` program will be given when the three processes of "Monitor" are described.

Although "Monitor" was specifically written to display system information for the INFORMIX OnLine Database Server, it is actually a very general program. This statement may sound contradictory, but it is true. "Monitor" resulted from a need to display OnLine system
information in an easy to read fashion. The program's design, however, allows it to display any kind of desired information with relatively few modifications to the display portion of the program. The main work involved in using this program with other system information comes in writing a program which retrieves the system data and puts it in the correct format, not in changing the way the data is presented graphically. This quality should make "Monitor" a very attractive product to all kinds of system administrators, since the same overall design works for many kinds of system monitoring tasks.

The X Window System

To display any kind of system information, "Monitor" uses the X Window System, in particular the OSF/Motif widget set. Douglas A. Young, in his book *The X Window System*: Programming and Applications with Xt, OSF/Motif® Edition, explains that the X Window System is an industry-standard software system that allows programmers to develop portable graphical user interfaces. This portability means X programs which display windows, text, graphics, etc., can be designed and run on any hardware which supports the X protocol, regardless of that hardware's operating system. In other words, X is device-independent: a program designed on one kind of hardware can be displayed on a different type of hardware without needing to be recompiled or relinked (Young, 1).
The X Window System implements device-independence through the client/server model. This model allows an application to be created and run independently of the process which does the X work (see Figure-1). The X server process handles all input and output devices, and creates graphics, windows, and text as it receives requests for them from the client applications. Once the requests are received and the graphics created, the server sends its work out to the client application where it can be viewed. As Figure-1 shows, the client and the X server need not be on the same machine for the X program to work. The devices need only be linked by a network using the X protocol. This feature is what makes X such a practical user interface development tool.

The flexibility given to clients to create their own user interface does not come without responsibility, however. Since clients can develop everything from the way windows are handled to the shape of push buttons, they are required to account for every detail necessary to create an X application. This requirement means a lot of extra work for the client because developing applications in pure X is a very difficult and tedious process; even the simplest program managing windows would require hundreds of lines of code to accomplish its task. Therefore, various programming groups have developed libraries containing certain X functions which applications can use to do X programming work without having to be concerned with the minute details of X. The most
Figure 1. The client-server model.

(Young, 3)

Figure 2. Programmer's view of the complete X Window System.

(Young, 12)
common of these libraries is the C based Xlib library (see Figure-2), which defines an extensive set of functions providing complete access and control over the display, windows, and input devices (Young, 11).

Although these functions are useful in bringing X to a slightly higher level of programming, Xlib is still a very low-level interface. Programming in Xlib, although done much more often than programming in pure X, is also very difficult and tedious. Most X programming is done using some type of standard toolkit which raises the programming to a higher level. "Monitor" uses the X Toolkit, which is made up of two layers: the Xt Intrinsics and one of many sets of functions used to develop graphical objects called widgets, the Open Software Foundation (OSF)/Motif widget set. Figure-2 shows how this toolkit relates to the Xlib interface and the X server. The toolkit, through widget sets, allows programmers to implement user interface components, such as popup windows, push buttons, and menus with relatively few lines of code (see Figure-3).

The OSF/Motif widget set does not directly support the dial widget shown in Figure-3c and used in "Monitor." The dial is actually a widget created by Douglas Young, who has given permission to users of his book to implement and use it as they please. "Monitor" uses the dial widget to display the current status of different shared memory structures, changing the position of the dials' indicators as the different
FIGURE-3 Widgets

Figure 3a. The XmMessageDialog displaying help messages.

(Young, 121)

Figure 3b

Figure 3c. A Dial widget.

(Young, 90) (Young, 344)
values change.

**INFORMIX OnLine and tbstat**

INFORMIX OnLine is the system administrating process used to control the INFORMIX relational database. Its architecture is similar to that of the X Window System's in that it uses the client/server model for its access to the database. Figure-4 gives a representation of how this particular client/sever model works. Application programs, usually written in relational database languages like INFORMIX-4GL, make requests to the database server through Unix pipes either to change information in the database or to retrieve something from the database. The database server accesses the information requested and returns the data or a notice that the change to the database failed or succeeded. In either case, the server has sole access to the database, just as the X server process has access to the X code needed to create the requested graphics (INFORMIX, 1002-4).

Shared memory on this server refers to the usage of the same memory segments by more than one OnLine user process, thereby allowing processes to access copies of the same database information. By using copies of the database tables, OnLine runs more efficiently than systems where data is privately stored for each user. For example, database table information can still be stored on multiple disks to reduce data storage limitations, but disk I/O is also reduced because
**Two Process Architecture**

Application programs, sometimes known as clients, communicate with the database server over UNIX pipes. The database server accesses the databases. This is generally known as a client/server or "two process" architecture.

![Diagram showing the relationship between an Application Program, UNIX pipes, a Database Server, and Database(s).]

**FIGURE 4**
(INFORMIX, 1002-4)

---

**INFORMIX-OnLine Shared Memory**

- **CHunks**
- **MIRROR CHunks**
- **DBSPACES**
- **USERS**
- **TRANSACTIONS**
- **LOCKS**
- **TBLSPACES**
- **BUFFER POOL**
- **LRUS**
- **BIG BUFFERS**
- **FLUSHERS**
- **LOGS**
- **PROFILES**

![Diagram showing the INFORMIX-OnLine Shared Memory Architecture.]

**FIGURE 5**
(INFORMIX, 1215-6)
commonly pooled buffers are flushed on a system-wide basis rather than for each user. As a result, execution time decreases: only one copy of the data is necessary in shared memory and users can access this memory rather than accessing the disk.

In addition to holding copies of database tables, shared memory is also used to maintain system resources for OnLine. OnLine creates various collections of data structures which are allocated in shared memory. OnLine then stores information in these structures that monitors how the database server is being used. Figure-5 lists the major resources of OnLine's shared memory. The specifics on the different components are beyond the scope of this thesis, but system administrators analyze these items to check the efficiency of the system (INFORMIX, 1715-6). As a result, the values of many of the components in Figure-5, along with several others, will be what the dials in "Monitor" will be displaying.

"Monitor" Program

To display the different shared memory structures, "Monitor" utilizes three independent processes: **dial**, **tbstat**, and **alarm**. The brief outline in Listing-1 shows the basic functions of the three processes. The first process, **dial**, acts as the main process for the program. An analogy which might be useful in understanding how the three processes work together is to think of **dial** as a client and the other two processes
as servers: **dial** runs the actual application program while the other two processes do some kind of work at the request of **dial**.

The first request **dial** makes is to the **tbstat** process. Remember that if "Monitor" were installed on a system which supported the INFORMIX OnLine Database Server, this call would be handled by the actual **tbstat** utility installed on that system. Since this program was not implemented on such a system, I developed a process which simulates what the utility does. By writing my own version of this utility, changing "Monitor" to run on an OnLine system becomes trivial.

The call to **tbstat** comes from within the function "call_tbstat" in the **dial.c** program. This function uses the Unix command "system" which allows for the execution of separate programs from within another program. To execute **tbstat** from within **dial**, we use the following line of code:

```c
system("tbstat > tbstat.info");
```

What this command says is to execute the **tbstat** utility and redirect its output into a file called "tbstat.info". The only syntax that would need to be changed to run this call on a system supporting OnLine is to add a parameter telling **tbstat** what information to print. Furthermore, since this code appears only once in the "call_tbstat" function, only one line of code needs to be altered. Below is how that line of code will look when the actual utility is being used:
THREE PROCESSES OF PROGRAM

**DIAL**

- Run `tbsstat` and save info in a file.
- Collect `tbsstat` info from the file.
- Create dials and display info from `tbsstat` file.
- If dial hits alarm position, launch subprocess to display the proper alarm. Do not display alarm if currently active.
- If help request given, launch subprocess to display proper help window.
- Repeat the above events at specified intervals.

**TBSTAT**

- Generate info just like the OnLine `tbsstat` command does.
- Display a file with correct info to screen.

**ALARM**

- Receives parameter telling it which alarm to print: warning or help. Process is complete when window is killed.
system("tbstat -a >tbstat.info");

If this parameter is added, no other changes would need to be made for the program to run on an OnLine system because the tbstat utility I designed creates the same output as the tbstat -a command. Listing-2 shows the format of this output.

The actual tbstat command, as I have already explained, reads shared memory structures to create its output. My version of the utility could not do this task however. Instead, "Monitor's" tbstat command displays an output file identical to the output generated by executing tbstat -a on a system that supports the OnLine Server. To ensure a wide variety of output which covered all boundary cases of the data created by tbstat, I made numerous copies of this output file and altered the data in each file in a manner which would properly test the "Monitor" program. Therefore, when dial makes the call to tbstat, my version simply lists one of these files to standard output, which dial redirects to another file.

After dial runs tbstat and creates the "tbstat.info" file, it continues by reading through this file to get the information which is to be displayed. The information is collected in an array of structures which holds the same kind of information for each shared memory component to be displayed. Appendix A shows all of the components on which dial
collects information, and includes explanations as to how that
information is manipulated to be put into the structure. The structures
holding this information are as follows:

```c
struct display_scale {
    char ds_name[MAX];
    XmString ds_label;
    char *ds_style;
    double ds_value;
    int ds_amount;
    char ds_redzone;
    char ds_alarm;
};
```

ds_name is the name of the shared memory component to be displayed;
ds_label is that same name in a form printable by Xt Intrinsics; ds_style
tells if the monitoring device is a dial or simply an alarm window;
ds_value is the true value of the component to be displayed; ds_amount
is the adjusted value that is actually displayed on the dial; and
ds_redzone and ds_alarm are set to 'y' if a component's value reaches
certain points. When "Monitor" was originally written, the intention was
to change the color of the dial if a value reached a given point on the
dial. However, through experiment it was discovered that the SGI Indigo
R3000 does not properly support the color changes necessary to make
this feature effective. Therefore, the ds_redzone field is set for every
component, but it is never utilized. I could have removed this field from
the structure, but since this application can be run on machines other
than the SGI's, I left it in to cut down on the work needed to allow for
color changes on other systems.

Once all relevant information has been stored in the array of structures, **dial** starts displaying the dials. The program does this work by first initializing a top level widget. This step sets up a connection to Xt Intrinsics which allows any widgets that are used to be displayed to the screen. Second, a base window is created which will organize the button and dial widgets in rows and columns. Next we create three buttons used to control the "Monitor" program: Pause, Continue, and Stop. These buttons, when activated, make a call to their respective functions: Pause causes the program to hold its current dial positions, Continue starts the dials moving again, and Stop kills the program.

Now that the preliminary work has been done, **dial** moves into its main loop to create a dial for each component of shared memory whose **ds_style** field is set to "dial". Attached to the top of each dial is a push button which displays the name of the shared memory component monitored by that dial. If the button is activated, the **alarm** process is called and told to display help information about that dial. In any event, the dials and buttons are displayed in rows and columns, and their indicators point to the position set by the **ds_amount** field in the array of structures. As the program runs, it updates the values of the dials at regular intervals, re-executing the **tbstat** utility and getting new information to be stored in the array of structures. Figure-6 shows an
Database attempted to exceed total

Database space may need to allocate more space.

The Monitor Program

FIGURE 6
example of what the "Monitor" program might look like at run time.

In addition to being called by some kind of user activity, alarm is also called automatically when the ds_alarm field has been set to 'y'. As dial gathers new information, it resets the fields of each structure in the array. When the ds_alarm field is set to 'y', dial creates a child, or subprocess, which calls the alarm program and requests that an alarm be created and displayed to the screen. This creation of the child process is shown in Figure-7. The important point to understand about creating a child process is that it creates an exact replica of the program currently running, maintaining all values as they have been set by the main program. Once this replica has been made, the program that created the subprocess becomes known as the parent process. Dial creates its child process by using the "fork" command, shown below:

```c
if ( !fork() )
{
    <do some kind of work>
}
```

By using this code, the program creates a subprocess and only runs the commands between the braces in the child process. That is, the parent process does not make a call to alarm, it simply continues on with the code that follows the last brace and leaves the alarm call to the child.

When the child calls alarm, it uses the "system" command just like the call to tbstat does. The child sends an argument with alarm to notify alarm which dial caused the alarm to be created. Alarm then
FIGURE 7

Creation of Child Process to call `alarm`

```plaintext
dial (parent)

if (alarm = 'y')
{
  if (!fork())
    dial (child)

  system("alarm <name>");
  write to pipe: `alarm` is done;
  exit process;
}
```
pops up the proper alarm message and waits until the window is killed. Once the window has been killed, control returns to the child process, which then notifies the parent process through a Unix pipe that alarm is finished. The parent process needs to be notified about alarm's conclusion because "Monitor" is designed not to display an alarm for a dial if an alarm is currently being displayed. This feature reduces the risk of running out of memory because alarms cannot continually pile up on themselves. Once the child process notifies the parent through the pipe, the child ends its own process, leaving the parent process to run alone once again.

The Unix pipe has been mentioned twice: once in the INFORMIX OnLine client/server model, and here with alarm. Pipes and child processes make this program run much more efficiently than if the program tried to handle all of the widget creation on its own because they eliminate a large amount of overhead. By having the alarms generated by a separate program, the dials can continue to be updated at regular intervals without having to wait for the alarms to be popped up. Pipes are important because they allow child processes to communicate with their parent processes. Although a child process is running the same program as its parent, the child is a completely independent process. Therefore there must be a link between the two processes if information is to be passed. The pipe is that link. A process
can send information into the pipe and it can come out "the other end" into the parent process. This information is then usable in the process that received it, and in the case of the dial parent process, that information means an alarm window may be displayed for the dial whose alarm window was just closed.

**Improvements**

There are a few improvements "Monitor" could undergo to become an even better and more user-friendly application. Of course as the program is used more often, other improvements will be certain to surface, but the following improvements are the ones which I found to be the most important. First of all, it would be convenient to have the color system work to provide a two levels of warning to the system administrator. Currently, if a dial's indicator reaches a certain position, a warning window is displayed telling the user that if something is not changed soon, there is a chance that the system could crash. Having a dial change color before its indicator reaches a position that triggers an alarm could tell the user to watch that dial's shared memory component more closely. The changing of the color would not mean that something must be done immediately to fix a current problem, it would only be used to raise awareness to that particular area.

Another improvement that might be useful for the system administrator would be to display the actual tbsstat information relevant
to a particular dial when the button above that dial is activated. This feature would still allow the user to get a general help message which explains what the dial is monitoring, but it would also tell the user exactly what the status of that dial's shared memory component is. Although the dial is extremely helpful for displaying the general status of its particular component, providing this additional information would give the system administrator the precise data necessary to make informed decisions about the efficiency of the system.

One final improvement could be to create a popup window which allows the user to change the time interval used to update the dials. It may be useful to change this interval as the program is running because there are times when the administrator will want to monitor the system more closely than other times, especially if a dial's color changes or if an alarm window is displayed. When the system seems to be running smoothly, it may not be necessary to update the dials nearly as often. The interval could then be increased to decrease the amount of work being done on the system. Currently the interval can only be controlled at the command line by following the dial command with an integer argument for time in seconds. If no argument is given, the program runs at the default of five seconds.

Conclusion

Even though the program could use improvements, "Monitor" is
certainly a useful program for system administrators. Not only does the program display information that monitors the system, but it displays it with simplicity. Dials are easy to read, popup windows can be eliminated when they are not needed, and the three command buttons make the program easy to control. "Monitor" is also a very general program, which means that it can be modified with relatively little work to display any kind of system information, not just information relating to the INFORMIX OnLine Database. Furthermore, since other programs can be written to simulate the type of information that "Monitor" displays, the program is very portable and can be shown on any system that supports C and the X Window System. In short, a graphical application like "Monitor" could be the type of tool that makes evaluating system efficiency easier because components are monitored independently. The system administrator no longer needs to search through cluttered files of information to understand how the system is using its shared memory components. With "Monitor", the information is made available at the click of a button.
BIBLIOGRAPHY


Appendices

A. Shared Memory Information Monitored by Dials

B. The Monitor Program
   dial.c
   get_display_info.c
   mon_fcts.c
   mon_interface.h
   mon_defs.h
   Makefile
   alarm.c
   Make_alarm
   tbstat.c
   Dial.c
   Dial.h
   DialP.h
Appendix A

SHARED MEMORY INFORMATION MONITORED BY DIALS

CHECKPOINTS: flush out the buffer to disk; system back up. This dial displays (average of all recorded checkpoints/CKPTINTVL) * 100. CKPTINTVL is the minimum checkpoint interval recommended. The dial hits its red zone if this display amount is less than half of CKPTINTVL; it hits an alarm if its amount reaches the red zone for five or move checkpoint times in a row.

USERS: number of processes running on the system. This dial displays (active users/total users allowed) * 100. The dial hits its red zone and triggers and alarm if the number of active users comes within five of the total users allowed.

TRANSACTS: shows number of all transactions associated with users on the system. This dial displays (active transactions/total transactions allowed) * 100. The dial hits its red zone if this amount is greater than 85; it hits an alarm if the amount exceeds 95.

LOCKS: used to manage access to shared memory buffers by limiting the number of users to buffers or by limiting the type of access users have to buffers. This dial displays (active locks/total locks available) * 100. The dial hits its red zone if this amount is greater than 85; it hits an alarm if the amount exceeds 95.

BUFFERS: system memory; Big Buffer = 8 pages. Reg. Buffer = 1 page. This dial displays (modified buffers/total buffers available) * 100. The dial hits its red zone if this amount is greater than 85; it hits an alarm if the amount exceeds 95.

TBLOSPACES: amount of disk spaces allocated to a specific database table. This dial displays (active tblspaces/total tblspaces available) * 100. The dial hits its red zone if this amount is greater than 85; it hits an alarm if it exceeds 95.

CHUNKS: largest unit of physical disk dedicated to OnLine data storage; measured in pages. All chunks available have their own dial: active chunk dials display ((chunk size - pages free)/chunk size) * 100. Each active chunk dial hits its red zone if this amount is less than 90; an alarm is set if all active chunks have hit their red zones. Inactive chunks are
labeled as such and display an amount of zero.

**PHYSICAL LOGS:**

collections of contiguous pages on disk used to record what pages looked like before they were modified; used for recovery purposes. This dial displays \(((\text{pages/io})/\text{bufsize}) * 100\). The dial hits its red zone and triggers an alarm if this amount falls below 75.

**LOGICAL LOGS:**

collections of contiguous pages on disk used to store all modifications since last system backup. This dial displays \(((\text{pages/io})/\text{bufsize}) * 100\). The dial hits its red zone and triggers an alarm if this amount falls below 75.

**READS CACHED:**

comparison of the number of times data is read from memory (buffers) versus number of reads from disk. This dial displays \(((\text{bufreads} - \text{diskreads})/\text{bufreads}) * 100\). The dial hits its red zone and triggers an alarm if this amount falls below 95.

**WRITES CACHED:**

comparison of the number of times data is written to memory (buffers) versus number of writes to disk. This dial displays \(((\text{bufwrits} - \text{diskwrits})/\text{bufwrits}) * 100\). The dial hits its red zone and triggers an alarm if this amount falls below 85.

The remaining five components are not represented by dials. They all trigger an alarm if their values exceed zero.

**OVTBLs:**

number of times OnLine attempted to exceed maximum number of available tbspaces.

**OVLOCK:**

number of times OnLine attempted to exceed maximum number of locks.

**OVUSER:**

number of times a user attempted to exceed maximum number of users.

**OVBUFF:**

number of times OnLine attempted to exceed maximum number of shared memory buffers.

**DEADLKS:**

increments each time a potential deadlock is detected and prevented.
Appendix B

The Monitor Program--Code

dial.c
get_display_info.c
mon_fcts.c
mon_interface.h
mon_def.h
Makefile
alarm.c
Make_alarm
tbstat.c
Dial.c
Dial.h
DialP.h
null

**FUNCTION**

1. Initialize the count, current, and next pointers.
2. Iterate through the linked list to count the number of elements.
3. Print the number of elements.

**METHODS**

1. Initialize the count, current, and next pointers.
2. Iterate through the linked list to count the number of elements.
3. Return the count.

**EXAMPLE**

```c
struct Node {
    int data;
    struct Node* next;
};

int countElements(struct Node* head) {
    int count = 0;
    struct Node* current = head;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}
```

**OUTPUT**

```
Number of elements: 5
```
"BEFORE transition to the next page:"
"..."
mess = msg_help_wcache;
}
) /* end if argv == "help" */
toplevel = XtInitialize(argv[0], "Alarm", NULL, 0, &argc, argv);
help_callback(NULL, mess, NULL);
XtRealizeWidget(dialog);
XtMainLoop();

} /***************************************************************************/
void ok_callback( w, client_data, call_data)
Widget w;
caddr_t client_data;
XmAnyCallbackStruct *call_data;
{
exit(1);
} /***************************************************************************/
void help_callback( w, mess, call_data)
Widget w;
char *mess[];
XmAnyCallbackStruct *call_data;
{
Widget label;
/* Create warning dialog to display tstat warnings */
n = 0;
XtSetArg(alarm_args[n], XmNautoUnmanage, FALSE); n++;
if (need_help == 'n')
dialog = XmCreateMessageDialog(toplevel, "WARNING", alarm_args, n);
else
dialog = XmCreateMessageDialog(toplevel, "HELP", alarm_args, n);
XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_CANCEL_BUTTON));
/* ok_callback closes the warning window */
XtAddCallback(dialog, XmNokCallback, ok_callback, NULL);
/* Get next message from mess array--read until a null is read */
for (i=0; mess[i][0] != '\0'; i++);
/* Make text left justified */
label = XmMessageBoxGetChild(dialog, XmDIALOG_MESSAGE_LABEL);
n = 0;
XtSetArg(alarm_args[n], XmNalignment, XmALIGNMENT_BEGINNING); n++;
XtSetValues(label, alarm_args, n);
/* change array of character strings into an XmString */
alarm_msg = xs_str_array_to_xmstr(mess, i);
/* display warning message to warning window */
n = 0;
XtSetArg(alarm_args[n], XmNmessageString, alarm_msg); n++;
XtSetValues(dialog, alarm_args, n);
/* Read for next message. If message is empty, messages complete */
if (mess[++i][0] == '\0')
XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON));
else
XtAddCallback(dialog, XmNhelpCallback, help_callback, &mess[i]);
XtManageChild(dialog);
clear
cc -o armco.o $(LIE)
alarms.o
cc $(LIB) -o alarm armco.o $(LIE)
alarms alarm
.o
LIES = -Wmort/ll/bx/a-xm -xrc -xrl -im
LIB = $(DCM) -I$(XSTBIDB) -I$(XSTBIDB)
cc $(LIE) -o arm $(LIE)
cc = cc

/**************************
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
 /*
/*****************************/

May 6 1993 1993 Page 1
/*
 * Draw the triangle on the output position.
 */

int draw_triangle(int x, int y, int length)
{
    int angle = (length * 180) / 26832;
    int x1 = x + length * sin(angle * 3.14159 / 180);
    int y1 = y - length * cos(angle * 3.14159 / 180);
    int x2 = x + length * sin(angle - 60 * 3.14159 / 180);
    int y2 = y - length * cos(angle - 60 * 3.14159 / 180);
    int x3 = x + length * sin(angle + 60 * 3.14159 / 180);
    int y3 = y - length * cos(angle + 60 * 3.14159 / 180);
    return 1;
}

int main()
{
    int x = 1000;
    int y = 500;
    int length = 100;
    draw_triangle(x, y, length);
    return 0;
}