ELF Communications Guide

This manual was produced using Applixware.

Printed: April 2000

Contents

Chapter 4 Creating an RPC Function

Figures

Tables

Preface

About This Manual

The *ELF Communication Guide* describes how ELF programs can interact with C language programs using the following mechanisms:

- Remote Procedure Calls (RPCs)
- Shared Libraries
- Sockets

Conventions Used in This Manual

The following typeface conventions are used throughout this manual:

Helvetica	Helvetica text indicates that this option or object appears in the document window. For example, "Type the name of the document in the File name entry area." File names and directories are also indicated by Helvetica text. For example, "This file is located in your axhome directory."

	Applixware keys are printed in a Helvetica uppercase typeface. For example, "Press the TAB key."
Helvetica Bold	Bold Helvetica text indicates an option to choose or text to type. It usually appears in numbered steps as shown in the following example: 1. Type **2.5** in the Line spacing entry area. 2. Click **Apply**.
Italics	Words are italicized for emphasis or to draw your attention to a new term. For example, "*Do not* press the RETURN key," or "This action is called *word wrapping*." Italic type is also used to indicate variable information, as in "Put Applixware in the /user/*your_name* directory."

Menu Name → Option Name	Whenever you see a reference to a menu option, the option is identified using the following notation: Menu Name → Option Name For example, "Choose **File** → **Save**."
OK and Apply	When numbered instructions are included in the text, we omit the final "Click **OK** or **Apply**" statement for brevity.

Applixware Window Environment and Interface

Consult your window manager and hardware documentation if you need information about how to operate in your window environment.

1 ELF Communication

Most ELF programs run within the Applixware environment. This book describes another kind of program that interacts with C. Typically, the C program is responsible for managing a scarce system resource, such as a special piece of hardware or a database.

This book assumes that you are an ELF and C language programmer. For some of the information presented in Chapter 5 and in the Appendices, you should also be familiar with socket programming.

Overview

The C language programs that you write will perform activities that cannot be performed within ELF or, if they can be performed, would be performed poorly. Here are some examples:

- You need the ability within Applix Spreadsheets to perform statistical analysis of your data beyond that offered by Applix. In this circumstance, you have two choices: (1) You can write a C language function and have ELF call this function to perform the analysis; or (2) You can purchase a statistical package from another vendor and write a program that allows ELF to send data to a C language program that, in turn, calls the statistical package. After the data is analyzed, your C routine returns the data to your ELF program.

- You need to use specialized hardware. This specialized hardware can be anything you can connect to your computer network, such as FAX system or a number-crunching computer.

  You could write a C program to manage the hardware. Typically, this would be a UNIX daemon that manages transactions between the users and the hardware.

- You need to interact with a database. Using databases is what people typically think of when client-server computing is discussed. This is very similar to managing hardware resources. In this case, a daemon process provided by the vendor controls how you interact with the database.

  A database could also be a store of information you manage. For example, you could keep your company's phone book on-line. In this case, you would allow any program to read the book. However, you would not allow more than one program to change it.

The interaction pattern is called *client-server*. In a "client-server" transaction, one program, called the *client*, requests that another program, called the *server*, do something for it. When the request is fulfilled, the server returns data to the client or returns an acknowledgement that the request was acted upon.

Communicating between ELF and C

Moving data between ELF and a C program presents special problems to the programmer. This section describes some of these problems.

Variables

The ELF language is designed to obscure the format in which information is stored. Any kind of data can be assigned to any variable. An array of data can either be homogeneous, where all the data is of the same kind, or heterogeneous, where the data is of different kinds. When data is assigned to a variable, ELF converts the data into an appropriate format.

When you pass data from ELF to C, you must be aware of the type of data that is being passed between the two programs. A variable declared as 'VAR' in ELF can be a string, an integer, or an array. When that variable is passed to a C program, you must be aware of the type of data being passed so that your C program handles it properly.

Pointers

When you pass data between two C programs, it can be passed by value or by reference. When a number is directly passed from one

program to another, it is passed by value. Data can also be passed by reference using a pointer, which is the memory address of the first element of the data. Passing a pointer is more efficient than passing the complete contents of an array or structure.

All data passing between ELF programs and C programs is passed by value.

Packing and Unpacking Data

Most programs that move data between ELF and C include routines to pack and unpack data. Applixware includes a set of functions that transform the data sent by ELF into a form usable by C language programs. Another set of functions transforms C data into a form usable by ELF.

Memory Allocation

ELF is designed to allocate and free the memory it needs to store your data. In contrast, a C language program must explicitly allocate and free the space it uses to store information. Because ELF data must be decoded before it can be used in a C language program, your program must be responsible for managing its memory.

Coding Conventions

The ELF / C communication layer has a set of conventions that must be rigorously followed.

RPC and Shared Libraries

You can implement C routines two ways:

- RPCs
- Shared Libraries

RPC is a general term meaning Remote Program Communication. The computer industry uses a number of different RPC protocols. The ELF RPC protocol is unique to Applixware.

Applix recommends that all initial development and debugging be done using RPCs. Implementing routines in this way is safer because the C programs run as separate processes. This means that no bug in your C program, no matter how serious, can affect Applixware.

Once the C programs are debugged and working, you can make some small modifications to your macro code, and install the C programs as shared libraries. Shared libraries are faster than RPCs, and the ELF interface to a shared library is simpler. The only risk associated with shared libraries is that they run in the same application space as Applixware itself. It is therefore possible for a poorly-written shared library to crash Applixware. Implementing each add-in function as an RPC first reduces this risk considerably.

The process of developing RPCs and shared libraries is described in detail in Chapters 2, 3, and 4 of this book.

2 Add-in Functions

You can implement add-in functions for Applixware using add-in libraries or RPC programs. This chapter describes the following:

- Creating add-in functions
- Building an RPC program
- Debugging an RPC program
- Important programming notes
- Building add-in libraries on various platforms
- Calling add-in libraries from ELF

Creating Add-in Functions

Add-in functions are implemented through the use of libraries which are linked with the Applixware process at run time. In a UNIX environment, these are called *shared libraries*, and in an WIndows NT environment these are called *Dynamic Link Libraries* (DLLs). The examples in this chapter apply to either Unix or Windows NT environments.

The basic structure of an add-in function is shown in the following code example:

```
/*
*  This module contains a sample user-defined function. This C program can be
*  run either as an RPC or a add-in library.
*/

#include "elfapi.h"
#define TRUE 1

elfData FUNC();

/*
* The first thing in the C program should be the Function Table.  The Function Table is an
* array of AxCallInfo_t structures.  The elements of this structure are described later in this
* chapter.   Note that the last entry in the array is all NULLs. The AxCallInfo_t structure is
* defined in elfapi.h.
*/

 AxCallInfo_t     funcTable[]       =         {
     { "MyCategory", FUNC, "FUNC", "FUNC()", TRUE },
     { NULL, NULL, NULL, NULL, NULL }
};
```

```
/*
* The function AxGetCallInfo returns the function table. Applixware calls AxGetCallInfo
* when you call either RPC_CALL@ or INSTALL_C_LIBRARY@ from your ELF macro.
* This way, Applixware can retrieve the names of your add-in functions from your C code.
*/

DLL_EXPORT AxCallInfo_t *AxGetCallInfo()
{
    return(funcTable);
}

/*============================================================*/
/*  A very simple user-defined function. This function returns a string.  */
/*============================================================*/

elfData FUNC(argsP)
{
    elfData retP;
    retP = AxMakeStrData(-1, "Hello There!" );
    return(retP);
}
```

The sections that follow describe the elements of an add-in function.

Function Table

The first element of the add-in function is the function table. The function table contains an array of AxGetCallInfo_t structures.

The AxGetCallInfo_t structure is defined in elfapi.h as follows:

```
typedef struct AxCallInfo_t {
    char *category;             /* func type ... "financial", "math" ... */
    elfData (*func)();          /* The C routine to call.
    char *alias;                /* name to use in Applixware (usually identical
                                        to the function name)   */
    char *argStr;               /* shows the function name and its arguments */
    int callMode;               /* An integer that governs the treatment of ERROR
                                   and NA values, and whether the function is
                                   displayed in the Spreadsheets Functions dialog
                                   box. */
} AxCallInfo_t;
```

Category

The first element in the AxCallInfo_t structure is the function category. Applixware Spreadsheets defines a number of function categories. These categories are displayed in the Categories list box when you select the Utilities→Functions... menu. The example function would create a new category called MyCategory, and would be the only function listed in that category.

Name of C Routine

The second element in the AxCallInfo_t structure is the name of a C routine that is defined in your add-in function. This name must be all uppercase, otherwise Applixware does not recognize it.

Name To Use in Applixware

The third element in the AxCallInfo_t structure is the name that is used within Applixware to call the function. Usually, this is identical to the name of the function. This must be in uppercase.

Argument List

The fourth element in the AxCallInfo_t structure is the name that you want to appear in the Functions: box on the Utilities→Functions... menu. As the example function shows, this string should be identical to the name of the C function, with the argument list for the function in parentheses, and separated by commas.

CallMode

The last field in the AxCallInfo_t structure is a callMode. This field can be set to one of four values, as follows:

- If callmode is set to 0x00, the function does not appear in the Spreadsheets functions dialog box. NA and ERROR values are passed as NULLS.
- If callmode is set to 0x01, the function appears in the Spreadsheets functions dialog box. NA and ERROR values are passed as NULLS.
- If callMode is set to 0x10, NA and ERROR values are passed to the C function as binary objects. Usually, NA and ERROR values are passed as NULLS.

- If callmode is set to 0x11, the function appears in the Spreadsheets Functions dialog box. NA and ERROR values are passed as binary objects.

AxGetCallInfo Function

AxGetCallInfo must be defined in your add-in function. Applixware uses this function to retrieve the function table from your add-in function. AxGetCallInfo is called by Applixware when you call either RPC_CONNECT@ or INSTALL_C_LIBRARY@ from ELF.

Functions

You can have as many functions as you like in your add-in library. The following example shows an add-in library with two C functions in it.

```
/*
* This C module contains two sample user-defined functions.
*/

#include "elfapi.h"
#define TRUE 1
elfData TESTFUNC(), ADDNUM();


AxCallInfo_t funcTable[] =          {
     { "MyCategory", TESTFUNC, "TESTFUNC", "testfunc()", TRUE },
     { "MyCategory", ADDNUM, "ADDNUM", "addnum()", TRUE },
     { NULL, NULL, NULL, NULL, NULL }
};


AxCallInfo_t *AxGetCallInfo()
{
     return(funcTable);
}
/*================================================================*/
/*                    Two Examples of User Defined functions                    */
/*================================================================*/

/* function TESTFUNC - returns a string                                        */
elfData TESTFUNC(argsP)
     elfData argsP;                /* not used in this function */
{
     elfData retP;
     retP = AxMakeStrData(-1, "Successful evaluation of testfunc" );
     return(retP);
}

/*
* function ADDNUM - adds two numbers
* Inputs - two numbers
* returns - sum of the two passed numbers
*/
elfData ADDNUM(argsP)
```

```
    elfData argsP;                  /* the array of arguments */
{
    double val;
    elfData retP, firstP, secondP;

/* make sure exactly two arguments are passed to this function                    */
    if(AxArraySize(argsP) != 2)
        AxError(1, "Wrong number of  arguments", "ADDNUM");

/* Use AxError() to throw errors back to Applixware                */

    firstP = AxArrayElement(argsP, 0);

    secondP = AxArrayElement(argsP, 1);

/* compute the sum of the two numbers */
    val = AxFloatFromDataPtr(firstP) + AxFloatFromDataPtr(secondP);

/* Return result in elf data format to Applixware */
    retP = AxMakeFloatData(val);
    return(retP);
}
```

Applixware and the add-in functions exchange data in ELF format. The input to the add-in function is an ELF datum created by the caller. The return value from the add-in function is an ELF datum you create in your add-in.

Building an RPC Function

A working add-in function can be installed either as an RPC or as a add-in library, with no changes to the C code.

Applix strongly recommends that you develop add-in libraries using a two-step procedure:

1. Create and debug your functions as an ELF/RPC.
2. Recompile the debugged source code to create a add-in library.

There are two reasons for this:

- ELF/RPC functions are much easier to debug than add-in libraries. In fact, some operating systems may not allow you to debug add-in libraries at all.
- When you load an add-in functions into Applixware, they become part of the Applixware process. Errors in the add-in functions can corrupt the entire Applixware process. You should convert the ELF/RPC code to an add-in function only when you are satisfied that your ELF/RPC code is working correctly.

Compiling an RPC

The following example describes how to build an RPC program called test from the source file test.c:

```
cc -o test test.c $ELFLIBPATH/elfapi.a
```

In this example, $ELFLIBPATH is the pathname to your elfapi.a library. The location of this library depends on your installation.

NOTE: Starting in Applixware version 4.2, ELF/RPCs are compiled and linked into executable programs. This differs from previous versions, where the ELF/RPC modules were compiled into object code.

Calling an RPC from ELF

The following macro calls a compiled ELF/RPC called test.

```
/*
* This macro calls an ELF/RPC called test that resides in the
*  directory /user/applix
*/
macro rpctest
' start the program called test
RPC_CONNECT@("/user/applix/test", 6162)

' Display the text output of test() in an info box
info_message@(test())
endmacro
```

Debugging an RPC Program

This section describes what to do before you can debug your RPC program.

1. Select an unused socket number such as 6162. Check your /etc/services file to find one that is unused.
2. Load your RPC program in the debugger.
3. Run the RPC program in your debugger passing it the socket number you selected in step 1.
4. Run RPC_CONNECT@() passing it the RPC program as the first argument and the socket number as the second argument. For example:

   ```
   RPC_CONNECT@("/user/applix/test", 6162)
   ```

When you pass a socket number to the RPC_CONNECT@() macro, Applixware assumes that the RPC program is running and listening on the specified socket and connects to the RPC program on that socket.

Now you are ready to debug your add-in functions.

5. Set your breakpoints in the appropriate functions and call those functions from Applixware.

Important Programming Notes

This section lists important notes to consider when you are creating your add-in functions and add-in libraries.

- Be sure to terminate the function's entry table with a NULL entry.
- The args array that is being passed to your add-in function is owned by Applixware and therefore you cannot free it.
- Do not directly return one of the arguments that was passed to you by the caller. Instead, use retP=AxCopyData(*xx*) to return an argument.
- If you create functions with static data, the data is add-in by all of the macros / spreadsheets that call the function. Expressed another way, if your functions maintains "state", the state is add-in by all callers unless you explicitly maintain an array of states - one for each caller.

- The memory associated with the ELF datum returned from the add-in function is owned by Applixware and will be freed by Applixware later on.
- All other memory allocated by your add-in function for temporary use should be freed by your add-in function before it returns.
- Beware of the call AxFreeData(). AxFreeData is a recursive call and when called with an array as an argument it would recursively free the array and the elements in it. So you should never free a sub element of an array and then free the array or vice-versa. Instead you should just free the array.

Building Add-in Libraries

Once your ELF/RPC function is working, you can quickly install the C program as an Applixware add-in library. To do this, follow these steps:

1. Re-compile the C module that you want to install as a add-in library. The procedure for doing this compilation varies with the platform that you are compiling on, and the compiler you are using. See the section "Compiling add-in Libraries" for example information on compiling your C code.
2. Write an ELF macro to call the add-in library.

Compiling Add-in Libraries

This section describes how to build add-in libraries on various hardware platforms and operating systems. The following platforms are discussed:

- SUN Solaris and Intel Solaris
- Windows NT

Please refer to the previous section "Important Programming Notes" before you create and install your add-in libraries.

NOTE: Applix, Inc. is not a supplier of compilers, linkers, or debuggers. The information in this section is provided as a courtesy. If you encounter any problems compiling, linking, or debugging add-in libraries to Applixware, contact the vendor of your software tools.

The following lists the compiler and linker flags necessary to build add-in libraries using compiler and linkers provided with the workstations that Applix supports. In the following lists, test.so (test.dll for NT) is the name of the library you want to create and test.c is the source file.

SUN Solaris

```
cc -K PIC -G  -g -o test.so test.c
```

Windows NT

To compile, use the standard compile line from your makefile, or from Visual C++. The only requirement is that you pass a "-DAXMS=1" switch as part of the compile flags, so that the DLL_EXPORT macro (found in elfapi.h) expands to generate the correct DLL linkage information.

NOTE: The DLL_EXPORT macro is defined in elfapi.a.

To link, follow the standard makefile to build a DLL, and include axmain.lib so that the DLL can resolve linkage information from the main Applixware process. The library axmain.lib is found in your Applixware axdata distribution directory.

Sources which are to be run in both Unix and Windows NT should declare the main add-in entry point AxGetCallInfo() as DLL_EXPORT. This macro does nothing under UNIX. Under Windows, assuming you use the Microsoft Visual C++ 2.x compiler, it generates the correct DLL linkage information.

Calling Add-in Libraries From ELF

To call a add-in library from ELF, you must install the library using the macro INSTALL_C_LIBRARY@. You can then call any of the functions defined in that library. The following macro installs the C library shown in the section "Functions" earlier in this chapter. The library contains two C functions: ADDNUM() and TESTFUNC(). This macro calls both of these functions.

```
include "errors_.am"

macro shlib

    var pathname, vals
    pathname = "/newuser/applix/shlib/test.so"
    INSTALL_C_LIBRARY@(pathname)          ' This macro installs the add-in library

/*
* ADDNUM is a user-defined function that adds two numbers.
*/
    vals = ADDNUM(5,7)
    info_message@("The answer is "++vals)

/*
* TESTFUNC is a user-defined function that returns a string.
*/
    vals = TESTFUNC()
    info_message@(vals)
    unbind_C_library@(pathname)

/*
* UNBIND_C_LIBRARY removes the add-in library from Applixware. This function is very
* handy for debugging sessions. You want to unbind your old library before you install a
* new one with the same name.
*/
endmacro
```

NOTE: The function INSTALL_C_LIBRARY@ has changed. Earlier versions of this function required two arguments: a path name to the add-in library, and an array of function names contained in the add-in library. The function table and the AxGetCallInfo function make the array of function names no longer necessary. The old calling convention is still supported for the sake of backward compatibility.

3 Add-in Function Examples

This chapter contains five add-in functions that can be used within ELF programs and from within a spreadsheet. The functions are as follows:

- ADDNUM1() adds two numbers and returns the sum.
- ADDNUM2() adds an array of numbers. The array can have any number of elements.
- ADDNUM3() adds all the elements of a multi--dimensioned array.
- ADDNUM4() uses the AxCube family of C functions to process an array passed from a spreadsheet.
- ADDARRAYS() accepts two arrays passed from ELF, and returns an array containing the sum of the corresponding elements.

Addnum1: Adding Two Numbers

To create a C language function that receives two numbers from ELF and returns the sum of the two numbers requires:

- A C language function that receives the data and sends the sum back to your ELF macro.
- An ELF macro that calls the C language function.

The C Language Function

The following code sample shows the complete C language program that receives two numbers from ELF and returns the sum.

```
/*
*    Code Sample 3-1. This function adds two numbers.
*/

#include "elfapi.h"
#define TRUE 1

elfData ADDNUM1();

/*
*    Define the function table
*/
AxCallInfo_t      funcTable[]       =          {
     { "MyCategory", ADDNUM1, "ADDNUM1", "addnum1()", TRUE },
     { NULL, NULL, NULL, NULL, NULL }
};

/*
*    Function AxGetCallInfo returns the function table.
```

```
*     This function is called by Applixware when you run the macro
*     RPC_CONNECT@ or INSTALL_C_LIBRARY@. This function must exist
*     in the RPC program or Shared Library.
*/
DLL_EXPORT AxCallInfo_t *AxGetCallInfo()
{
     return(funcTable);
}


/*=======================================*/
/* ADDNUM1 - This function adds two numbers            */
/*=======================================*/


elfData ADDNUM1(argsP)
     elfData argsP;                    /* argsp is an array of arguments passed from ELF */
{
     double val;
     elfData retP, firstP, secondP;

/* make sure exactly two arguments are passed to this function */
     if (AxArraySize(argsP) != 2)
          AxError(1, "Not enough arguments", "ADDNUM1");

/* make sure first argument is a number */
     firstP = AxArrayElement(argsP, 0);
     if (!AxIsNumber(firstP))
          AxError(1, "Invalid first argument", "ADDNUM1");

/* make sure second argument is a number */
     secondP = AxArrayElement(argsP, 1);
     if (!AxIsNumber(secondP))
          AxError(1, "Invalid second argument", "ADDNUM1");

/* compute the sum of the two numbers */
     val = AxFloatFromDataPtr(firstP);
     val += AxFloatFromDataPtr(secondP);
```

```
/* Pass the result back to Applixware */
    retP = AxMakeFloatData(val);
    return(retP);
}
```

The structure of Code Sample 3-1 is typical for an add-in function. The common elements are as follows:

- The elfapi.h file. This file contains the function prototypes Applixware provides that convert data between ELF and C. These are called *convenience functions*. Convenience functions are described in the next section.
- All C functions must return something to the calling ELF macro, even if the returned value is NULL.
- The function table. This data structure lists all of the functions in the C module, and provides information about the function to Applixware. This data structure is described in detail in Chapter 2.
- The function AxGetCallInfo. This function is used by Applixware to retrieve information about the function in the RPC program or shared library.

Convenience Functions

ADDNUM1() uses several Applixware *convenience* functions. A convenience function is a C-language function that manipulates data passing between C and ELF. The convenience functions used in ADDNUM1() are:

- AxArraySize() examines an ELF data array and returns the number of elements in the array.
- AxFloatFromDataPtr() creates a floating point value from a C pointer.

- AxMakeFloatData() creates an ELF floating point value from a C floating point value.
- AxError() stops execution of the function, and displays an error message on the screen.
- AxArrayElement() extracts an element from an ELF array.
- AxIsNumber() tests a value to see if it is a number, and returns a Boolean.

Compiling Your C Program

Assuming that your C code is in a file named addnum1.c in your current directory, you can compile the file as follows:

```
cc -o addnum1 addnum1.c  /applix/axexec/axdata/elf/elfapi.a
```

If the program compiles and links without error, the compiler and linker should generate an executable called addnum1. This program can be called from an ELF macro using the ELF/RPC mechanism. The macro is shown in the following section.

The ELF Macro

The macro that calls the function ADDNUM1() is as follows:

```
/*
*    Code Sample 3-2. This macro calls the C function shown in
*    Code Sample 3-1.
*/
macro rpctest

RPC_CONNECT@("/newuser/robert/shlib/addnum1", 6163)
info_message@("The total is "++ADDNUM1(6,5))

endmacro
```

The rpctest macro:

- Uses RPC_CALL@ to open a channel to the executable addnum1. The second parameter is an unused socket number. This can be any socket number above 100 that is not reserved in the file /etc/services.
- Passes two arguments, 5 and 6, to the function ADDNUM1(), which is built into the addnum1 executable.
- Displays the value returned by ADDNUM1() in an info box.

RPC_CALL@ is used when you call an RPC function once. Using RPC_CALL@ is not an efficient way to use system resources if you are going to repeatedly make calls to a function.

Building a Shared Library

Once you are certain that your RPC is working properly, you can re-compile the C source code to build an Applixware shared library. No modifications to the C source code are necessary.

To compile the source code in Code Sample 3-1 into a shared library under SUN OS, use these commands:

```
cc -g -c -PIC addnum1.c
ld -o addnum1.so addnum1.o -assert pure-text -lm
```

NOTE: You should not include elfapi.a when building shared library since elfapi.a already exists in Applixware. Since your C module is installed as part of Applixware, linking with elfapi.a is unnecessary.

Calling a Shared Library from ELF

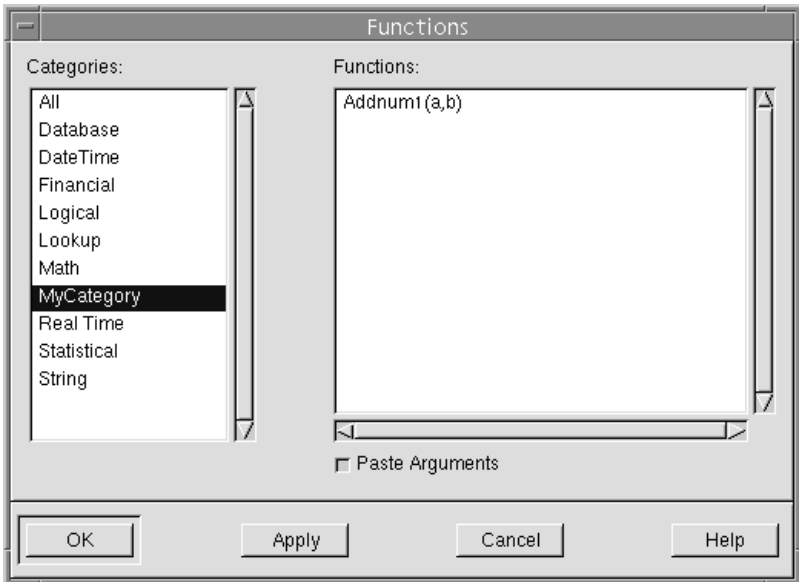
The following macro installs and runs the shared library generated in the previous section.

```
/*
*    Code Sample 3-3. This macro installs the C function shown in
*    Code Sample 3-1 as a shared library.
*/

include "errors_.am"

macro shlib

    var pathname, vals
    pathname = "/newuser/applix/shlib/addnum1.so"

    INSTALL_C_LIBRARY@(pathname)

    vals = ADDNUM1(5,7)
    info_message@("The answer is "++vals)

    unbind_C_library@(pathname)

endmacro
```

Calling Addnum1 from Spreadsheets

Once you have run the Addnum1() function either as an RPC or a shared library, you can call the function from Spreadsheets. Follow these steps:

1. Open Applixware Spreadsheets.
2. Choose Insert → Functions. The Functions dialog box appears.
3. Select **MyCategory** from the Categories list. The Addnum1() function appears in the Functions list.



From this screen, you can double-click the function and run it from the Spreadsheets command line.

Addnum2: Adding an Array of Numbers

The Addnum1() function adds only two numbers together. The function in this section accepts an array of any size from ELF, adds the numbers, and returns the total to ELF.

The C Program

The following C program adds the values of an ELF data array, and returns the total.

```
/*================================================================*/
/*    Code Sample 3-4.                                            */
/*================================================================*/
#include "elfapi.h"
#define TRUE 1

elfData ADDNUM2();

/*
*    Define the function table
*/
AxCallInfo_t  funcTable[]          =          {
     { "MyCategory", ADDNUM2, "ADDNUM2", "addnum2()", TRUE },
     { NULL, NULL, NULL, NULL, NULL }
};

/*    Function AxGetCallInfo returns the function table.
*/
DLL_EXPORT AxCallInfo_t *AxGetCallInfo()
{
     return(funcTable);
}
```

```
/*=======================================================*/
/* ADDNUM2 - This function adds an array of numbers                  */
/*=======================================================*/

elfData ADDNUM2(argsP)
     elfData argsP;                  /* argsp is an array of arguments passed from ELF */
{
     int counter, number;
     double val, total=0;
     elfData retP, firstP;

/* make sure at least two arguments are passed to this function */

     number = AxArraySize(argsP);
     if (number < 2)
          AxError(1, "Too few arguments", "ADDNUM2");

/* Add the numbers in the passed array, and return the total */

     for (counter = 0; counter < number; counter++)
     {
          firstP = AxArrayElement(argsP, counter);
          val = AxFloatFromDataPtr(firstP);
          total = total + val;
     }

     retP = AxMakeFloatData(total);
     return(retP);
}
```

You can now write a macro such as :

```
macro foo
        var arg
        arg = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
        info_message@(Addnum2(arg))
endmacro
```

Extracting Data From an ELF Array

The ELF / C interface contains functions that allow you to extract data from an array passed from an ELF macro. Two of these functions are used in Code Sample 3-4: AxFloatFromDataPtr() and AxArrayElement().

When you receive an ELF array in your C function, you must perform a series of steps to retrieve data from the array.

1. Declare the passed argument as elfData. All of the elements of the array are elfData values when they are initially received by the C routine.
2. Check the numbers of elements in the array with the function AxArraySize().
3. Extract an element from the array.
4. Convert the extracted element to a C data type.

Functions for Unpacking Arrays

The following are prototypes for functions that are useful for unpacking ELF data arrays. These functions can be used in RPCs or shared libraries.

```
elfData     AxArrayElement(elfData arrayData, int index)
elfData     AxArrayFromArray(elfData arrayData, int index)
```

```
int       AxBinaryFromArray(elfData arrayData, int index, int * binp)
int       AxBoolFromArray(elfData arrayData, int index)
double    AxFloatFromArray(elfData arrayData, int index)
int       AxIntFromArray(elfData arrayData, int index)
char      *AxStrFromArray (elfData arrayData, int index)
```

Some of these function perform two steps: they retrieve an ELF data element from the array, and convert that element into the appropriate C data format. For example, the function AxFloatFromArray() is the same as the following statement:

```
return(AxFloatFromDataPtr(AxArrayElement(data, index)));
```

Functions that Convert elfData to C Data Types

There are several functions that convert ELF data to C data types. These functions only work on a single ELF datum. They do not extract elements from ELF data arrays. The following lists the prototypes of these functions:

```
int       AxBinaryFromDataPtr(elfData data, char **binP)
int       AxBoolFromDataPtr(elfData data)
double    AxFloatFromDataPtr(elfData data)
int       AxIntFromDataPtr(elfData data)
char      *AxStrPtrFromDataPtr(elfData data)
```

Testing Functions

In many programs, you always know the type of data received from ELF. However, there are times when your C program must determine what type of data is being sent to it. The ELF / C interface contains the following testing functions:

```
int AxIsArray(elfData data)
int AxIsBinary(elfData data)
int AxIsFloat(elfData data)
int AxIsInt(elfData data)
```

```
int AxIsNumber(elfData data)
int AxIsString(elfData data)
```

These functions return TRUE or FALSE (1 or 0) values. Most of the testing functions are only used when you want to test the data before it is converted. By performing your own tests, you can prevent conversion errors, either by skipping the data item or generating an error condition and exiting the code.

Addnum3: Multi-Dimensional Arrays

Addnum1() and Addnum2() both assume that the array passed from ELF is one-dimensional. If you were trying to add up the values in a set of spreadsheet ranges, both of these functions would fail because the data in a spreadsheet range is passed as a multi-dimensional array. See the Preface of the *ELF Spreadsheets Macros Reference* for more information. The next function, Addnum3(), adds values from multi--dimensioned variables.

```
/*
*    Code Sample 3-5. This C module uses a recursive function to add all the elements
*    in a multi-dimensional array.
*/
#include "elfapi.h"
#define TRUE 1

elfData ADDNUM3();

/*
*    Define the function table
*/
AxCallInfo_tfuncTable[] =          {
     { "MyCategory", ADDNUM3, "ADDNUM3", "ADDNUM3()", TRUE },
     { NULL, NULL, NULL, NULL, NULL }
};

/*
*    Function AxGetCallInfo returns the function table.
*/
DLL_EXPORT AxCallInfo_t  *AxGetCallInfo()
{
     return(funcTable);
}

/*
*    SumArray() is a recursive function that adds the numbers in a multi-dimensional array
*    no matter what the dimensions of the array might be. This function is called from the
*    Addnum3 function. It is not declared in the function table, and cannot be called from
*    ELF.
*/
float SumArray(data)
elfData data;
{
     int num, i, numArgs;
     float result = 0;

     if (AxIsNumber(data))
          return (AxFloatFromDataPtr(data));
```

```
    if (AxIsArray(data))
    {
        numArgs = AxArraySize(data);
        for (i = 0; i < numArgs; i++)
            result = result + SumArray(AxArrayElement(data,i));
    }
        return (result);
}

/*
*   ADDNUM3() is a function that is called from ELF. ADDNUM3() uses the SumArray()
*   function to add all the elements in a multi-dimensional array.
*/

elfData ADDNUM3(funcData)
elfData funcData;

{
    elfData rData = NULL;
    float result;

    result = SumArray(funcData);
    rData = AxMakeFloatData(result);
    return (rData);
}
```

All the work of breaking apart the array into its elements is done in the SumArray() function. The function uses recursion to extract individual array elements.

Addnum4: Using AxCube Functions

Add-in functions written in C are very commonly used with Applixware Spreadsheets. There are three C-language functions that are specifically designed for add-in functions that process spreadsheet data. They are:

- void *AxInitCube(aP, numSheets, numRows, numCols) returns a handle.
- elfData AxCubeElement(handle, ix, jx, kx) to access a datum from the array.
- void AxFreeCube(handle) frees the handle.

To use these functions, follow these steps:

1. Call AxInitCube() with the passed elfData. AxInitCube() returns a handle. When the call completes, the numsheets, numrows, and numCols pointers contain the number of sheets, the number of rows, and the number of columns, respectively.

   AxInitCube can handle any data up to 3-dimensions. If the data is more than 3-dimensions it returns a null.

2. Set up three loops using the number of sheets, number of rows, and number columns.
3. Call AxCubeElement(handle, ix, jx, kx). This call returns the array items. Always use three loops regardless of the dimensions of the ELF array.
4. Call AxFreeCube(handle) to free the handle before you return from the function.

Code Sample 3-6 shows a function where these calls are used.

```
/*
```

```
*     Code Sample 3-6. This C module uses the AxCube functions to break down
*     an array passed from Spreadsheets.
*/
#include "elfapi.h"
#define TRUE 1

elfData ADDNUM4();

/*
*  Define the function table
*/
AxCallInfo_t       funcTable[]    =    {
   { "MyCategory", ADDNUM4, "ADDNUM4", "ADDNUM4()", TRUE },
   { NULL, NULL, NULL, NULL, NULL }
};

/*
*  Function AxGetCallInfo returns the function table.
*/
AxCallInfo_t  *AxGetCallInfo()
{
   return(funcTable);
}

elfData ADDNUM4(funcData)
elfData funcData;          /* is an array of args that the function was called with */
{
          void *handle;
          int ix, jx, kx, numRows, numCols ,numSheets;
          elfData dP;
          double val=0.0;
          elfData argP;


          argP = AxArrayElement(funcData, 0);
          handle = AxInitCube(argP, &numSheets, &numRows, &numCols);

          /*
          *  If the array passed from ELF is more than three dimensions,
```

```
        *  AxInitCube() returns a NULL.
        */

        if (NULL == handle)   {
                /* handle error */
                AxError(99, "AxInitCube returned error", NULL);  /* will not return */
        }

        /*
        *  If the handle is not NULL, then the ELF array passed to the
        *  function is an array of three dimensions or less. Processing
        *  of the data occurs in the loops that follow.
        */

        val = 0;
        for (ix=0 ; ix < numSheets; ix++)    {
                for(jx=0; jx < numRows; jx++) {
                        for (kx=0; kx < numCols; kx++)  {
                                dP = AxCubeElement(handle, ix, jx, kx);
                                val += AxFloatFromDataPtr(dP);
                        }
                }
        }
        AxFreeCube(handle);
        return(AxMakeFloatData(val));
}
```

Addarrays: Returning an Array to ELF

All of the examples in this section have passed a single value back to the ELF macro that calls them. Code Sample 3-7 shows how to pass an entire array back to ELF.

In this program, the client ELF function sends two arrays to C. The C program returns an array that adds the corresponding array elements together. For example:

```
Array0 = 1, 2, 3          ' This is passed from ELF to C
Array1 = 4, 5, 6          ' This is passed from ELF to C
Array2 = 5, 7, 9          ' This is passed from C to ELF
```

```
/*
*    Code Sample 3-7. This C module contains the routine ADDARRAYS() which adds
*    corresponding array elements together.
*/
#include "elfapi.h"
#define TRUE 1

elfData ADDARRAYS();

/*
*    Define the function table
*/
AxCallInfo_t  funcTable[]          =          {
     { "MyCategory", ADDARRAYS, "ADDARRAYS", "ADDARRAYS()", TRUE },
     { NULL, NULL, NULL, NULL, NULL }
};

/*   Function AxGetCallInfo returns the function table
*/
DLL_EXPORT AxCallInfo_t *AxGetCallInfo()
{
     return(funcTable);
}

elfData ADDARRAYS (funcData)
elfData funcData;

{
     elfData rData = NULL;
```

```
    int numArgs1, numArgs2, counter,  numArgs;
    double val;
    elfData array1, array2, array3;

    array1 = AxArrayElement(funcData,0);
    array2 = AxArrayElement(funcData,1);

/* Determine which array is larger and set  numArgs to its value */

    numArgs1 = AxArraySize(array1);
    numArgs2 = AxArraySize(array2);
    if (numArgs1 > numArgs2)
        numArgs = numArgs1;
        else
        numArgs = numArgs2;

/* Create a third array that will be populated and returned to ELF      */

    array3 = AxMakeArray(0);

/* Copy to the limits of the smaller array to array3 */

    for (counter = 0; counter < numArgs; counter++)
    {
        val = AxFloatFromArray(array1, counter) + AxFloatFromArray(array2, counter);
        array3 = AxAddFloatToArray(array3, counter, val);
    }

    rData = array3;
return (rData);
}
```

Calling ADDARRAYS from ELF

ADDARRAYS returns an array to the calling ELF macro. The following macro calls ADDARRAYS as an RPC, and displays the array returned from the C function in a window.

```
/*
*  Code Sample 3-8. This macro calls ADDARRAYS() as
*  an RPC.
*/
macro rpctest
var args, moreargs, arg2
RPC_CALL@("/newuser/applix/shlib/addnum4", 6162)
/*
* Define two arrays to pass to ADDARRAYS()
*/
        args = 1, 2, 3, 4, 5
        moreargs = 5, 4, 3, 2, 1

        arg2 = ADDARRAYS(args, moreargs)
/*
*       DUMP_ARRAY@ displays the contents of an array
*       in a window.
*/
        DUMP_ARRAY@(arg2)
endmacro
```

NOTE: The conversion routine AxFloatFromArray returns a zero when the argument is an array element that is out-of bounds. For example, the sixth element in an array that has only five elements is assumed to have a value of zero. This means that the ADDARRAYS function can process arrays that are different sizes.

Returning Data to ELF

Only one data item can be returned to an ELF macro from C. However, the data item can be an array. In addition, the items in the array do not have to be homogenous - you can pack different types of data into the array, and return it to ELF.

In your C code, Use AxMakeArray() to create elfData arrays that are returned to ELF from C. The AxMakeArray function dynamically manages the space that it needs. In Code Sample 3-6 for example, array3 is a pointer to the memory that is allocated for the array. Each time an element is added to the array, a new pointer to this data is returned. This pointer changes each time an element is added to the array.

The following functions are used when building data arrays to be sent back to ELF:

```
elfData AxAddArrayToArray(elfData data, int index, elfData ptr)
elfData AxAddBinaryToArray(elfData data, int index, int dataSize,
                                        char *binaryData)
elfData AxAddBoolToArray(elfData data, int index, int n)
elfData AxAddDataToArray(elfData data, int index, elfData ptr)
elfData AxAddFloatToArray(elfData data, int index, double n)
elfData AxAddIntToArray(elfData data, in index, int n)
elfData AxAddStrToArray(elfData data, int index, char *str)
```

Notice that each function returns an elfData pointer. You use the AxMakeArray to return a pointer to the array. As you add elements to the array, AxMakeArray returns new values for this pointer.

4 Creating an RPC Function

This chapter describes the ELF/RPC interface. This interface is primarily used to develop and debug shared libraries. However, it can be used in place of shared libraries.

This chapter describes a set of functions that adds numbers together. The first program is very limited in scope. This program is extended in a series of steps so that most of its limitations are removed.

The examples in this chapter contain many ELF RPC macros and Applixware RPC C language functions. These functions use the older AxDispatch structure. This structure is still supported for backward compatibility, but Applix recommends structuring new add-in functions according to the parameters described in Chapter 2.

AddNums: Version 1—Basic Features

To create a C language function that receives two numbers from ELF and then returns the value of the two numbers added together requires that you create:

- An ELF macro that calls the C language function.
- A C language function that receives the data and sends it back to your ELF macro.
- A header file that contains information that will be used by the ELF macro and the C language function.

The ELF Macro

The first version of AddNums is a shell that sends and receives data.

```
include "add.h"

macro AddNums(num1, num2)
    var arg

    arg = num1, num2
    return (RPC_CALL@(SERV_NAME, ADD_NUMS, arg))
endmacro
```

The AddNums macro:

- Includes the constants contained within the add.h file.
- Combines its two arguments into one array.
- Calls the C language program whose identifier is SERV_NAME. Because the SERV_NAME program could perform more than one

action, the ADD_NUMS constant identifies which action is performed.

- Returns the value returned by the RPC function.

The RPC_CALL@ macro is used when a macro calls an RPC function once. RPC_CALL@ is not an efficient way to use system resources if you are going to repeatedly make calls to a function.

The add.h Header File

The header file add.h is used by the AddNums macro and the C language program. Add.h contains the following lines:

```
#define SERV_NAME "/user/applix/bin/dispatch"
#define ADD_NUMS 1
```

These lines define two constants: SERV_NAME and ADD_NUMS. SERV_NAME is the name of the C language program that contains the functions. ADD_NUMS identifies the function being executed within that program.

NOTE: Values less than or equal to zero are reserved by Applix.

The C Language Program

The following listing shows the complete C language program that receives the AddNum information from ELF and sends back the computed result.

```
#include "elfapi.h"
#include "add.h"

elfData AxDispatch (funcId, funcData)
int funcId;
elfData funcData;
```

```
{
  elfData rData = NULL;

  switch (funcId)
        {
             case ADD_NUMS:
             {
                  int num1, num2;
                  AxDecomposeArray(funcData,"ii", &num1, &num2);
                  rData = AxMakeIntData(num1+num2);
                  break;
             }
        }
  return (rData);
}
```

The structure of this function is typical for the AxDispatch RPC functions. The common elements are as follows:

- All RPC functions include elfapi.h file. This file contains the function prototypes for all Applixware functions that you will be using.
- The function must be declared as shown in this example. Its name must be AxDispatch. (The functions described in Chapter 2 can have any name, but if you do not define a function table, you must name the function AxDispatch.)
- All functions should examine the funcId argument using switch or if ... else if ... statements. Because the ELF / RPC system can send its own function IDs, you must check to insure that your program is only responding to data your data requests. If you assume that what is received is *only* what your program sent, a severe problem could occur.

  NOTE: The funcID constants must have a value greater than zero. Negative values and zero are reserved by Applix.
- All functions must return something to the calling ELF macro, even if the returned value is NULL.

This function uses two Applixware convenience functions. The first, AxDecomposeArray, transforms the data from the form in which it was sent to your function into a form usable by a C program. Similarly, the AxMakeIntData transforms C data into a form that can be used by ELF.

AxDispatch Function

The ELF / RPC interaction in AddNums is handled in a C-language function named AxDispatch. This function accepts two arguments:

- funcID is the integer command code for the function as it is defined in a header file shared with the ELF application.
- funcData is the data passed from ELF to the C function. It is passed as an opaque data structure.

The AxDispatch function returns an elfData datum created by C functions such as AxBuildArray. These routines are described later in this chapter.

Here is a synopsis of what an AxDispatch function might look like:

```
#include "elfapi.h"
#include ‘‘func_defs.h’’
elfData AxDispatch (funcId, funcData)
int funcId;

elfData funcData;
{
elfData retValue = NULL;
        switch (funcId)
        {
                case FUNC_1:
                retValue = firstFunction(funcData);
                break;
```

```
                case FUNC_2:
                retValue = secondFunction(funcData);
                break;
/*
* Process Other function IDs here
*/
        }
        return (retValue);
}
```

It is very important to notice that the switch statement does not have a default clause. If you include a default clause to the switch statement, its sole function must be to return a NULL value to the calling ELF function.

The elfData data type received from and sent to ELF is ‘‘opaque’’. Use the functions provided by Applix to manipulate this information. Do not examine the data passed between ELF and C directly.

AxDispatch Switch Values

The constants used to represent function names within the switch statement must have values greater than or equal to 1. Values less than 1 are reserved by Applix for its own use. Some of the Applixware reserved values have special meaning and can be used in certain circumstances:

0 Can be used to initialize your environment.

-2 Provides C servers access to axnet services. If you want a gateway to function as an axnet service, you should return an array of two elements in this switch case. The first element is the desired axnet service name. The second element is a Boolean value that if TRUE specifies that the service is private (and should not be listed when the user calls the LIST_OF_SERVICES@ macro. If a gateway calls this option, it can be connected to using axnet and called using AXNET_RPC@ and AXNET_RPC_CHANNEL@ macros.

-3 Allows C servers to choose what port they will listen on rather than obtaining the port number passed on the command line. The server should return an elfDatum number from AxDispatch. An example would be a program that gets its port from /etc/services by calling getservbyname().

AxDecomposeArray and AxBuildArray

The AddNums macro has clearly defined arguments. It always sends two integer arguments. In a similar fashion, the data returned by a C function may be just as well-defined as in this example.

If your arguments are clearly defined, you can use the AxDecomposeArray function to transform the received array into a form usable by C. Similarly, you can use the AxBuildArray function to create an array of information to return to your ELF program.

Prototypes for these functions are as follows:

```
void    AxDecomposeArray(funcData, format, *arg1, *arg2,...,
        *argN)
elfData AxBuildArray(format, arg1, arg2,..., argN)
```

(The actual prototypes found in the elfapi.h differ because they use the C language variable argument syntax.) The format argument is a null-terminated string containing any number of the following characters:

b argN is the address of an integer (int *). If the passed integer is non-zero (especially ELF's -1 representing TRUE), argN is assigned a value of 1.

c argN is the address of a string buffer (char *). Text will be copied to this buffer. The next argument (argN+1) must specify the size of the buffer.

d argN is the address of a double precision variable (double *).

f argN is the address of a floating point variable (float *).

i argN is the address of an integer (int *).

s argN is the address of a string pointer (char **).

For example, the following declaration sets a character and integer value:

```
AxDecomposeArray(funcData, "ci", charBuf,30, &num)
```

The only difference between the formats used by the AxDecomposeArray and AxBuildArray functions are:

- s and c mean the same thing.
- b converts all non-zero values to ELF's TRUE value (which is -1).

Creating ELF Data

The AxBuildArray function is used when more than one value is being returned to ELF. In many cases, you will only be returning one value. In this example, only an integer was being returned. This meant that the AxMakeIntData function could be used.

Here are the prototypes of other functions that you can use:

```
elfData AxMakeBinaryData(int num)
elfData AxMakeFloatData(double num)
elfData AxMakeIntData(int num)
elfData AxMakeStrData(int len, char *str)
```

Compiling Your C Program

Assuming that your C code is in a file named dispatch.c in your current directory, you can compile the file as follows:

```
cc -g -o dispatch dispatch.c /applix/axexec/elf/elfapi.a -lm
```

NOTE: This example assumes you are compiling on a Sun OS workstation. See Chapter 2 for information on compiling RPC functions in other environments.

You are now ready to execute the AddNums macro. For example:

```
macro foo
        info_message@(AddNums(17, 23))
endmacro
```

After executing this macro, ELF displays the value 40 in a message box.

AddNums: Version 2—Reducing ELF RPC Overhead

Although the AddNums macro is designed to add two numbers together, it can actually be used to add many numbers together. For example:

```
AddNums(17, AddNums(38, 426))
```

This example adds three numbers together. When it executes, the RPC_CALL@ macro is executed twice. Each of these times, the following operations occur:

- A channel is opened or ELF searches for the correct channel to send data.
- Data is sent to a C program.
- Data is received from a C program.

In terms of system resources and time, opening and checking for channels is expensive. The following rewrite of the AddNums macro shows how you can eliminate some of this overhead.

```
include "add.h"

macro AddNums(num1, num2)
    var arg

    arg = num1, num2
    return (RPC_CHANNEL_CALL@(GetChannel(), ADD_NUMS,
arg))
endmacro

function GetChannel
    var channel

    channel = SYSTEM_VAR@("add_num_channel")
    if (IS_NULL@(channel))
    {
            channel = RPC_CONNECT@(SERV_NAME)
            RPC_CHANNEL_MASTER@(channel,
                        ELF_TASK_ID@())
            SET_SYSTEM_VAR@("add_num_channel", channel)
    }
    return (channel)
endfunction
```

Only one change was made to the AddNums macro: RPC_CALL@ was replaced with RPC_CHANNEL_CALL@. The difference between these two macros is that RPC_CHANNEL_CALL@ only sends and receives data over an already existing channel. This channel number is supplied by the GetChannel function, which does one of the following:

- If a channel is open, it returns the channel number.
- It calls RPC_CONNECT@ to open a channel. This channel's number is returned.

The RPC_CHANNEL_MASTER@ macro tells Applixware what it should do with the channel when a task finishes executing. If you send the current task's task number, the C program will terminate after the named task terminates. A zero argument indicates that the channel should be left open.

When a function is started using RPC_CONNECT@ or RPC_CALL@ , it will continue to execute after the calling task stops. Using RPC_-CHANNEL_MASTER@ in this example indicates that the task should stop executing when the ELF task terminates.

At this time, you can recompile the ELF file and run AddNums. It is not necessary to recompile the C program because no changes need to be made.

AddNums: Version 3—Calling Different C Routines

While the AddNums macro can add any number of arguments by calling it within itself, it might be easier if the macro were a bit more general. For example, clever solutions could be created by using recursion. However, suppose you know that no more than four arguments will ever be used. In this case, you could try the following rewrite:

```
include "add.h"

macro AddNums(num1, num2, num3, num4)
        var arg

        if (IS_NUMBER@(num4))
        {
             arg = num1, num2, num3, num4
```

```
            return (RPC_CHANNEL_CALL@(GetChannel(),
                                ADD_4, arg))
        }
        else if (IS_NUMBER@(num3))
        {
            arg = num1, num2, num3
            return (RPC_CHANNEL_CALL@(GetChannel(),
                                ADD_3, arg))
        }
        else if (IS_NUMBER@(num2))
        {
            arg = num1, num2
            return (RPC_CHANNEL_CALL@(GetChannel(),
                                ADD_NUMS, arg))
        }
        else if (IS_NUMBER@(num1))
            return (num1)
        else
            ERROR@(1,"No numeric arguments")
endmacro
```

This macro now calls the same C function in three different ways. The number of non-null arguments determines which way the C function is called.

No changes will be made to the GetChannel function.

The add.h Header File

The add.h header file must be changed to include the two new constants. The new file is as follows:

```
#define SERV_NAME "/user/applix/axhome/macros/dispatch"
#define ADD_NUMS 1
#define ADD_3 3
#define ADD_4 4
```

The numeric values used for the define constants are arbitrary. The values of 3 and 4 were picked to be similar to the ADD_3 and ADD_4 names.

The C Language Program

The C language program must be changed so that it can process the two new constants. Here is the program:

```
#include "elfapi.h"
#include "add.h"

elfData AxDispatch (funcId, funcData)
int funcId;
elfData funcData;

{
        elfData rData = NULL;

        switch (funcId)
        {
             case ADD_NUMS:
             {
                  int num1, num2;
                  AxDecomposeArray(funcData,"ii", &num1, &num2);
                  rData = AxMakeIntData(num1+num2);
                  break;
              }
             case ADD_3:
             {
                  int num1, num2, num3;
                  AxDecomposeArray(funcData,"iii", &num1, &num2,
                                        &num3);
                  rData = AxMakeIntData(num1+num2+num3);
                  break;
             }
             case ADD_4:
```

```
            {
                int num1, num2, num3, num4;
                AxDecomposeArray(funcData,"iiii", &num1, &num2
                                        &num3, &num4);
                rData = AxMakeIntData(num1+num2+num3+num4);
                break;
            }
        }
        return (rData);
}
```

This program is compiled in the same way as the previous version.

AddNums: Version 4—Invoking from a Spreadsheet

The AddNums macro can add two, three, or four integer numbers when called from ELF. However, it does not work when called from within a spreadsheet. Because of the way tasks interrelate, Applix Spreadsheets wants the channel to remain open. In this way, it can efficiently call your macro whenever it performs a recalculation.

The only change you will need to make is to the GetChannel macro. Here is a revised version of this macro:

```
function GetChannel
    var channel

    channel = SYSTEM_VAR@("add_num_channel")
    if (IS_NULL@(channel))
    {
            channel = RPC_CONNECT@(SERV_NAME)
            RPC_CHANNEL_MASTER@(channel, 0)
```

```
            SET_SYSTEM_VAR@("add_num_channel", channel)
    }
    return (channel)
endfunction
```

In this example, RPC_CHANNEL_MASTER@ is called with an argument of 0, which means keep the channel open when the task completes executing.

AddNums: Version 5—Passing an Array of Numbers

A slightly more realistic version of the AddNums macro would pass any number of numbers to C and return a value back to ELF.

The ELF Program

If the purpose of the ELF macro is simply to send an array of information, the ELF side is very simple and is as follows:

```
macro AddNums(anArray)
        return (RPC_CHANNEL_CALL@(GetChannel(), ADD_NUMS,
                                  anArray))
endmacro
```

The GetChannel macro would not be changed. The add.h header file would be the same as was used in the first version of AddNums.

The C Program

While the fundamental structure of the C program is unchanged, it will now need to unpack the array.

```
#include "elfapi.h"
#include "add.h"

elfData AxDispatch (funcId, funcData)
int funcId;
elfData funcData;
{
    elfData rData = NULL;

    switch (funcId)

    {
    case ADD_NUMS:
        {
            int result = 0;
            int numArgs, counter;

            numArgs = AxArraySize(funcData);
            for (counter = 0; counter < numArgs; counter++)
                result += AxIntFromArray(funcData, counter);
            rData = AxMakeIntData(result);
            break;
        }
    }
    return (rData);
}
```

You can now write a macro such as :

```
macro foo
    var arg
    arg = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    info_message@(AddNums(arg))
endmacro
```

AddNums: Version 6—Checking Data Type

While the AddNums function is becoming closer to a general purpose function that can add any number of elements, one limitation is that it can only add integers. The next version of the C program solves this problem.

```
#include "elfapi.h"
#include "add.h"

elfData AxDispatch (funcId, funcData)
int funcId;
elfData funcData;

{
    elfData rData = NULL;

    switch (funcId)
    {
        case ADD_NUMS:
        {
            double result = 0;
            int numArgs, counter;
            elfData temp;

            numArgs = AxArraySize(funcData);
            for (counter = 0; counter < numArgs; counter++)
            {
                temp = AxArrayElement(funcData, counter);
                if (AxIsNumber(temp))
                    result += AxFloatFromDataPtr(temp);
                else
                    AxError(2,"Bad Data Type");
            }
            rData = AxMakeFloatData(result);
```

```
            break;
        }
    }
    return (rData);
}
```

This version only differs slightly from the previous version. The differences are:

- AxArrayElement is used so that the individual elements of the funcData array can be examined. Because the program is directly manipulating an element, the program uses the ...DataPtr functions rather than the ...FromArray functions.
- A data type testing function is used.

Data Conversion

The AxIsNumber test actually represents poor programming practice. In this example, if the data type of the data contained within temp were the string "2", the value would not be added to the result. In addition, an error would be thrown ending the program.

If you had let the AxFloatFromDataPtr function make the conversion without checking for the data type, AxFloatFromDataPtr would throw an error if it could not make the conversion. This leads to the seeming contradiction that while you can check for a data's type, there is seldom any reason to make the check.

Here is how the for loop in the previous example should be coded:

```
for (counter = 0; counter < numArgs; counter++)
{
    temp = AxArrayElement(funcData, counter);
    result += AxFloatFromDataPtr(temp);
}
```

The next section will use one of the type testing functions, AxIsArray to control how data is unpacked. This is the only function used with regularity.

AddNums: Version 7—Multi-Dimensional Arrays

The previous version of the AddNums program assumed that the array was a vector; that is, it was a one-dimensional array. If you were trying to add up the values in a set of spreadsheet ranges, the previous function would fail. (It fails because the data in the range is passed as a multi-dimensional array. This is discussed in the Preface of the *ELF Spreadsheets Macros Reference.*) The next version of AddNums will add values from multi-dimensioned variables.

```
#include  "elfapi.h"
#include "add.h"

float SumArray(data)
elfData data;

{
     int num, i, numArgs;
     float result = 0;

     if (AxIsNumber(data))
          return (AxFloatFromDataPtr(data));

     if (AxIsArray(data))
     {
          numArgs = AxArraySize(data);
          for (i = 0; i < numArgs; i++)
               result = result + SumArray(AxArrayElement(data,i));
```

```
    }
        return (result);
 }

elfData AxDispatch (funcId, funcData)
int funcId;
elfData funcData;

{
    elfData rData = NULL;

    switch (funcId)
    {
        case ADD_NUMS:
        {
            float result;

            result = SumArray(funcData)
            rData = AxMakeFloatData(result);
            break;
        }
    }
    return (rData);
}
```

All the work of breaking apart the array into its elements is done in the SumArray function. Other than the use of recursion to walk through the array, nothing new was added.

Version 1 of AddNums could only add two numbers. This is a rather simple problem that would never be solved using the ELF/RPC interface. However, Version 7 represents a program that you might want to actually use. While it could be done efficiently using ELF, the C language version would be more efficient if the range contained a lot of data. Also notice that the ELF side could pack as many ranges as were selected by the user into one macro and send the resulting array to the AddNums function.

Error Handling

The previous versions of the AddNums program did not contain much error checking code. Here is a version that shows how you would incorporate error handling into this macro:

```
include "add.h"
include "errors_.am"

macro AddNums(range1, range2)
    var arg, error_num
    arg = range1, range2

    on error
    {
        error_num = ERROR_NUMBER@()
        if error_num = ERR#ERR_FEEPIPE_ or
            error_num = ERR#ILLVAL_
        {
            channel = RPC_START_SERVER@(SERV_NAME)
            RPC_CHANNEL_MASTER@(channel,0)
            SET_SYSTEM_VAR@("add_num_channel", channel)
        }
    }

    return (RPC_CHANNEL_CALL@(GetChannel, ADD_NUMS, arg))
endmacro
```

If a problem occurs with the channel, a new channel will need to be opened. For example, if a problem had occurred on the C side and the program crashed, the existing channel would still exist; however, it would not be usable. Using the RPC_START_SERVER@ macro gets around this problem as it always returns a new channel. In contrast, the RPC_CONNECT@ macro will try to check to see if it can return an existing channel.

After the error handler executes, it is disabled. This means that the on error statements will not execute if the problem was not fixed. In this

case, the error would filter up a layer with ELF so that an error message is thrown. (If you reenabled the error handler, you would probably put the program into an infinite loop if an error occurred.)

AddNums: Version 8—Returning an Array to ELF

All previous versions of AddNums have shared one trait: only one value was returned. The following C function shows a radically different version of AddNums. In this program, the client ELF function will send two arrays to C. The returned result is an an array that adds the comparable array elements together. That is:

arrayNew[i] = array1[i] + array2[i]

for all values in both arrays.

```
#include "elfapi.h"
#include "add.h"

elfData AxDispatch (funcId, funcData)
int funcId;
elfData funcData;
{
    elfData rData = NULL;

    switch (funcId)
    {
        case ADD_ARRAYS:
        {
            int numArgs1, numArgs2, counter,  numArgs;
            double val;
            elfData array1, array2, array3;
```

```
                array1 = AxArrayElement(funcData,0);
                array2 = AxArrayElement(funcData,1);

/* Determine which array is larger and set numArgs to its value */
                numArgs1 = AxArraySize(array1);
                numArgs2 = AxArraySize(array2);
                if (numArgs1 > numArgs2)
                     numArgs = numArgs1;
                else
                     numArgs = numArgs2;

                array3 = AxMakeArray(0);

/* Copy to the limits of the smaller array to array3 */
                for (counter = 0; counter < numArgs; counter++)
                {
                     val = AxFloatFromArray(array1, counter) +
                               AxFloatFromArray(array2, counter);
                     array3 = AxAddFloatToArray(array3, counter, val);
                }

                rData = array3;
                break;
                }
          }
     return (rData);
}
```

In all previous examples, only one value was returned. Here, an array of values is returned. While the requirement still exists that only one data item be returned, the data item can be multi-valued. In addition, the returned item need not be homogenous. That is, different items can be of differing data types.

Because the AxFloatFromArray function is used, the fact that out-of--bounds array referencing is occurring can be ignored. That is, the conversion routines within AxFloatFromArray know how to deal with an

out-of-bounds reference. Specifically, an element that does not exist is assumed to have a value of zero. This means that the program does not have to worry that one array is smaller than the other.

Use the AxMakeArray function to create the multi-valued elfData datum that will be returned. The AxMakeArray function dynamically manages the space that it needs. For example, array3 is simply a pointer to the memory that is allocated for the array. Each time an element is added to the array, a new pointer to this data is returned. This pointer can (and probably will) differ each time an element is added to the array.

Declaring the number of elements that will be created when you first call AxMakeArray is more efficient than dynamically resizing the array being created every time you create the array. However, both methods are available. You even have the freedom to create a fix-sized array; then, if circumstances require it, you can add elements to it.

Passing ELF data arrays

Addnums8 passes an array of floting point numbers back to the calling macro. The arrays you pass to ELF do not have to be heterogenous. They can contain floating point number, integers, characters, or any other data. They can also be an array, format, an array of formats, and so on. Only one ELF data element is sent from C to ELF, but this element can be as complex as it needs to be.

The transformation of a set of arguments into one argument is the step that causes the most problems. Not only must single elements be combined into elements of an array, but elements must be added as well.

For example, suppos element 0 is a string, element 1 is a 3-element array, element 2 is an integer, and element 3 is a formatted variable containing 4 elements. The contents of this array are as follows:

0: string	array[0]
1: 3-element array	array[1,0] array[1,1] array[1,2]
2: integer	array[2]
3: 4-element format	array[3,0] array[3,1] array[3,2] array[3,3]

When this datum is sent to C, it will be received as a 4-element array. The C program will need to understand what the contents of each array items are and how they are transformed to be used within C.

The transformation of a formatted variable to a C structure must also be managed. For example, if a 6-element format is sent to C, it is received as an array of 6 items. If these items are to be placed into a structure, each item must be individually extracted and assigned to the structure.

Returning Data

The most central construct of client / server interactions is that the client asks the server to do something and the server responds. In the Applixware implementation of this interaction, the server must always send something back to the client. However, sending NULL back to the client is acceptable.

In some RPC implementations, you can call the server in such a way that the client does not need to wait for a reply to be sent back. If you are in a situation where you do not want to wait for the completion of the task, you can begin by spawning a new task and have that task

make the RPC call. While the task making the call will be blocked waiting for a reply, you can continue executing. When the blocked task becomes unblocked, it can signal that it is complete. For example, it could initialize a system variable that the original task watches or it could send a poke.

Localization

In all of the versions of the AddNums ELF macro in this chapter, very little work was performed in ELF. Instead, most of the work was deferred to the ELF/RPC add-in function.

The first rule in creating an RPC is that you want to minimize the data sent from one process to another.

The simplicity of the AddNums example could suggest that you would write your ELF macros in the same way for RPC use as you would without RPC interactions. However, it often occurs that a set of related ELF macros share some general characteristics. For example, if you were defining five ELF functions that would interact with one C program, all functions would share an error handler. It is also possible that the data used needs to be handled in a uniform way.

For example, suppose you are building a system that requires five functions to communicate with C functions. All other ELF functions that you would be creating are to be normal ELF functions. In this case, you might build your functions as follows:

```
define ADD_2 2
define ADD_3 3

macro func_1(arg1, arg2)
    ...
```

```
    return(my_call_func(ADD_2, arg1, arg2))
endmacro

macro func_2(arg1, arg2, arg3)
    ...
    return (my_call_func(ADD_3, arg1, arg2, arg3)
endmacro

macro func_3...

macro func_4...

macro func_5...

function my_call_func(whichFunc, arg1, arg2, arg3, arg4, arg5, arg6,
                      arg7)
    var arg
    ...

    on error
    {
        ...
    }
    case of (whichFunc)
    case ADD_2
        arg = arg1, arg2
    case ADD_3
        arg = arg1, arg2, arg3
    ...
    endcase
    return (RPC_CHANNEL_CALL@(GetChannel(), whichFunc, arg)
endfunction
```

At first glance, it may not appear that you have gained anything by building another layer. However, the benefits of localizing the calls into one macro outweigh the drawbacks. The first benefit is simplicity. The functions that are at the top layer look and feel like normal ELF functions.

The second benefit is that data must often be checked and prepared when it is sent to C and when it comes back from C. Placing these functions in one file ensures that these actions are handled in a uniform way and that when changes occur, they occur to all data being affected.

As you will see in the next chapter, a second method of interacting with C language functions exists. This method, which uses shared libraries, passes data between ELF and C in a different way. If the interface is within one file, you can replace the interaction macro instead of replacing every macro that talks to a C RPC function.

Non-local Machines

The examples that were shown earlier assumed that your C function and Applixware operated upon the same computer. Using additional arguments to RPC_CALL@, RPC_CONNECT@, and RPC_START_SERVER@ allows you to use server programs on other machines. Here, for example, is the complete prototype for the RPC_CALL@ macro:

```
[value=] RPC_CALL@(progName, cmdCode, passedData[,
                        portName[, machineName]])
```

where:

progName	The pathname of the server.
cmdCode	A unique integer number that defines the C function to be called. These command codes are usually defined in a header file that is included by your ELF program and by your C language program.

passedData — An ELF data element. It can be a single-valued data element such as an integer. It can also be an array, format, an array of formats, and so on.

portName — If no machineName is specified, the portName refers to the UNIX domain socket that the server uses to communicate with clients. This server should be an interactive server that can service more than one user. Normally, the server is started from the machine's boot sequence. However, it can also be a service that is started by the portmapper.

machineName — If specified, indicates the machine upon which progName will run. If this argument is omitted, the default is the current machine.

In a similar fashion, only the simplest form of the RPC_CONNECT@ macro was used. Its prototype also includes the portName and machineName arguments and is as follows:

RPC_CONNECT@(progName[, socketName[, machineName]])

The following table shows the kinds of values that can be used as arguments to the RPC_CONNECT@ macro:

Table 4-1 Connection State Values

prog-Name	socket-Name	machine-Name	Comments
NULL	NULL	NULL	Invalid.
String	NULL	NULL	The program will be a slave on the current machine.
String	String	NULL	Uses a named socket to communicate with a program already running as a server.
String	NULL	String	Not supported.

prog-Name	socket-Name	machine-Name	Comments
String	String	String	Uses customer socket name.

This table identifies five cases:

- The first line indicates an invalid combination; that is, you must use at least one argument when calling RPC_CONNECT@.
- The second line's combination lets you interact with a program that is running on the same machine that is running Applixware. Communication will occur using a named pipe. Applixware will start progName and will be slaved to your ELF task. By default, the task will remain running until axmain stops executing.
- The third line's combination lets you interact with a program that is already running on the same machine that is running Applixware. Communication will occur using the named socket. This socket can be a TCP/IP socket or it can be a named temporary file.

  The server is independent of axmain. That is, after axmain goes away, the server will continue to run.
- The fourth is not supported.
- The fifth allows you to run a program on a different machine. ELF and your program will communicate using the socket you have named. The socketName must be a TCP/IP socket number. The program runs independently of Applixware and must already be running when it is called. The running program must have established communciation with the named socket.

Whenever you are referring to an existing socket, the server is already running. If there is no socket, the service is started as a slave.

RPC Function Summary

The following is a brief overview of all RPC ELF functions:

RPC_CALL@(progName, cmd, data[, socketname[, host]])
: Calls an RPC function. This macro will open and close a channel. If the server is shared by many users, you must use this macro so that your task does not monopolize the channel.

RPC_CHANNEL_CALL@(channel, cmd, data)
: Calls an RPC function using an already open channel.

RPC_CHANNEL_MASTER@(channel, taskID)
: Sets the channel's ''existence'' state when the task stops executing.

RPC_CHANNEL_USE_HOURGLASS@(channel, hourglassFlag)
: Displays an hourglass during ELF and RPC calls.

RPC_CONNECT@(progName[, socketName[, host]])
: Connects to a channel or creates the initial channel connection. The channel number is returned. This returned value is used in most other RPC macros.

RPC_DISCONNECT@(progName)
: Closes a channel.

RPC_START_SERVER@(progName[, socketName[, host]])
: Starts a server process and returns a channel number. A new server is always created, even though the server may already be running or attached to a channel.

5 Network Communications

This chapter describes the network communications available with ELF. The following topics are discussed:

- The TCP/IP network
- Data transmission considerations
- Sending and receiving ELF arrays over a network
- Sending and receiving binary data over a network
- Closing channels and sockets
- A socket communications example

TCP/IP Network

ELF includes a collection of built-in macros for communicating among processes on a TCP/IP network. These macros allow you to send ELF data or binary data between machines on a network.

A typical network communication involves a machine designated as a server and one or more machines designated as *clients*. The server accepts messages from client machines and performs actions based on the messages it receives. In practice, you do not have to restrict a server to receiving messages and performing actions based on messages it receives; it can be used to send messages to initiate actions as well. The explanations and examples in this chapter, however, are based on the traditional server/client roles.

Communication is established by opening *sockets* on the server and client machines. Sockets are given socket identification numbers. If a server and a client are given identical socket identification numbers when the sockets are opened, a communication *channel* between the machines is opened.

A socket is opened on the server using the SOCKET_OPEN_SERVER@ macro. When this macro is executed, it waits for a connection from a client machine. When a connection is made, the macro returns a channel ID. A socket is opened on a client machine using the SOCKET_OPEN_CLIENT@ macro.

The SOCKET_WRITE@ macro is used to send an ELF array over the communications channel. The SOCKET_READ@ macro is used to read the array sent by the client. You can send a binary object over a communications channel using the SOCKET_WRITE_BINARY@ macro. The SOCKET_READ_BINARY@ macro is used to read the binary object sent over the network.

The following figure illustrates network communication concepts.

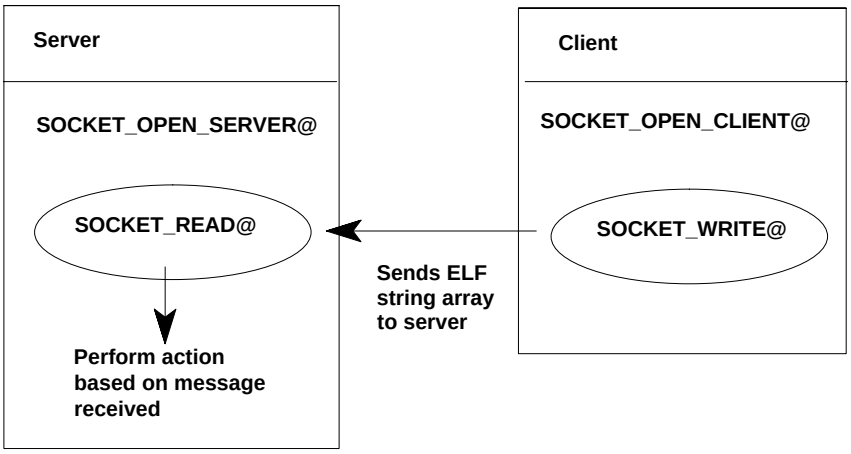


Figure 5-1 Network Communications

The macro documents axdata/eng/Demos/elfnet.am and axdata/eng/Demos/livedata.am provided with Applixware present examples of socket communications. You can use these example programs as guides to the use of the ELF communication macros.

A communication channel is closed using the SOCKET_CLOSE_CHANNEL@ macro. Usually, when a communication is completed, both the server and the client use SOCKET_CLOSE_CHANNEL@ to close the communication channel.

You can close a socket on a server using the SOCKET_CLOSE@ macro. It is not necessary to close sockets on client machines.

A typical communications macro sequence involves opening a channel, sending or receiving a message, and then closing the channel. This ensures that no interim processing can interfere with the communication. The following are examples of functions that could be used to send an ELF array between a client machine and a server.

```
' Server macro
```

```
FUNCTION ReceiveMsg(socket_number)
    VAR channel, message
    channel = SOCKET_OPEN_SERVER@(socket_number, -1)
    message = SOCKET_READ@(channel, -1)
    SOCKET_CLOSE_CHANNEL@(channel)
    RETURN(message)
ENDMACRO

' Client macro
FUNCTION SendMsg(the_array, hostname)
    VAR channel
    channel = SOCKET_OPEN_CLIENT@(socket_number, -1,
                hostname)
    SOCKET_CLOSE_CHANNEL@(channel)
ENDMACRO
```

Data Transmission Considerations

Using ELF, you can transmit data over a network as either ELF arrays or as binary objects.

Whenever possible, it is recommended that data be transmitted in the form of ELF arrays. ELF arrays provide a convenient method for transmitting data since ELF macros can use the information in ELF arrays directly; no byte-level data manipulation is necessary. Therefore, if you are creating ELF-based applications that involve the exchange of data over a network, using ELF arrays as the data carrier provides the easiest and most efficient way to transmit the data.

There might be instances, however, in which you need to transmit data that is not represented as an ELF array. For example, the designated server might need to transmit data from a database that has its own data format. In such instances you can use one of two strategies:

- Convert the data to ELF format. The following section describes the ELF data format.

- Transmit the data as a binary object rather than as an ELF array.

If you transmit the data as a binary object, the client macro used for receiving the data must understand the format in which the data is sent and manipulate it appropriately. For example, if the binary object represents a database data format in which the first two bytes specify the number of records, the second two bytes specify record length, and so on, then the receiving macro must interpret the bytes appropriately. ELF provides a set of macros for handling binary data; see the macro reference listings for information.

When you transmit data as an ELF array, ELF automatically interprets the data format so your macros do not have to interpret and manipulate data at the byte level. As a result, transmitting data using ELF arrays is more efficient than transmitting data using binary objects and then using built-in macros to interpret and manipulate the binary objects.

ELF Data Format

This section describes the ELF data format. If you are using an application that generates data in a different format, you can use the information in this section to convert the data to the ELF format before transmitting the data over a network if desired. If your application is ELF-based and generates data in the form of ELF arrays, there is no need to convert the data when transmitting over a network; it can be transmitted as an array using the macros described in the following sections.

The ELF data format is composed of long words. A datum is composed of long integers (32-bit) representing information as follows:

message type
data type
data size

data

The message type long integer should always be 0.

The data type long integer indicates the type of the datum. The data type can be one of the following numbers:

1	Number
2	String
3	Binary
4	Array
5	Null datum

The data size long integer indicates the size of a datum if the datum is a string, binary object, or an array. If the datum is null or a number, this long integer is omitted. For arrays, data size is the number of elements in the array. For binary objects, data size is the number of bytes in the binary object. For strings, data size is the amount of characters in the string plus 1 for a null terminator. For example, the string "home" has a data size of 5.

The data is the actual data you are sending. For instance, if the datum is the integer 12, data is 12.

For strings, data is not a long word, but rather, a separate byte is used for each character in the string and for the null terminator. For example, the following data format is used to represent the string "home":

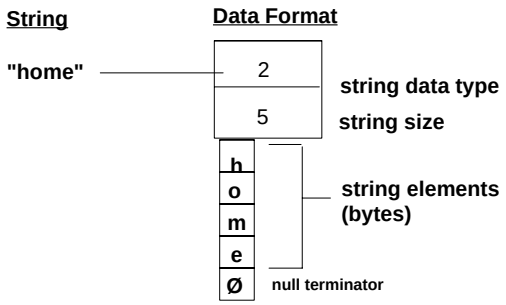


Figure 5-2 String Data Format

For arrays, each array element is represented by data type, data size, and the data. For example, the following data format represents an array containing the strings "fig" and "plum," and a sub-array containing the string "apple" and the integer 12:

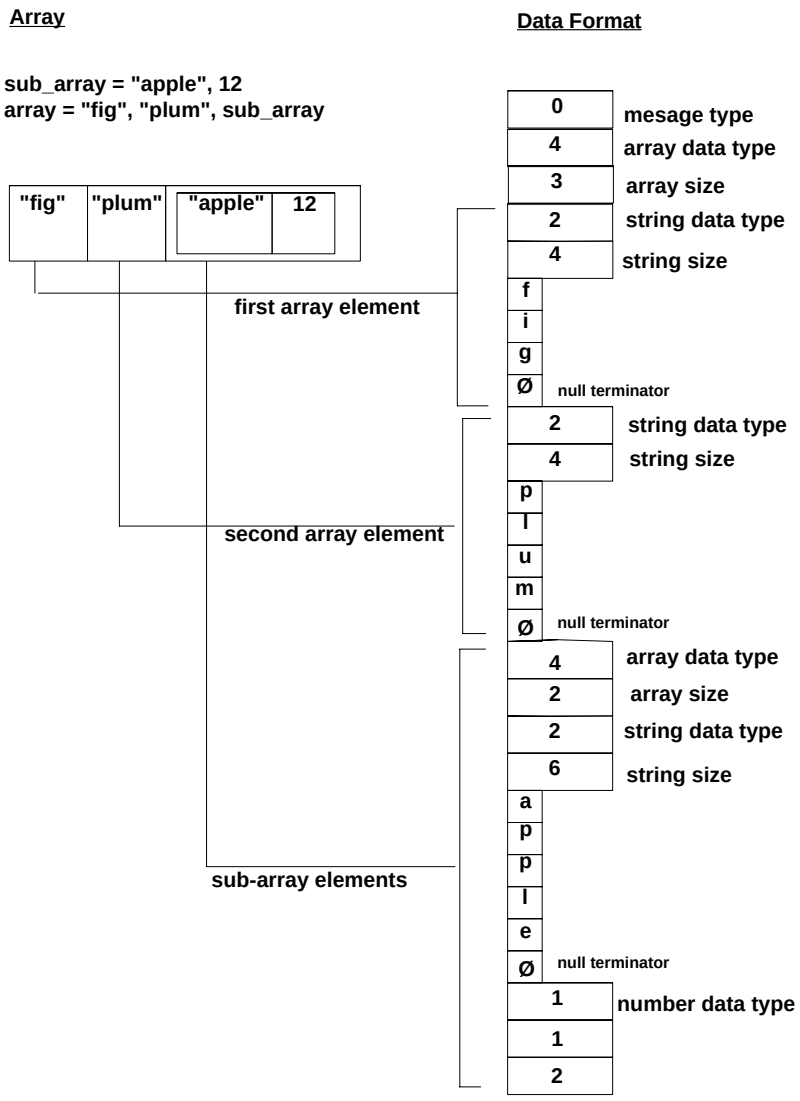


Figure 5-3 Array Data Format

The following sections describe the various ELF macros used for communicating on a TCP/IP network.

Opening Sockets

You use the SOCKET_OPEN_SERVER@ macro to open a socket on the machine you want to designate as a server on the network. The format for the macro is as follows:

```
SOCKET_OPEN_SERVER@(socket_num, timeout)
```

The socket_num is a unique socket number. This number should not be a socket number that is already in use on the system. The etc/services file on the system lists any socket numbers that are used by the system. It is good practice to add your ELF server socket number values to this file so that the numbers are not inadvertently used in the future.

The timeout argument is reserved for future use. You must specify -1 for this argument.

When the SOCKET_OPEN_SERVER@ macro is executed, it is placed in a waiting state until a connection from a client is established. In order for a connection to be made, a socket that references the server's socket_num must be opened on a client machine. When the connection is accepted, SOCKET_OPEN_SERVER@ returns a channel ID. The channel ID is used with other socket macros to identify the channel on which to receive messages. The following are typical statements used to open a socket on a server and assign the socket ID to a variable. In this example, 2310 is used as the socket number.

```
VAR channel
channel = SOCKET_OPEN_SERVER@(2310, -1)
```

You use the SOCKET_OPEN_CLIENT@ macro to open a socket on a client machine. The format for the macro is as follows:

```
SOCKET_OPEN_CLIENT@(socket_num, timeout, hostname)
```

The socket_num is the number of the socket with which you want the client machine to communicate.

The timeout argument is reserved for future use. You must specify -1 for this argument.

The hostname is the name of the machine to which the connection is to be made. For example, if the hostname of the server is "central," you would specify central as the hostname.

SOCKET_OPEN_CLIENT@ returns a channel ID.

Sending and Receiving ELF Arrays over a Network

The SOCKET_WRITE@ macro is used to send an ELF array over a communications channel. The format for the macro is as follows:

```
SOCKET_WRITE@(channel, array)
```

The channel is the channel ID as returned by the SOCKET_OPEN_CLIENT@ macro (or as returned by the

SOCKET_OPEN_SERVER@ macro if you are sending a message from the server).

The array is the ELF array to send.

For example, the following statements open a socket on a client and send an ELF array over the communications channel:

```
VAR elf_array, channel
elf_array = "string1", "string2", "string3"
channel = SOCKET_OPEN_CLIENT@(2310, -1, central)
SOCKET_WRITE@(channel, elf_array)
```

The SOCKET_READ@ macro is used to read an array that has been sent over the network. The format for the macro is:

```
SOCKET_READ@(channel, timeout)
```

The channel is the channel ID as returned by the SOCKET_OPEN_-SERVER@ macro (or as returned by the SOCKET_OPEN_CLIENT@ macro if a client is receiving a message). The timeout argument is reserved for future use; you must specify -1 for this argument.

The SOCKET_READ@ macro returns the ELF array. For example:

```
VAR channel, message
channel = SOCKET_OPEN_SERVER@(2310, -1)
message = SOCKET_READ@(channel, -1)
```

Sending and Receiving Binary Data over a Network

In addition to ELF arrays, you can send binary data over a communications channel. For example, you could use the READ_BINARY_FILE@ and WRITE_BINARY_FILE@ macros to convert a file to a binary object and then send the binary object from one machine on the network to another.

The SOCKET_WRITE_BINARY@ macro is used to send binary data over a communications channel. The format for the macro is as follows:

```
SOCKET_WRITE_BINARY@(channel, object)
```

The channel is the channel ID as returned by the SOCKET_OPEN_CLIENT@ macro (or as returned by the SOCKET_OPEN_SERVER@ macro if you are sending a message from the server).

The object argument is the ELF binary object to send.

The SOCKET_READ_BINARY@ macro is used to read binary data that has been sent over the network. The format for the macro is:

```
SOCKET_READ_BINARY@(channel, timer, maxbytes)
```

The channel is the channel ID as returned by the SOCKET_OPEN_-SERVER@ macro (or as returned by the SOCKET_OPEN_CLIENT@ macro if a client is receiving a message).

The timer is a number indicating the amount of seconds SOCKET_-READ_BINARY@ should read data. Specify -1 if you do not want to

set a time limit for a data read. Typically, a timer is used when you are not sure of the number of bytes that are going to be read.

The maxbytes argument is a number indicating the maximum number of bytes to be read by SOCKET_READ_BINARY@.

SOCKET_READ_BINARY@ returns when either the time limit specified by timer is reached, or the maximum number of bytes, as specified by maxbytes, are read.

SOCKET_READ_BINARY@ returns a binary object the size of which equals the amount of bytes read. If no bytes are read, NULL is returned.

Closing Channels and Sockets

The SOCKET_CLOSE_CHANNEL@ macro closes a channel. The format for the macro is as follows:

```
SOCKET_CLOSE_CHANNEL@(channel)
```

The channel is the channel ID as returned by the SOCKET_OPEN_-CLIENT@ macro or the SOCKET_OPEN_SERVER@ macro.

Typically, the channel is closed on a client after it has sent a message and the channel will be closed on the server after it has received a message.

If you want, you can close the socket on a server using the SOCKET_-CLOSE@ macro. It is not necessary to close sockets on clients. The format for the macro is:

```
SOCKET_CLOSE@(socket_num)
```

The socket_num is the name of the server socket you want to close. Once a socket is closed, any attempts to send messages to the socket will result in an error message.

All sockets that were opened by ELF macros are automatically closed when Applixware is exited.

NOTE: Because socket communication is asynchronous, the socket will remain bound for up to a minute or two after it is closed.

A Socket Communications Example

This section presents an example of TCP/IP communications using ELF macros. In this example, the server macro accepts an ELF array from a client and simply displays that first element of the array in an INFO_MESSAGE@ dialog box. This example is provided with Applixware in the file axdata/eng/Demos/elfnet.am.

The client macro sends a message to the server. If the name of the machine to which to send the message, the socket number of the server, and the message are not specified as arguments to the client macro, PROMPT@ is used to display dialog boxes that prompt for this information.

The server macro is as follows:

```
MACRO ElfNet(passed_socket_num)
    VAR socket_num        ' the socket number
    VAR channel           ' incoming message channel
    VAR hostname' name of this machine
    VAR message           ' string transmitted by client


' Get socket number from either command line or user.
```

```
    IF IS_NULL@(passed_socket_num)
    {
        INFO_MESSAGE@("To run the ElfNet server, you\n" ++
                      "must select an unused\n" ++
                      "socket number.  You should\n" ++
                      "select a number above 1000\n" ++
                      "and not already in use.\n\n" ++
                      "To see what sockets are in use\n" ++
                      "on your machine, review the\n" ++
                      "file 'etc/services'.\n")
        socket_num = PROMPT@("Server socket number",
                             null, 4300) + 0
        IF socket_num = 0   'user must have cancelled
            RETURN
        INFO_MESSAGE@("By the way, you can run this\n" ++
                      "macro from the macro prompt\n" ++
                      "by typing 'elfnet " ++ socket_num
                      ++ " ' ")
    }
    ELSE
        socket_num = passed_socket_num + 0

' Give directions on how to transmit to this socket.

    hostname = SHELL_COMMAND@("hostname")
    hostname = hostname[0]
    IF IS_NULL@(passed_socket_num)
        INFO_MESSAGE@("To send a message to this\n" ++
                      "socket, from either this\n" ++
                      "machine or another, run the\n" ++
                      "ElfMsg macro, specifying\n" ++
                      "Hostname "++hostname++ "\n" ++
                      "and socket " ++ socket_num ++
                      "\n" ++ "Do this after pressing" ++
                      "the OK/Proceed button here.")
' Open the socket as a server, wait for a connect.

' Once the socket is bound to, SOCKET_OPEN_SERVER@ just
' waits for the next connection.
```

```
 WHILE TRUE
    channel = SOCKET_OPEN_SERVER@(socket_num, -1)
    message = SOCKET_READ@(channel, -1)
    SOCKET_CLOSE_CHANNEL@(channel)
    INFO_MESSAGE@(message[0])
 WEND
ENDMACRO
```

The client macro is as follows:

```
MACRO ElfMsg(passed_hostname, passed_socket_num, passed_msg)
    VAR hostname  ' computer server is running on
    VAR socket_num          ' TCP/IP socket ID
    VAR msg         ' text of message to send
    VAR channel             ' communications channel opened to server
    VAR arr      ' array holding message text as first argument

    IF IS_NULL@(passed_hostname)
        hostname = PROMPT@("Hostname to send to")
    ELSE
        hostname = passed_hostname

    IF hostname = " "          'user must have cancelled
        RETURN

    IF IS_NULL@(passed_socket_num)
        socket_num = PROMPT@("Socket to send to")
    ELSE
        socket_num = passed_socket_num

    IF socket_num = " "       'user must have cancelled
        RETURN

    IF IS_NULL@(passed_msg)
        msg = PROMPT@("Message")
    ELSE
        msg = passed_msg

    IF msg = " "       'user must have cancelled
```

```
        RETURN
    arr[0] = msg

' Open channel, send message, close channel
    channel = SOCKET_OPEN_CLIENT@(socket_num, -1,
                                  hostname)
    SOCKET_WRITE@(channel, arr)
    SOCKET_CLOSE_CHANNEL@(channel)
ENDMACRO
```

A C Function Summary

This appendix contains a short description of Applixware C language functions. The macros are divided into the following categories:

- Extracting Data from ELF
- Creating ELF Data
- Information and Testing Functions
- axnet Functions
- Advanced Functions

Extracting Data from ELF

void **AxDecomposeArray**(elfData arrayData, char *str, args,...)
Decomposes an elfData array into its parts.

elfData **AxArrayFromArray**(elfData arrayData, int index)
Extracts an ELF array from an elfData format.

int **AxBinaryFromArray**(elfData arrayData, int index, char ** binP)
Extracts binary data from an elfData array. It returns the size of the data. binP points to the location of this data.

int **AxBinaryFromDataPtr**(elfData arrayData, char **binP)
Returns the size of the binary data. binP points to this data.

int **AxBoolFromArray**(elfData arrayData, int index)
Returns a Boolean value or integer from an elfData array.

int **AxBoolFromDataPtr**(elfData arrayData)
Returns a Boolean value or integer from an elfData format.

elfData **AxCopyData**(elfData data)
Copies elfData and returns a pointer to this copy.

double **AxFloatFromArray**(elfData arrayData, int index)
Returns a C floating point number from an elfData array.

double **AxFloatFromDataPtr**(elfData arrayData)
Returns a floating point number from an elfData element.

int **AxIntFromArray**(elfData arrayData, int index)
Returns a C integer from an elfData array.

int **AxIntFromDataPtr**(elfData arrayData)
Returns a C integer from an elfData array.

char * **AxStrFromArray**(elfData arrayData, int index)
Extracts a string from an elfData array and returns a pointer to it.

char * **AxStrFromDataPtr**(elfData arrayData)
Extracts a string from an elfData element and returns a pointer to it. This function differs from the next in that if the string was a number and was converted, space for the string is created. Thus, another conversion operation will not overwrite this string.

char ***AxStrPtrFromArray**(elfdata arrayData, int index)
Extracts a string from an ELF data array and returns a pointer to it. The memory being pointed to is unstable; that is, it can be reused by other Applixware functions.

char * **AxStrPtrFromDataPtr**(elfData arrayData)
Extracts a string from an elfData element and returns a pointer to it.

Creating ELF Data

elfData **AxBuildArray**(char *str, args,...)
Builds an elfData array from C language arguments.

elfData **AxAddArrayToArray**(elfData ap, int index, elfData ptr)
Adds an elfData array to an existing elfData array, returning a pointer to this array. This function allows you to create a multidimensional array.

elfData **AxAddBoolToArray**(elfData ap, int index, int num)
Adds a C Boolean value to an existing elfData array, returning a pointer to this array.

elfData **AxAddDataToArray**(elfData ap, int index, elfData ptr)
Adds an arbitrary elfData to an existing elfData array, returning a pointer to this array. This function is identical to AxAddArrayToArray.

elfData **AxAddFloatToArray**(elfData ap, int index, double num)
Adds a C floating point number to an existing elfData array, returning a pointer to this array.

elfData **AxAddIntToArray**(elfData ap, int index, int num)
Adds an integer to an existing elfData array, returning a pointer to this array.

elfData **AxAddStrToArray**(elfData ap, int index, char *str)
Adds a C string to an existing elfData array, returning a pointer to this array.

elfData **AxMakeArray**(int numOfElements)
Creates an elfData array, returning a pointer to this array.

elfData **AxMakeBinaryData**(int numOfElements, char *dataBytes)
Creates an elfData array that will contain the number of elements specified.

elfData **AxMakeFloatData**(double num)
Creates an elfData floating point number based upon a C floating point number.

elfData **AxMakeIntData**(int num)
Creates an elfData integer based upon a C integer.

elfData **AxMakeStrData**(int len, char *str)
Creates an elfData string based upon a C string.

Information and Testing Functions

int **AxArraySize**(elfData arrayData)
Returns the array size of an elfData array.

elfData **AxArrayElement**(elfData arrayData, int index)
Returns an element from within an array, returning this information in elfDataForm. This is the same as AxArrayFromArray if the element is not an array.

char * **AxBinaryBytesPtr**(elfData arg)
Returns the address of a binary block.

int **AxBinarySize**(elfdata arrayData)
Returns the size of a binary array.

void **AxError**(int code, char *str, char *object)
Throws an error back to ELF.

int **AxIsArray**(elfData data)
Returns TRUE if data is an array; otherwise, FALSE is returned.

int **AxIsBinary**(elfData data)
Returns TRUE if data is binary; otherwise, FALSE is returned.

int **AxIsFloat**(elfData data)
Returns TRUE if data is floating point; otherwise, FALSE is returned.

int **AxIsInt**(elfData data)
Returns TRUE if data is an integer; otherwise, FALSE is returned.

int **AxIsNumber**(elfData data)
Returns TRUE if data is an floating point or integer number; otherwise, FALSE is returned.

int **AxIsString**(elfData data)
Returns TRUE if data is a string; otherwise, FALSE is returned.

axnet Functions

void **AxClientDisConnect**(int channel)
Terminates the connection to a server. channel can represent a real or a non-block channel. (Non-blocking channels have values greater than 1000000-one million.)

elfData **AxNetElfChannelCall**(int channel, char *funcName, elfData Args, int *code, char **str, char **obj)
Calls ELF code from a C client. If an error occurs, code is set; str is set to the error message and obj is the object of the error. channel is the value returned from AxNetElfServiceConnect. funcName must be a name that is registered with axnet.

int **AxNetElfServiceConnect**(char *hostName, char *fullServiceName, int *code, char **str, char **obj)
Allows C clients to talk to ELF services. This function returns a communication channel that can be used by AxNetElfChannelCall. If an error occurs, -1 is returned and code is set. str is set to the error message and obj is the object of the error.

If you are executing on a local machine, hostName can be NULL. fullServiceName is in the form username:servicename.

Advanced Functions

NOTE: Applix discourages the use of the functions listed in this section. The only circumstance in which these functions should be used is when you are writing a very specialized RPC application. For example, your program needs to run as main. Appendices C and D to this book contain examples of how you may use these functions. For example, the AxSockTop function is used to call a separately executing AxDispatch function. The 200 or so lines of code surrounding the call to AxSockTop make up a minimum context for using this function. You will have to create the context that is suitable for your use.

elfData * **AxArrayDataAddr**(elfData arrayData)
: Returns the address of the first datum within arrayData.

void **AxFree**(char *p)
: Deallocates memory.

void **AxFreeData**(elfData data)
: Frees the memory allocated to an elfData item. This is a recursive ‘‘free’’ and can be dangerous if all pointers are not properly de-allocated. (When memory is freed recursively, the elements pointed to by a freed element are also freed. That is, if the passed ELF data is an array, the elements of the array are also freed.)

char * **AxMalloc**(int size)
Allocates size bytes of memory.

void **AxMallocAborter**(jmp_buf abortbuf)
Recovers from a memory allocation error.

int **AxMemWrite**(elfData data, int makeRoom, char **buf)
Writes elfData to a single contiguous memlory buffer, which it creates. It returns an integer and assigns buf with a pointer address. After processing the returned buffer, the caller must free it.

int **AxOpenServer**(char *sockName, forkIt)
Creates a socket and binds the process to it.

elfData **AxRPCRead**(char *bufP)
Reads a memory buffer that contains elfData.

int **AxSockTop**(int readFileDescrip, int writeFileDescrip)
Gets a messages and sends it to AxDispatch. The following values are returned:

ELFSVR_OK	0	success
ELFSVR_ERROR	-1	error
ELFSVR_NOCHANNEL	1	obsolete
ELFSRVR_NOCLIENT	2	no client

int **AxSSSetMathErr**(int (*funcPtr)())
Defines an error handler to the used with shared library math functions. This should be the first functio called within their ELF wrapper to their C function.

B ELF Communication Macro Summary

AXNET_CLOSE_CHANNEL@(channel)
Closes a channel.

AXNET_OPEN_CHANNEL@(hostname, service)
Opens a channel. Returns the channel number.

AXNET_READ@(channel)
Reads a buffer of information. Returns data as an ELF array.

AXNET_RPC@(hostname, service, cmdCode[, arg1[, arg2[,...,arg10]]]
Directs a function to be executed within a service. Optionally returns a value.

AXNET_RPC_CHANNEL@(channel,cmdCode[, arg1[,arg2[,...,arg10]]]
Directs a function to be executed. Optionally returns a value.

AXNET_SERVICES_LIST@(hostName)
Returns a list of services as an ELF array.

AXNET_SERVICE_REGISTER@(serviceName, macroNames[, isPrivateFlag])
Adds a service to axnet.

AXNET_SERVICE_UNREGISTER@(serviceName)
Removes information about a service.

AXNET_START_SERVICE@(hostName, service, pathname[, argumentList])
Starts a server program.

AXNET_START_SLAVED_PROCESS@(hostName, pathname, argArray)
Starts a private server. The channel is returned.

RPC_CALL@(progName, cmd, data[, socketname[, host]])
Invokes an RPC function. This macro will open and close a channel. If the server is shared by many users, you must use this macro so that your task does not monopolize the channel.

RPC_CHANNEL_CALL@(channel, cmd, data)
Invokes an RPC function using an already open channel.

RPC_CHANNEL_MASTER@(channel, taskID)
Sets the channel's ''existence'' state when the task stops executing.

RPC_CHANNEL_USE_HOURGLASS@(channel, hourglassFlag)
Displays an hourglass during ELF and RPC calls..

RPC_CONNECT@(progName[, socketName[, host]])
Connects to a channel or creates the initial channel connection. The channel number is returned. This returned value is used in most other RPC macros.

RPC_DISCONNECT@(progName)
Closes a channel.

RPC_START_SERVER@(progName[, socketName[, host]])
Starts a server process and returns a channel number. A new server is always created, even though the server may already be running or attached to a channel.

SOCKET_CLOSE@(*num*)
Closes the specified TCP / IP socket on a server.

SOCKET_CLOSE_CHANNEL@(*channel*)
Closes the specified TCP / IP network channel.

SOCKET_OPEN_CLIENT@(*nu, timeout, hostname*)
Opens a socket on a TCP / IP network.

SOCKET_OPEN_SERVER@(*channelName, timeout[, phase]*)
Opens a socket on a TCP / IP network. Returns *channelID*.

SOCKET_READ@(*channel, timeout*)
Reads the ELF array sent over a TCP / IP network channel. Returns *array*.

SOCKET_READ_BINARY@(*channel, timer, maxbytes*)
Receives an ELF binary object over a TCP / IP network channel. Returns *data*.

SOCKET_RPC_READ@(uid, timer)
Reads a message from a socket into a returned ELF array. timer represents the number of seconds. Special values are -2 (read whole buffer as a message) and -1 (read forever).

SOCKET_RPC_WRITE@(uid, code, data)
Writes to a socket using RPC style packing.

SOCKET_WRITE@(*channel, array*)
Sends an ELF array over the specified TCP / IP network channel.

SOCKET_WRITE_BINARY@(*channel, object*)
Sends an ELF binary object over a TCP / IP network channel.

C Creating a main() Procedure

The procedures described in the previous chapters have assumed that you do not need to run your own main procedure. Instead, you use a special main contained within elfapi.a. In most cases, this is a reasonable assumption.

In some circumstances, you may need to have your own main. For example, you are creating an application using C++ instead of C. The procedure for this is straight--forward. However, it requires that you create a considerable amount of code. In addition, you must be familiar with and able to program at the socket level.

Programming at the socket-level can be quite demanding. If you are not familiar with socket level programming, you are strongly urged to consult one of the standard texts that describe how sockets are used and programmed. Very subtle errors can instantly cause major program failures. Also, because you are dealing with more than one process, debugging is extremely difficult.

A Sample Program

This section contains an example program that demonstrates how you would bind an AxDispatch function into a program that must use its own main entry point. That is, this programs starts a server that runs independently of Applixware. This program will listen for data on a known socket.

The C program that will be shown does very little. It reads stdin and then writes this information back to stdin. It will also responds to ELF calls.

The AxDispatch function supports two requests:

- Request 1 tells it to write "Hello world" to stdout.
- Request 2 tells it to return an environment variable to the ELF macro. This environment variable is retrieved using a the getenv function.

The ELF Macro

This macro connects to our rpcside server three times. Each time its sends the "hello" message and a request for the value of an environment variable.

```
macro rpcside
    var channel
                    ' left nameless... connecting to
                    ' exiting server
    var server
    var hostname    ' left empty... current machine
                    ' matches sockname in server...
                    ' can be pathname
    var socket
```

```
    var names, value, i

    names = "USER", "HOME", "SHELL"
    for i = 0 to array_size@(names)-1
        socket = 6162
        channel = RPC_CONNECT@(server, socket, hostname)
        RPC_CHANNEL_CALL@(channel,1)
        value = RPC_CHANNEL_CALL@(channel,2,names[i])
        INFO_MESSAGE@(names[i] ++ ": " ++ value)
        RPC_DISCONNECT@(channel)
    next i
endmacro
```

The C Program

This section contains two routines: main and AxDispatch. Like all AxDispatch routines, it is disconnected; that is, examining the code does not reveal the point at which it is invoked. In Applixware RPC programming, this function is invoked by the AxSockTop function.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <netinet/in.h>
#if !SCOUNIX
#include <sys/un.h>
#endif
#include <netdb.h>
#include <errno.h>
#include <signal.h>

#define TRUE 1
#define FALSE 0
#include <elfapi.h>


extern char       *sys_errlist[];  /* Unix errro strings */
```

```
extern void       AxErrnoError();
void              SocketMain();
void              (*AxDisconnectHandler) ();

/*****************************************************************
 * function: main - Top of server
*****************************************************************/
void
main(argc, argv, env)
     int          argc;
     char         *argv[];
     char          *env[];
{
     jmp_buf      dieGraceFully;       /* allocator error catcher */
     int          ix;
     int          isUnix;
     int          sockfd, ElfFd;
     char         sockname[300];
     int          clilen;
     struct sockaddr_in in_cli_addr;
     struct sockaddr_un un_cli_addr;
     int          code;
     static fd_set   read_fds, wtfds;
     int          events, i;
     char         onechar;

/* Choose a socket to listen on... can be either a pathname
(named pipe) or a number (tcp) */

                                  /* my favorite socket number */
     sprintf(sockname, "%d", 6162);

/* set and catch out of memory condition */

     if (setjmp(dieGracefully)) {
          printf("Out of memory\n");
          exit(1);
     }
     AxMallocAborter(dieGracefully);
```

```
/* Open listener channel.  This gets the connect/accept messages
from ELF */

    sockfd = AxOpenServer(sockname, FALSE);
                    /* isUnix means named pipe; else TCPIP */
    isUnix = (sockname[0] == '/');
    ElfFd = sockfd;                     /* get a connect on ElfFd */

/* Program mainloop.
 *      This program simply reads and echos stdin.
 *      It also responds to ELF calls.
 *      Thus, the "select" listens on stdin & ElfFd.
*/

    while (TRUE) {

        /* clear the descriptors, and then set the ones we want */
        FD_ZERO(&read_fds);
        FD_ZERO(&wtfds);
        FD_SET(ElfFd, &read_fds);
        FD_SET(fileno(stdin), &read_fds);

        /* wait for event on either stdin or ElfFd */
        events = select(FD_SETSIZE, &read_fds, &wtfds, NULL,
                NULL);

        if (events < 0)                         /* error */
        {
            if (errno == EINTR)
                continue;
            fprintf(stderr, "error? (events = %d, errno = %d))\n",
                events, errno);
            continue;
        }

        if (0 == events)                        /* timed out */
        {
            continue;
```

```
}
                                    /* Got Elf message! */
if (FD_ISSET(ElfFd, &read_fds))
{
     if (ElfFd == sockfd)        /* got connection request */
     {
          if (isUnix)
          {
               clilen = sizeof(un_cli_addr);
               ElfFd = accept(sockfd, &un_cli_addr, &clilen);
          }
          else
          {
               clilen = sizeof(in_cli_addr);
               ElfFd = accept(sockfd, &in_cli_addr, &clilen);
          }

          if (ElfFd == -1)
          {                         /* ignore the interrupt */
               if (errno == EINTR)
               {
                    ElfFd = sockfd;
                    continue;
               }
               printf("Accept failed(%d).\n", errno);
               exit(1);
          }
          continue;
     }
     else
     {                              /* got Elf->C RPC call */
                                    /* calls AxDispatch! */
          code = AxSockTop(ElfFd);
          if (code != 0)
          {
               if (code == -1)
                    fprintf("error %d\n", errno);
               shutdown(ElfFd, 2);
               close(ElfFd);
```

```
                    ElfFd = sockfd;     /* wait for another client */
                }
            }
        }
                                    /* Got typing & RET key */
        if (FD_ISSET(fileno(stdin), &read_fds))
        {
            if (1 == read(fileno(stdin), &onechar, 1))
                write(fileno(stdout), &onechar, 1);
            else
                break;          /* line was ^D */
        }
    }

    shutdown(ElfFd, 2);
    close(ElfFd);
    if (ElfFd != sockfd)
    {
        shutdown(sockfd, 2);
        close(sockfd);
    }
    if (isUnix)
        unlink(sockname);

    printf("Goodbye!\n");
    exit(0);
}


/*****************************************************************
 * function: AxDispatch - Call customer specific C code
 *****************************************************************/
elfData
AxDispatch(funcId, funcData, quitNow)
    int         funcId;
    elfData     funcData;
    int32       *quitNow;
{
                    /* return data, presumed no-return */
```

```
    elfData        rData = NULL;
    char          *name, *value;

    switch (funcId)
    {
    case 1:
        printf("Hello world!\n");
        break;

    case 2:                         /* string arg, string return datum */
                                     /* get var name */
        name = AxStrFromDataPtr(funcData);
                                     /* call non-Applix */
        value = getenv(name);
                                    /* package its return data */
        rData = AxMakeStrData(-1, value);
        break;
    }
    return (rData);
}
```

Building *main*

On a SUN OS compiler, you would create this program as follows:

```
cc -o axnextcli \
        -I/applix/axdata/elf \
        rpcside.c \
        /applix/axdata/elf/elfapi.a
```

This command compiles the file named rpcside.c into an output file name axnetcli. This program uses functions contained within the elfapi.a archive library.

Consult your compiler documentation for instructions on how to create this program using other operating systems.

D Calling Applixware from C Programs

The program presented in the appendix demonstrates registering a service with axnet that can be invoked by a C program. While the program in Appendix C assumed that Applixware was a client that utilitized the services of a server program, this program reverses that relationship. That is, Applixware will act as a server providing a service to a main program.

A Sample Program

This example program is general enough that it can call any ELF macro that is registered with axnet.

The ELF Macros

The folowing two ELF macros register and unregister a service with axnet. The AXNET_HELP_SERVER@ macro creates a server named help that responds to messages set to it. Note that the real service name is <username>:help. Two ELF macros are associated with this server: WP_HELP_HYPERLINK@ and HELP_SEARCH_DLG@. After the macros are registered, they can be invoked by another process.

```
macro axnet_help_server
    var info

    info = "wp_help_hyperlink@", "help_search_dlg@"
    AXNET_SERVICE_REGISTER@("help", info)
endmacro

macro axnet_drop_help_server
    AXNET_SERVICE_UNREGISTER@("help")
endmacro
```

The C Program

The following program invokes the help service registered with axnet.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <fcntl.h>
```

```
#include <netinet/in.h>
#include <sys/un.h>
#include <netdb.h>
#include <errno.h>
#include <assert.h>
#include "elfapi.h"

extern elfData AxNetElfChannelCall();

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif

main(argc,argv)
int argc;
char *argv[];
{
     char *hostName, *serviceName, *funcName;
     char fullServiceName[300];
     int code;
     char *str, *obj;
     int channel, ix;
     elfData response, Args;

/* Open connection to Elf Server. This is done ONCE. */

     hostName = argv[1];
     serviceName = argv[2];
     funcName = argv[3];
     sprintf(fullServiceName, "%s:%s", getenv("USER"),
                    serviceName);

     channel = AxNetElfServiceConnect(hostName,
                    fullServiceName, &code, &str, &obj);
     if ( code != 0 )
          AbortOut(code,str,obj);

/* Call ELF code!!!!  This can be done MANY times. */
```

```
    Args = AxMakeArray(0);                    /* create args array */
                                    /* args cmdline args to array */
    for ( ix = 4; ix < argc; ix++ )
        Args = AxAddStrToArray(Args,ix-4,argv[ix]);

    /* now call the function */
    response = AxNetElfChannelCall(channel, funcName, Args,
                &code, &str, &obj);
    if ( code != 0 )
        AbortOut(code,str,obj);

    AxClientDisConnect(channel);
    exit(0);
}

AbortOut(code,str,obj)
int code;
char *str;
char *obj;
{
    if ( obj[0] )
        fprintf(stderr,"%s: %s (error %d)\n",obj,str,code);
    else
        fprintf(stderr,"%s (error %d)\n",str,code);
    exit(code);
}

                        /* needed just to satisfy elfapi.a's linkage */
elfData AxDispatch(){}
```

Compiling and Using

If the C program is stored in a file named axnetcli.c and you are compiling it on a computer named fastbox, your command line on most computers is as follows:

```
fastbox% cc -o axnextcli \
            -I/applix/axdata/elf \
            axnetcli.c \
            /applix/axdata/elfapi.a
```

Before invoking the compiled program, you must execut the axnet_-help_server macro described earlier in this appendix. It can be invoked using the F8 key, from within a login file, or from the Applix command line. The command usage is as follows:

```
axnetcli <hostname> <servicename> <macroname> <helpTopic>
```

For example, if your machine's name is fastbox, you would type:

```
fastbox% axnetcli fastbox help wp_help_hyperlink@
                INFO_MESSAGE@
```

The axnetcli program runs any ELF macro that has been registered with any service. For example, assume you registered a function named "hello_world", which displays a message box with the text "Hello World" in it, as follows:

```
macro axnet_hello_server
    var info

    info[0] = "hello_world"
    AXNET_SERVICE_REGISTER@("hello", info)
endmacro

macro hello_world
    info_message@("Hello World")
endmacro
```

To invoke this macro from axnetcli, you would type:

```
fastbox% axnetcli fastbox hello hello_world
```

Index

A

B

C

D

E

F

G

H

S

T

U

W

Z