UNC-CH Force-Feedback Library,
Revision C

William R. Mark
Scott C. Randolph
Mark Finch
James M. Van Verth

University of North Carolina at Chapel Hill
Department of Computer Science

Technical Report #TR96-012

January 30, 1996

Note: As this library continues to be developed, updated copies of the documentation will be available from ftp://ftp.cs.unc.edu/pub/packages/GRIP/armlib

(C) 1996, University of North Carolina at Chapel Hill
# Table of Contents

Table of Contents .................................................................................................................. 1  
Important Information ............................................................................................................ 1  
  IMPORTANT SAFETY AND LEGAL INFORMATION ..................................................... 1  
  Contact and Update Information ......................................................................................... 1  
  Copyright ............................................................................................................................. 1  
I. Acknowledgements ............................................................................................................. 2  
II. Introduction ....................................................................................................................... 3  
  Force-Feedback Devices .................................................................................................... 3  
  Force-Feedback Library Goals ......................................................................................... 3  
  Arm Library Overview Diagram ....................................................................................... 5  
III. Using an Arm Application ............................................................................................... 7  
IV. Coordinate System and Unit Conventions ..................................................................... 8  
  Coordinate System Orientation ....................................................................................... 8  
  Coordinate System Origin ............................................................................................... 8  
  Position Units ................................................................................................................. 9  
  Angular Units .................................................................................................................. 9  
  Force Units ...................................................................................................................... 9  
  Torque Units ................................................................................................................... 9  
  Time Units ....................................................................................................................... 9  
V. Arm Library Concepts ....................................................................................................... 10  
  Synchronous Operation .................................................................................................. 10  
  Asynchronous Operation .............................................................................................. 10  
  Event-Driven Operation ............................................................................................... 10  
  Time Measurement ....................................................................................................... 11  
  Planar Approximation to Hard Surfaces ....................................................................... 12  
VI. Application Programmer Interface ................................................................................. 15  
  Overview ....................................................................................................................... 15  
  Basic Routines ............................................................................................................. 16  
  Planar Surface Approximation Routines ...................................................................... 27  
  User Definable Extensions ............................................................................................ 32  
VII. Compiling and Linking .................................................................................................. 33  
  At UNC ........................................................................................................................... 33  
  Not at UNC ..................................................................................................................... 33  
VIII. Example Programs ....................................................................................................... 34  
  Program Using a Single Arm ......................................................................................... 34  
  Program Using Two Arms ............................................................................................ 36  
  Program Using Dials/Buttons ....................................................................................... 39  
IX. Backward Compatibility Library ..................................................................................... 41  
  Backward Compatible Routines: .................................................................................. 41  
  Backward Compatible Global Variables: ...................................................................... 41  
  Backward Compatible Include File: ............................................................................. 41  
X. ArmLib Internals ............................................................................................................... 42  
  Diagnostic Routines ..................................................................................................... 42  
  Source Files .................................................................................................................. 42  
  Installing ArmLib .......................................................................................................... 45  
  Recompiling Test Applications .................................................................................... 47  
  ArmLib Organization ................................................................................................... 47  
XI. Notes, Bugs, etc. ............................................................................................................. 51  
Appendix A — Phantom Joint Axes ..................................................................................... 52  
Appendix B — Additional Phantom Server Information .................................................. 55
Motor Overload Protection .............................................................. 55
Maximum Force Clipping ............................................................. 55
**Important Information**

**IMPORTANT SAFETY AND LEGAL INFORMATION**

Force-feedback devices, even small ones, can be dangerous. As a user of this software you accept FULL responsibility for assuring the safety of users. Measures you are responsible for include, but are not limited to:

1. Appropriate testing of hardware and software.
2. Use of appropriate hardware safety measures, such as kill switches and circuitry to detect dangerous conditions.
3. Assuring that the software contains appropriate safety checks for your application and hardware.
4. Maintenance of appropriate training and operating procedures at your site.

The University of North Carolina and the authors of this product are not responsible for any injuries, damages, etc. caused in part or in whole by the use of this software, or by any errors or omissions in its documentation. The software and documentation are provided AS IS, with no warranty, express or implied, as to their fitness for anything. This software is a research tool, not a commercial product, and as such should be expected to contain some bugs, oversights, etc. Please be careful!

**Contact and Update Information.**

You may send comments, questions, or reports of problems to:

- Email: armlib@cs.unc.edu
- US Mail: Project GRIP
- UNC Computer Science
- CB #3175
- Chapel Hill, NC 27599
- Phone: (919) 962-1838 (GRIP Project Director)

We would also appreciate getting email from you indicating that you're using our software so we can keep track of the number of users.

New releases of the software may occasionally be made available by anonymous ftp from ftp.cs.unc.edu, /pub/packages/GRIP/armlib. Make sure to carefully examine and test new versions of the software to check for potential safety problems that may have appeared.

**Copyright**

This software is (c) 1995, University of North Carolina at Chapel Hill. It may be used and/or modified without further permission, provided that all copyright, legal, and authorship notices are preserved and that it is not redistributed without prior written permission in any commercial manner (eg: for money, or in conjunction with a product for which money is charged).
I. Acknowledgements

The force-feedback arm library described in this document is the second such library written at UNC. Its initial architecture was sketched by Bill Mark, and the architecture was updated and the library written jointly by Bill Mark and Scott Randolph. Further enhancements were made by Mark Finch and Jim Van Verth. We took many ideas from an earlier library, designed to control the Argonne Remote Manipulator (ARM), which was written by Russell Taylor. Russell Taylor's library was in turn based in part on routines which were taken from Ouh-young Ming's docker program.

Dr. William Wright and Dr. Fred Brooks have provided guidance, support, and patience throughout this project.

John Hughes has done much of the work required to keep our force-feedback hardware running.

Funding for this work was provided by the National Institutes of Health (under the Research Resource program in Molecular Graphics), the Advanced Research Projects Agency, and the National Science Foundation:

NIH National Center for Research Resources, RR-02170

ARPA DAEA 18-90-C-0044

NSF/ARPA Science and Technology Center for Computer Graphics and Scientific Visualization, ASC-8920219.
II. Introduction

WARNING! UNLIKE OTHER I/O DEVICES, FORCE-FEEDBACK HARDWARE HAS THE ABILITY TO DESTROY ITSELF, ITS ENVIRONMENT, AND ITS USER. FOR THIS REASON, HANDLE FORCE-FEEDBACK DEVICES, AND THEIR CONTROLLING SOFTWARE, WITH CARE.

Force-Feedback Devices

A Force-feedback device is most simply described as a user-interface device which relates positions (typically 6 or more degrees of freedom), to forces (again, typically 6 or more DOF). A software application program controls the relationship between position and force.

Such a device can be used to create a wide variety of virtual environments. The most well-developed examples at UNC are the docker program, which presents to the user the forces encountered by a drug molecule as it interacts with a protein, and the atomic force microscope program, which allows the user to interact with atomic-scale surfaces.

Force-Feedback Library Goals

This force-feedback library was developed with a number of goals in mind:

Device Independence

The armlib API (Application Programmer Interface) is designed so that with a minimum of effort, most applications will run unchanged on any supported force-feedback device. The choice of devices is made by setting a UNIX shell environment variable. The library currently supports the following devices:

1) Sensable devices Inc, PHANToM
   A three DOF input, three DOF output device.
   Referred to throughout this documentation as the “Phantom”

2) Sarcos Research Corporation Dextrous Master Arm
   A ten DOF input, ten DOF output device.
   Our software makes it appear to be a six DOF input, six DOF output device.
   Referred to throughout this documentation as the “Sarcos arm”

3) Argonne Remote Manipulator (ARM), located at UNC.
   A six DOF input, six DOF output device, originally constructed at Argonne National Labs and modified for virtual environment applications by UNC.
   Referred to throughout this documentation as the “Argonne arm”, or just “ARM”

The internal design of the armlib compartmentalizes device-dependent code and parameters in a small, well-defined set of routines and arrays. Thus, the addition of new arm types is relatively easy. Note that the FTPable software release includes only the Phantom driver as the other two devices are unlikely to be encountered outside of UNC. The other two drivers are available upon request.
The library is designed to support a particular class of force-feedback devices, and it would require extra work to use with other types of devices. The class of devices for which it was designed is defined as follows:

1) Single end effector: Support for multiple fingers attached to the end of the device, as for full support the Sarcos arm, could be added with a reasonable amount of effort. We just haven't done it yet.

2) "Linear" chain of links. The library assumes internally that the device's configuration can be represented in Denavit-Hartenberg form. "Parallel" links would require some changes to the library.

Additionally, the library makes some assumptions about the information that the application needs. Specifically, the library only provides information about end effector position and orientation, and not about the position and orientation of every link in the device. Thus, for a device with greater than six degrees of freedom some information is effectively discarded. Finally, the library is designed to relate position to force using the "sense position, apply force" paradigm.

**Operate Multiple Devices Simultaneously**

The armlib allows an application to use multiple force-feedback devices simultaneously. This ability is useful for multi-user or telepresence applications, or to provide a user with a force-feedback device for each hand.

**Distributed Operation**

The library is designed using a client-server model. The computer which controls the force-feedback hardware runs the server side of the library. The application (including the client side of the library) may run on any computer, communicating via a network with the server. The client and server may run on the same computer, which may be desirable for some applications.

The routines used to communicate between the client and server are well-defined and compartmentalized. The current communications routines are based on UDP and TCP communications, but a set of routines could easily be added to communicate using shared memory or any other communications medium.

**High Performance**

A number of features are present in the armlib to enhance force-feedback application performance, especially in the context of compute intensive applications and network latency. These features include asynchronous operation to reduce application wait time, support for prediction of future position/orientation, and the efficient transmission of data between client and server.

**Simplicity of Use**

Although the armlib API supports a wide variety of options, the library is designed so that very few parameters need to be specified for most API calls. Default parameters have been carefully chosen which are appropriate for most applications.
Compatibility with Legacy Applications (UNC only)

A compatibility library is provided which presents UNC’s legacy applications with the previous armlib's API. These applications need only be relinked with the compatibility library and the new armlib to function correctly. They will immediately reap the benefits of support for multiple devices provided by the new armlib.

Arm Library Overview Diagram

The diagram on the following page gives an overview of the entire UNC-CH Force-feedback System.
The UNC-CH Force Feedback System is organized as a client-server system. A force-feedback application may run on any workstation on the LAN, taking advantage of any available force-feedback device. The system supports multi-user applications by allowing an application to simultaneously use an arbitrary number of force-feedback devices.
III. Using an Arm Application

Applications which use the arm library have the opportunity to specify which arm device they would like to use. Most applications, however, do not specify a specific device, but instead leave the choice open. In this case, the user may select which arm they wish to use by setting the following shell environment variables:

\[
\text{ARM}x = \text{The arm device to be used (set to Phantom, Sarcos, or ARM)}
\]
\[
\text{ARMAD}x = \text{The button/knob A/D device to be used}
\]

[May not be relevant at sites other than UNC]

Since an application may open more than one arm device simultaneously, the "x" in the above environment variable names indicates the open statement to which the information applies. See the discussion of the “opennum” parameter to ArmLib_Open().

For typical single-arm applications, the environment variables which may be set are ARM0 and ARMAD0.

If these variables are left unset, the armlib currently defaults to the use of the Phantom arm and the SGI button/dial box.
IV. Coordinate System and Unit Conventions

The coordinate system and units used by the arm library are consistent across all supported arm types.

Coordinate System Orientation

The orientation of the arm library's base frame is as follows:
- Z points "up", towards the ceiling.
- Y points to the user's right. (user in standard position)
- X points to the user's back.. (user in standard position)

View from above

The old arm library had a different convention, which is given here to aid in converting applications at UNC:
- Z pointed "down".
- Y pointed to the user's back.
- X pointed to the user's right.

View from above

Both the old and new coordinate systems are different from the system used in UNC’s vlib. vlib sets the coordinates relative to compass directions, while armlib sets the coordinates relative to the user's "standard" position. The choice for the arm library was made to improve the device-independence of applications using the arm library.

Coordinate System Origin

The "standard origin" of the arm library's base frame is a location which is at the approximate center of the working volume of the arm. This origin is chosen because it provides the most device-independent choice for applications which wish to use any arm device without difficulty.

Each arm device type also has a device-specific origin. This origin is used internally by the arm library, but may also be used by applications which need to accurately know what arm positions mean relative to the real world. The ArmLib_OpenArm() call allows either choice of origin. Additionally, the ArmLib_QueryConfig() call allows the application to determine the exact offset between the device-independent and device-dependent origins for a specific arm type.

The device-dependent origin for the ARM is the "base" of the arm, at the intersection of the "roll" and "swing" axes for the upper arm. The device-dependent origin for the Sarcos
arm is at the intersection of the first and second joint axes. The device-dependent origin for the Phantom is at the top, back (from sitting user), left corner of the “bar” forming the top of the device.

The old UNC arm library had a different convention for the arm library base frame. It used the "device-dependent" origin just mentioned for the Argonne arm as its base frame. This base frame was deemed too device-dependent to use a universal base frame for the new arm library.

**Position Units**

Position is measured in units of meters (m). In the old UNC arm library, position was measured in cm.

**Angular Units**

Angles are measured in radians. Thus, angular velocity is measured in radians/sec.

**Force Units**

Force is measured in units of Newtons (N). In the old UNC arm library, force was also measured in units of Newtons.

**Torque Units**

Torque (for moments) is measured in units of Newton-Meters (Nm). In the old UNC arm library, force was also measured in units of Newton-Meters.

**Time Units**

Time is represented throughout the arm library in the unix "struct timeval" format. This format provides a 32-bit unsigned integer representing seconds, and a 32-bit unsigned integer representing microseconds. Timestamps for position reads and force writes are always measured relative to the server machine's clock. For more information, see the discussion of *Time Measurement* in the *Arm Library Concepts* section.
V. Arm Library Concepts

Synchronous Operation

The most basic interactions between the client and the server are accomplished synchronously. In these cases, the client sends a request to the server, and then waits until it receives a response back before proceeding. All mode changing operations are accomplished this way (Open, Close, StartSurface, etc.) Position reading and force writing can be accomplished this way. For a position read, a request is sent to the server, the server reads the position and sends the result back to the client, who then continues on. For a force write, the client sends the request to the server then waits. The server processes the message and applies the force, then sends an acknowledgement back to the client.

In all cases, the client waits while the message makes a round trip (and the server performs the desired action). Since this type of waiting is undesirable in interactive applications, where the time could be spent on computation or display, the arm library also provides asynchronous operation modes for the most time critical and most often used operations.

Asynchronous Operation

When the arm library is put into an asynchronous read mode, the server sends a continuous stream of position/velocity updates to the client. When the application wants to read position, it simply scoops up one of these "streamed" reports. The decision as to whether to wait for the next report or use the previously received one is determined by the particular asynchronous mode which is used. The rate at which the server generates the position/velocity reports can also be controlled.

When the arm library is asked to perform an asynchronous write, the client sends the force output information to the server, and does not wait for an acknowledgement.

Most applications will benefit from the improved performance offered by the asynchronous modes of operation, and these modes are thus selected by default. It must be noted, however, that the current implementation of the asynchronous modes employ UDP communications services. As a result, messages are not guaranteed to be delivered. For example, it is possible that a valid force write request could be ignored by the server in asynchronous mode. It is also possible for messages to be received out of order, resulting in old information replacing newer information. We have observed, however, that these problems are not likely to be troublesome while the system is running over a tightly coupled LAN at moderately high update rates. Operation at very low update rates or over high latency WANs is not recommended in any mode, but especially not in asynchronous mode.

Event-Driven Operation

Many arm applications function by continuously running through a main loop which might look something like this:

while (!done) {
    Check for Keyboard Input, and act on it.
    Read Arm Position
Compute appropriate forces for this position
Write Arm Forces
}

This type of main loop leads to simple program structure and often works very well. Unfortunately, it does not work well in conjunction with an event-driven system such as X-windows. The arm library thus provides support for integration into an event-driven application.

When the library is in an asynchronous read mode, the user may, using the ArmLib_QueryConfig() call, obtain the number of a file descriptor which may be used in a unix select() call. The event-loop may then check this file descriptor, with a select(), to determine if new position information has become available.

By adjusting the "asyncintrvl" parameter to ArmLib_OpenArm, the interval between position update events can be controlled. Arm position updates can then be handled as just another event in an event-driven application. The event will typically be handled by computing forces appropriate for the new position, and sending these forces (asynchonously) to the arm.

**Time Measurement**

When using predictive techniques, it is important to be able to precisely measure time. Thus, the arm library device-driver standard currently requires that arm servers have clocks capable of measuring time to a resolution of 100 msec. All critical time measurements and computations are made relative to the server clock.

In all cases to date the 100 msec precision requirement has necessitated the development of special time routines, since the routines provided by the operating systems in use do not supply time with the necessary resolution. The Sarcos controller (a MV167 running VxWorks) obtains its precise time from a routine which reads a hardware timer on the MV167's pccc2 timer chip. This routine actually provides 1 msec precision. The ARM controller (a 486 PC running DOS) obtains its precise time from hardware timers on the National Instruments AT-MIO-64F-5 multifunction I/O board. The precision of this time is 100 msec. This precision could be improved by using a higher frequency clock to drive the timer. The Phantom controller (a Pentium PC also running DOS) uses a custom assembly language routine which relies the standard 8254 system timer chip. The precision of this time is nominally 1 msec, although it takes 10 msec or more just to query the clock.

Although the time values provided have a precision equal to or better than 100 msec, they are not necessarily accurate to 100 msec. It is expected that they will drift slowly over time. In addition, no effort is made to synchronize the client clocks with the server clock. In the case of the ARM server clock, the time is reset to 0:00 every time a new session is started. Thus, it is important that any adjustment of time values be done only using server times. For example, when doing position prediction, the "read time" is returned along with the data. The "predict interval" should be added to this time to produce the "output time" for the resulting force command. Thus, all time computations are made relative to the server clock.

If at some point it becomes important to render graphics images at a precise time in an arm library application, it would probably become necessary to synchronize the clocks on the arm server and the graphics rendering machine. A possible method of accomplishing this task is to distribute a central clock signal via coaxial cables to a special counter on each machine. Less accurate sychronization could be done via the standard Ethernet network.
Planar Approximation to Hard Surfaces

Ordinarily, arm library applications read a force using the `ArmLib_ReadPosVel()` call, and then apply a force using the `ArmLib_SendForce()` call. The resulting control loop requires a round-trip between the client and server machines, thus limiting the force update rate. Since high-quality hard surfaces require a high update rate, the library provides the ability to close the force-update loop on the server machine by downloading a planar approximation to a surface.

By evaluating the position of the arm’s end effector relative to the defined plane, the server can determine whether or not the arm has penetrated the plane, and if so, by how much. Using this information, a linear restoring force is computed and applied. Once the plane equation is received by the server, no additional communication or client interaction is required. As a result, the server can autonomously adapt to changes in the arm’s position and orientation very rapidly, and ensure that the force applied is appropriate. Essentially this same approach is discussed in [Adachi, Y., et al, VRAIS ’95] under the name "Intermediate Representation", although we investigated and implemented the idea independently at UNC.

The plane is represented to the server in the following form:

\[ dx = Ax + By + Cz + D \]

The plane can be specified by setting A,B,C,D for the plane equation above, or by defining a normal N and a point on the plane P0.
In order to simulate surfaces more complex than a simple plane, the client application must periodically update the plane equation so that it remains a good approximation of the local hard surface being modeled.

• It is often desirable to model a virtual probe that is larger than a single point. To support this goal, the arm library allows the user to define a set of virtual probes, or “fingers”, each of which can be constrained by its own respective plane. This feature also allows for more complex surface representations. For example, one could model the corner of a box by setting three fingers to the same position, and setting their three respective planes to the three planes of the box.

• Friction is a very important component of haptic feedback; without it, all surfaces would feel like oiled glass. What is required is a simple friction model that allows for a variety of simulated surfaces without compromising update time. The model used in the arm library treats each finger as a probe passing over a surface that has alternating smooth and sticky spots, or snags. When moving across a smooth spot the probe is opposed by a force proportional to kK, the coefficient of kinetic friction, so that \( \text{opposing.force} = kK * \text{current.normal.force} \). After some random distance dependant on the variables MeanDist and SigmaDist (the mean distance and standard deviation between snags, respectively), the probe encounters a snag. There the probe will catch and bend, outputting a force proportional to the variable kStick, until it moves dSnap in distance and pops out. While this model is not a perfect physical model, it does meet the goals above.
• The local surface representation just discussed is very useful for representing object boundaries, but is not appropriate for representing more general force fields. To address this need, the arm library provides a second kind of local constraint, point-to-point springs. The fingers can be constrained by springs attached to points in space in addition to the previously discussed surface constraints. This provides a local first order approximation to a general force field. The springs are specified by setting a point in space, a spring constant, and the rest length of the spring. By setting the attachment point appropriately, the application can control the forces delivered to the user. As with surfaces, multiple fingers with multiple springs can be specified to create more complex forces. Each finger, in fact, can be constrained by both a surface and a spring at the same time.

To accomplish the operations required for the local planar approximation, surface friction, and point-to-point springs, the arm library provides three calls: ArmLib_StartSurface(), ArmLib_StopSurface(), and ArmLib_SendSurface(). These routines are described in detail in the API section of this document.
VI. Application Programmer Interface

Overview

∞ All of the globally visible armlib routines and variables begin with the prefix ArmLib_. This prefix is used to prevent naming conflicts with user routines. Some #define'd constants begin with AL_.

∞ All armlib routines take a single C parameter. This parameter is a pointer to a structure which contains the actual parameters. This technique is used because it (1) allows easy addition of new parameters without requiring changes to applications, (2) allows defaults to be easily established, and (3) makes it simple for the application to ignore parameters for which the default is adequate. The name of the parameter structure for a given routine is derived from the name of that routine by using the prefix ArmLib_P_ instead of ArmLib_. For example, the parameter structure for the ArmLib_OpenArm routine is named struct ArmLib_P_OpenArm.

∞ Before using a parameter structure, a routine must be called to initialize the parameter structure and set the default parameters. This call need only be made once for a given parameter structure, but it may be called again later to reset the parameters in the structure to their default values. The name of this initialization routine is derived from the name of the "actual" routine by using the prefix ArmLib_I_ instead of ArmLib_.

∞ All armlib routines return a success/failure status code. The return value is either ArmLib_ERROR or ArmLib_SUCCESS.

∞ Most armlib routines require an "opennum" parameter. This parameter allows the arm library to distinguish between multiple arms that may be open. This parameter always defaults to zero. If a second arm is opened, all calls referring to it should use a different number (such as 1).

Example code fragment, which calls the ArmLib_Open routine:

```c
#include <armlib.h>

int ret;
struct ArmLib_P_OpenArm p;    /* Parameter structure */
ArmLib_I_OpenArm(&p);          /* Init param struct */
p.opennum = 0;                 /* Set opennum parameter */
...                           /* Set other parameters */
ret = ArmLib_OpenArm(&p);      /* Call the routine */
if (ret != ArmLib_SUCCESS)     /* Check return value */
    fprintf(stderr, "appl: Failed to open arm\n");
```
Basic Routines

(Default parameter values are in brackets after the name of the parameter. Unless otherwise specified, the arm library treats all parameters as “read-only”. That is, they are not modified and may be reused.)

ArmLib_OpenArm

Parameter Summary:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>opennum</td>
<td>0</td>
</tr>
<tr>
<td>*whicharm</td>
<td>&quot;&quot;, meaning examine ARM0, or use Phantom</td>
</tr>
<tr>
<td>tooltype</td>
<td>ArmLib_Tooltype_Hand</td>
</tr>
<tr>
<td>orig_wvcneter</td>
<td>1 = true</td>
</tr>
<tr>
<td>stdscale</td>
<td>1 = Scale both positions and forces</td>
</tr>
<tr>
<td>predictintrvl</td>
<td>0 = no prediction</td>
</tr>
<tr>
<td>jointflag</td>
<td>0 = no joint info</td>
</tr>
<tr>
<td>readmode</td>
<td>ArmLib_Read_Mode_Latest_Unread</td>
</tr>
<tr>
<td>timeval</td>
<td>device-dependent default</td>
</tr>
</tbody>
</table>

Detailed Routine/Parameter Description:

This routine opens an arm for use. This open call is necessary before positions can be read from, or forces written to, an arm. If multiple arms are used by an application, each one must be individually opened. For many applications, the default values will be correct for all parameters.

opennum  An integer used by the application to distinguish between multiple arms that are open. If a single arm is opened, the programmer will normally use the default value of 0. This number must be between 0 and ArmLib_MaxArms-1. This same number must be used in later API calls dealing with the same arm.

whicharm  A string indicating which physical arm should be opened. The three current choices are “Phantom”, "ARM" and "Sarcos". Alternatively, if whicharm is the null string (""), then the environment variable ARMx is examined to determine which arm should be opened, where x is the opennum described above. Finally, if whicharm begins with the "@" character, then the environment variable specified by the remainder of the whicharm string will be examined. For the usual case where opennum == 0, ARM0 will be examined (for the string “Phantom”, "ARM" or "Sarcos"). If the environment variable is undefined, the default arm (currently Phantom) will be used. The default value for the whicharm parameter is "".

tooltype  A number indicating which tool is mounted on the end of the arm. In practice this option is used to decide what the origin and orientation of the tool coordinate frame are, so different tooltypes can be used with the same physical end effector to give different origins for the end effector frame.

Values for tooltype (more could be added to armlib):
<table>
<thead>
<tr>
<th>ArmLib_Tooltype_LastJoint</th>
<th>Use last joint’s Denavit-Hartenberg coordinate frame as the tool frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArmLib_Tooltype_Hand</td>
<td>Use a location corresponding to the center of the user’s hand as the tool frame. With the user in the &quot;resting&quot; position, the orientation of this frame is the same as that of the arm library base frame.</td>
</tr>
<tr>
<td></td>
<td>The default value is ArmLib_Tooltype_Hand, since it is a device-independent choice.</td>
</tr>
</tbody>
</table>

| orig_wvcenter            | A flag which if set (!= 0), indicates that the armlib should use a base coordinate system with its origin at the center of the device’s working volume. If not set (= 0), indicates that the armlib should use a device-specific origin. The default is to set the flag, since this choice provides device-independence. |

| stdscale                 | A variable which controls the scaling of coordinates and forces to a standard sized working volume and force. The standard sized working volume is that of a the Sarcos arm, approximately 0.5 m x 0.5 m. The standard forces are also appropriate for the Sarcos arm. With scaling enabled, the application should request forces appropriate for the Sarcos arm. They will be scaled appropriately for the device which has been opened. stdscale = 0: Do not scale either position or forces stdscale = 1: Scale both positions and forces stdscale = 2: Scale positions but not forces stdscale = 3: Scale forces but not positions |

| predictintrvl            | Predict-ahead interval, in microseconds. If this value is non-zero, then when a position is requested, the actual position is not returned. Instead, a predicted future position is returned. predictintrvl determines how far into the future the prediction routine should predict. The default value is currently 0, ie. no prediction. The prediction feature is not currently implemented. |

| jointflag                | Flag indicating whether information should be provided about joint angles. If this flag is not set, only position and orientation are returned on read calls. If this flag is set, information on individual joint angles is also returned. It is not recommended that this flag be set unless the extra information is needed, since the extra information slows down client-server communications slightly. The default value is jointflag = 0. Any other value is currently unsupported. |
readmode

Initial position reading mode. This mode determines how position and velocity data is acquired from the server. This mode may be changed later with the `ArmLib_StartStream()` and `ArmLib_StopStream()` calls. The four options are:

- **ArmLib_Read_Mode_Sync**
  Synchronous reads (query server, wait for response)

- **ArmLib_Read_Mode_Latest**
  Asynchronous (use most recent data received even if it was already returned by the previous Read)

- **ArmLib_Read_Mode_Latest_Unread**
  Asynchronous (use most recent data received if it hasn't been already returned by a read. Otherwise, wait for next data report from the server)

- **ArmLib_Read_Mode_Next_Receive**
  Asynchronous (wait for next data report from the server, and return that).

The default [Subject to change] is **ArmLib_Read_Mode_Sync**.

asyncintrvl

Update interval for asynchronous read modes. This interval indicates how frequently the arm server sends position data asynchronously to the client. The default is device-specific and should normally be left alone if the application has a tight main loop. If the application's main loop is very long, then a longer interval could be used which is somewhat less than the time through the main loop.
**ArmLib_CloseArm**

**Parameter Summary:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>opennum</td>
<td>int</td>
<td>The same number used when opening the arm.</td>
</tr>
<tr>
<td>statflag</td>
<td>int</td>
<td>Indicates whether or not to print statistics.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = don't print performance stats [default]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = print performance stats summary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = print full performance stats</td>
</tr>
</tbody>
</table>

Note that the printing of performance statistics is not currently fully implemented.
ArmLib_ReadPosVel

Parameter Summary:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int opennum</td>
<td></td>
<td>The same number used when opening the arm.</td>
</tr>
<tr>
<td>ArmLib_FOURMAT T</td>
<td>(RETURNED)</td>
<td>A 4x4 transformation matrix. Currently, the element type is float. The format of the matrix is as follows:</td>
</tr>
<tr>
<td>ArmLib_THREEVEC Vel</td>
<td>(RETURNED)</td>
<td>A three element vector giving the velocity of the tool frame. Vel[0] = xvelocity, Vel[1] = yvelocity, Vel[2] = zvelocity. Note that these values will be affected by the setting of the stdscale flag in ArmLib_OpenArm().</td>
</tr>
<tr>
<td>ArmLib_THREEVEC OrientVel</td>
<td>(RETURNED)</td>
<td></td>
</tr>
<tr>
<td>struct timeval readtime</td>
<td>(RETURNED)</td>
<td></td>
</tr>
<tr>
<td>short numjoints</td>
<td>(RETURNED)</td>
<td></td>
</tr>
<tr>
<td>ArmLib_JOINTVEC Joints</td>
<td>(RETURNED)</td>
<td></td>
</tr>
</tbody>
</table>

Detailed Routine/Parameter Description

This routine reads position and (optionally) velocity and joint data from an arm. Velocity data is only returned if it was requested in the ArmOpen call. Joint angles are also returned if they were requested in the ArmOpen call.

The position/velocity/joint data is obtained from the arm server using the current read mode. Refer to the description of the ArmOpen readmode parameter for details.

opennum The same number used when opening the arm.

T A 4x4 transformation matrix. Currently, the element type is float. The format of the matrix is as follows:

T[0][0] = a1  T[0][1] = a4  T[0][2] = a7  T[0][3] = xpos
T[1][0] = a2  T[1][1] = a5  T[1][2] = a8  T[1][3] = ypos
T[3][0] = 0  T[3][1] = 0  T[3][2] = 0  T[3][3] = 1

Where a1-a9 are the components of the 3x3 orientation matrix used to rotate from the base frame to the tool frame, and xpos, ypos, and zpos give the position, in meters, of the tool frame relative to the base frame. So, if P is a point represented in the tool coordinate system, and Q is the same point represented in the base coordinate system, then Q = T*P. Q and P are both column vectors. The tool frame is normally defined so that the orientations of the tool frame and base frame are the same when the user holds the arm in a standard "resting" position. Note that xpos, ypos, and zpos will be affected by the setting of the stdscale flag in ArmLib_OpenArm().

Vel A three element vector giving the velocity of the tool frame. Vel[0] = xvelocity, Vel[1] = yvelocity, Vel[2] = zvelocity. Note that these values will be affected by the setting of the stdscale flag in ArmLib_OpenArm().
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
</table>
| OrientVel | A three element vector giving the "orientation velocity" of the tool frame.  
\( \text{OrientVel}[0] \) = rotation about x axis, in radians/sec.  
\( \text{OrientVel}[1] \) = rotation about y axis, in radians/sec.  
\( \text{OrientVel}[2] \) = rotation about z axis, in radians/sec. |
| readtime | Time at which the position data was read. The time is given relative to the server's clock. When position prediction is used, the data returned is the predicted position at \( \text{readtime} + \text{predictinterval} \). See the documentation for the ArmOpen routine for details. |
| numjoints | Not fully implemented or documented yet. Intended to allow the return of joint angles. |
| Joints | Not fully implemented or documented yet. Intended to allow the return of joint angles. |
**ArmLib_WriteForce**

**Parameter Summary:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>opennum</td>
<td>The same number used when opening the arm.</td>
</tr>
<tr>
<td>ArmLib_THREEVEC</td>
<td>FVec</td>
<td>A three-element vector of float values, in Newtons, giving the forces to be applied. FVec[0] = x-force, FVec[1] = y-force, FVec[2] = z-force. Defaults to all zeros. This parameter will be affected by the setting of the stdscale flag in ArmLib_OpenArm().</td>
</tr>
<tr>
<td>ArmLib_THREEVEC</td>
<td>MVec</td>
<td>A three-element vector of float values, in Newton-meters, giving the moments to be applied. MVec[0] is the torque about the X-axis, etc. Defaults to all zeros. This parameter will be affected by the setting of the stdscale flag in ArmLib_OpenArm().</td>
</tr>
<tr>
<td>struct timeval</td>
<td>writetime</td>
<td>The time, relative to the server's clock, at which the forces/moments should be applied. It is intended that this parameter be used in conjunction with Kalman filtering, where writetime would be set equal to readtime+predictintrvl. If the writetime parameter is {0,0}, then the write will occur with no delay.</td>
</tr>
<tr>
<td>int</td>
<td>writemode</td>
<td>The mode used for the write. There are two options:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ArmLib_Write.Mode_Sync</strong> Perform a synchronous write. ie, send the command to the server, then wait for an acknowledgement.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ArmLib_Write.Mode.Async</strong> Perform an asynchronous write. There is no wait for an acknowledgement from the server.</td>
</tr>
</tbody>
</table>

**Detailed Routine/Parameter Description:**

This routine causes a message to be sent to the arm server to apply a certain force and moment (torque) to the user's hand. The vectors FVec and MVec are defined with respect to the standard base frame, but with its origin shifted to the current tool position. FVec and MVec default to zero, but the programmer should be careful when relying on this (ie, call ArmLib_I_WriteForce to reset the defaults before every call).
**ArmLib_StartStream**

**Parameter Summary:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Parameter</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>opennum</td>
<td>[0]</td>
</tr>
<tr>
<td>struct timeval</td>
<td>asyncintrvl</td>
<td>[device-dependent default]</td>
</tr>
<tr>
<td>int</td>
<td>ReadMode</td>
<td>[ArmLib_Read_Mode_Latest_Unread]</td>
</tr>
</tbody>
</table>

**Detailed Routine/Parameter Description**

This routine causes the arm server to begin asynchronous transmission of position data. It also sets the readmode to be used by `ArmLib_ReadPosVel()`. Note that it is unnecessary to call this routine if the appropriate readmode was supplied to `ArmLib_OpenArm()`. Typically this routine would be called to change modes or to resume asynchronous operation after a call to `ArmLib_StopStream()`.  

- **opennum** The same number used when opening the arm.
- **asyncintrvl** The interval for asynchronous position updates from the server. See `ArmLib_OpenArm()` for a detailed description.
- **readmode** The read mode. See `ArmLib_OpenArm()` for a detailed description. The default is `ArmLib_Read_Mode_Latest_Unread`. 
**ArmLib_StopStream**

**Parameter Summary:**

```c
int opennum [0]
```

**Detailed Routine/Parameter Description:**

This routine informs the arm server that it should cease asynchronous transmission of position data. Any future calls to `ArmLib_ReadPosVel()` will be satisfied synchronously.

| opennum | The same number used when opening the arm. |
ArmLib_SetMsg

Parameter Summary:

int msglvl [AL_NORMAL]

Detailed Routine/Parameter Description:

This routine tells the arm library what type of error, warning, debugging, informational, etc. messages should be printed.

msglvl | A bit mask indicating which messages types should be printed. The following bits may be set:
---|---
AL_ERROR | Fatal errors (-> function exit)
AL_WARNING | Non-fatal errors
AL_MESSAGE | Various informational msgs
AL_ENTRY | Msg at each function entry
AL_EXIT | Msg at each function exit
AL_CHECKPOINT | Debugging checkpoints

The following constants combine the above bits in useful ways:

AL_SILENT | No messages
AL_NORMAL | Errors, warnings, and msgs
AL_EVERYTHING | Everything.

The default value for the msglvl parameter is AL_NORMAL. Other settings may produce extremely verbose output.
ArmLib_QueryConfig

Parameter Summary:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int opennum</td>
<td>The same number used when opening the arm.</td>
</tr>
<tr>
<td>int armtyp</td>
<td>The type of arm which has been opened. Currently the three possibilities are ArmLib_Armtype_ARM, ArmLib_Armtype_Sarcos, and ArmLib_Armtype_Phantom.</td>
</tr>
<tr>
<td>char * name</td>
<td>A short name for the arm type. This is essentially just a string version of armtyp.</td>
</tr>
<tr>
<td>char * addevice</td>
<td>The name of the button/knob A/D device associated with this arm. It is intended that this name then be passed to UNC’s adlib ad_open() routine. The name is obtained from the environment variable ARMADx, where x is the opennum. If this environment variable is not set (probably the typical case), then an arm-specific default A/D device name is returned.</td>
</tr>
<tr>
<td>float baseoffset[]</td>
<td>The X, Y, and Z offset from the standard coordinate frame origin (at center of working volume) to the device-specific origin. So,</td>
</tr>
<tr>
<td>int datafd</td>
<td>File descriptor number which may be used in a select() call to determine when there is asychronous position data available from the server. This feature was added so that applications with an event loop, such as X-windows applications, could use a select() call to determine when to call ArmLib_ReadPosVel. X-Windows provides a call which can be used to register a callback based on the results of a select().</td>
</tr>
</tbody>
</table>

Detailed Routine/Parameter Description:

This routine allows the application to obtain various information about an arm which has been opened.

opennum The same number used when opening the arm.

armtype The type of arm which has been opened. Currently the three possibilities are ArmLib_Armtype_ARM, ArmLib_Armtype_Sarcos, and ArmLib_Armtype_Phantom.

name A short name for the arm type. This is essentially just a string version of armtyp.

addevice The name of the button/knob A/D device associated with this arm. It is intended that this name then be passed to UNC’s adlib ad_open() routine. The name is obtained from the environment variable ARMADx, where x is the opennum. If this environment variable is not set (probably the typical case), then an arm-specific default A/D device name is returned.

baseoffset The X, Y, and Z offset from the standard coordinate frame origin (at center of working volume) to the device-specific origin. So,

Xpos_in_device_coord = Xpos_in_std_coord + baseoffset[0];
Ypos_in_device_coord = Ypos_in_std_coord + baseoffset[1];
Zpos_in_device_coord = Zpos_in_std_coord + baseoffset[2];

datafd File descriptor number which may be used in a select() call to determine when there is asychronous position data available from the server. This feature was added so that applications with an event loop, such as X-windows applications, could use a select() call to determine when to call ArmLib_ReadPosVel. X-Windows provides a call which can be used to register a callback based on the results of a select().
Planar Surface Approximation Routines

ArmLib_StartSurface

Parameter Summary:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>opennum</td>
<td>The same number used when opening the arm.</td>
</tr>
<tr>
<td>int</td>
<td>fingernum</td>
<td>The finger to be used. Indexes between 0 and ARMLIB_MAX_FINGERS-1. A finger must be turned on by a call to ArmLib_StartSurface() before it can be used with ArmLib_SendSurface().</td>
</tr>
<tr>
<td>int</td>
<td>NRecCycles</td>
<td>The number of update cycles taken to smoothly change from one surface definition (given by ArmLib_SendSurface()) to another. If NRecCycles ≤ 0, then no recovery will be performed. This value should really be in seconds, and will be changed in a future version.</td>
</tr>
<tr>
<td>struct</td>
<td>timeval</td>
<td>The time allowed to elapse before the force output is recomputed by evaluating the current plane equation. If asyncintrvl is 0 (the default), the server will update the force output as often as possible.</td>
</tr>
</tbody>
</table>

Detailed Routine/Parameter Description

This routine causes the arm server to begin periodically evaluating the current plane equation with the specified interval between iterations. Once this routine is called, ArmLib_WriteForce() should not be called until ArmLib_StopSurface() is called, since the force output will be recomputed and overridden every time the plane equation is evaluated.

opennum: The same number used when opening the arm.

fingernum: The finger to be used. Indexes between 0 and ARMLIB_MAX_FINGERS-1. A finger must be turned on by a call to ArmLib_StartSurface() before it can be used with ArmLib_SendSurface().

NRecCycles: The number of update cycles taken to smoothly change from one surface definition (given by ArmLib_SendSurface()) to another. If NRecCycles ≤ 0, then no recovery will be performed. This value should really be in seconds, and will be changed in a future version.

asyncintrvl: The time allowed to elapse before the force output is recomputed by evaluating the current plane equation. If asyncintrvl is 0 (the default), the server will update the force output as often as possible.
**ArmLib_StopSurface**

**Parameter Summary:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int opennum</td>
<td>The same number used when opening the arm.</td>
</tr>
<tr>
<td>int fingernum</td>
<td>The finger we wish to stop evaluating.</td>
</tr>
</tbody>
</table>

**Detailed Routine/Parameter Description**

This routine tells the server to stop evaluating the current plane equation to generate forces for the given finger. Upon receipt, the server immediately sets the force output to zero for that finger. Once this routine is called for all previously active fingers,ArmLib_WriteForce() may again be used to directly control the force output.
ArmLib_SendSurface

Parameter Summary:

int opennum [0]  
int fingernum [0] 
int setmask [0]  

_used only if (setmask & ARMLIB_SET_FINGER)_
float O[3] [0,0,0]  

_used only if (setmask & ARMLIB_SET_PLANE)_
float N[3] [0,0,0] 
float P0[3] [0,0,0] 
float A [0] 
float B [0] 
float C [0] 
float D [0] 

_used only if (setmask & ARMLIB_SET_SURF)_
float kN [0] 
float Kspring [0] 
float kStick [0] 
float kK [0] 
float MeanDist [0] 
float SigmaDist [0] 
float dSnap [0] 
float kS [0] 

_used only if (setmask & ARMLIB_SET_SPRING)_
float kAttach [0] 
float RestLen [0] 
float AttachPt [0,0,0] 

Detailed Routine/Parameter Description

This routine causes a message to be sent to the arm server to apply a certain force and moment (torque) to the user's hand. The vectors FVec and MVec are defined with respect to the standard base frame, but with its origin shifted to the current tool position. FVec and MVec default to zero, but the programmer should be careful when relying on this default (ie, call ArmLib_I_WriteForce() before every call).

opennum Specifies which of the open arms should be affected by this call. The same number used when opening the arm should be used here.

fingernum Specifies which finger is affected by this call.

setmask Specifies which set(s) parameters are active for this call. If the flag ARMLIB_SET_FINGER is enabled, then the finger position will be set. If the flag ARMLIB_SET_PLAN is enabled, then the plane definition will be read and set. If ARMLIB_SET_SURF is enabled,
then the friction parameters for the surface will be set. If ARMLIB_SET_SPRING is enabled, then the point-to-point spring for this finger will be set. More than one flag can be enabled; the armlib will set all appropriate parameters.

O Offset from the current tracker position to this finger in hand coordinates. This means that if the hand rotates, so does the finger. This value is measured in meters (m), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies.

N Normal to the plane, with magnitude normalized to 1. If π, then N and P0 will be used to compute the plane, and A,B,C,D below will be ignored.

P0 A point on the plane where the plane equation is valid. It is measured in meters (m) in the coordinate system specified by ArmLib_OpenArm(), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies.

A, B, C, D Coefficients of the plane equation $Ax + By + Cz + D = 0$. The equation will be evaluated in the armlib’s current frame of reference, as set in the call to ArmLib_OpenArm(). A, B, and C are unitless. D is in units of meters (m), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies. These parameters are included for backwards compatibility, and will be ignored if N and P0 are set.

kN This factor determines the “stiffness” of the surface. Its units are Newtons/meter (N/m). The distance the arm has penetrated the surface (if at all) is multiplied by this factor to arrive at the magnitude of the restoring force. (The direction of the restoring force is always normal to the plane.) This value has force and/or position scaling applied to it, if the stdscale parameter of ArmLib_OpenArm() so specifies.

Kspring Another name for kN. Included for backwards compatibility, and only used if kN £ 0.

kStick The spring constant of the probe while caught on a snag. It is measured in Newtons/meter (N/m) and has force and/or position scaling applied to it, if the stdscale parameter of ArmLib_OpenArm() so specifies.

kK The coefficient of kinetic friction. Specifies the force when the probe is moving through smooth patches. Unitless.

MeanDist The mean distance between snags. This is measured in meters (m), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SigmaDist</td>
<td>The standard deviation between snags. SigmaDist and MeanDist specify a Gaussian distribution which is used to randomly determine the next time the probe will hit a snag. This is measured in meters (m), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies.</td>
</tr>
<tr>
<td>dSnap</td>
<td>The displacement distance after which the probe snaps out of the snag, i.e. out of static friction back into kinetic friction. This is measured in meters (m), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies.</td>
</tr>
<tr>
<td>kS</td>
<td>The maximum static friction, defined to be equal to dSnap*kStick. It is measured in Newtons and has force scaling applied to it, if the stdscale parameter of ArmLib_OpenArm() so specifies. Note that either kS or dSnap should be &gt;0, and the other will be derived from it. If both are set, dSnap will be used. Note that kS is not necessarily the coefficient of static friction (which is unitless), though it may be defined that way in a future version.</td>
</tr>
<tr>
<td>kAttach</td>
<td>Spring constant of the attached spring. The force computed will be kAttach*(displacement - RestLen), which is a standard Hoare spring. It is measured in Newtons/meter (N/m) and has force and/or position scaling applied to it, if the stdscale parameter of ArmLib_OpenArm() so specifies.</td>
</tr>
<tr>
<td>RestLen</td>
<td>Resting length of the attached spring. Measured in meters (m) and has position scaling applied to it, if the stdscale parameter of ArmLib_OpenArm() so specifies.</td>
</tr>
<tr>
<td>AttachPt</td>
<td>Attachment point of the other end of the spring. Measured in meters (m) in the coordinate system specified by ArmLib_OpenArm(), and will have position scaling applied to it if the stdscale parameter of ArmLib_OpenArm() so specifies.</td>
</tr>
</tbody>
</table>
**User Definable Extensions**

These have yet to be documented.
VII. Compiling and Linking

At UNC

When compiling an application, the include directory /afs/unc/proj/grip/arm/armlib/src should be used. Alternatively, as a convenience for UNC HMD library users, /afs/unc/proj/hmd/include may be used, since the necessary include files have a softlink from that directory to their true location.

When linking an application, the library directory /afs/unc/proj/grip/arm/armlib/src/hw_os should be used. Alternatively, the directory /afs/unc/proj/hmd/lib/hw_os may be used. It has softlinks in it to the appropriate files.

So, a typical compile will include the switch 
"-I/afs/unc/proj/grip/arm/armlib/src".

A typical link will include the switches 
"-L/afs/unc/proj/grip/arm/armlib/src/hw_os -larm."

If the compatibility library is being used (UNC only), then the link will also need 
"-L/afs/unc/proj/hmd/lib/hw_os -lsdi -lad"

Not at UNC

Suppose that you installed the arm library in /usr/local/lib/armlib at your site, with the .a file residing in the default “bin” subdirectory of that directory.

Then, a typical compile for a force-feedback application would include the switch: 
“-I/usr/local/lib/armlib"

A typical link for a force-feedback application would include the switches: 
“-L/usr/local/lib/armlib/bin -larm”
VIII. Example Programs

Program Using a Single Arm

/*************************************************************************
 * pp.c -- Print arm position every second *
 *************************************************************************

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include "armlib.h"

int main(int argc, char** argv) {

    int ret, i, j;
    struct ArmLib_P_OpenArm p;
    struct ArmLib_P_CloseArm p2;
    struct ArmLib_P_ReadPosVel rpvInit, rpv;
    struct timeval tdelay;

    /* open the arm */
    ArmLib_I_OpenArm(&p);
    p.whicharm = ""; /* use ARM0 variable */
    p.orig_wvcenter = 0; /* use device-specific origin */
    p.stdscale = 0;     /* do not scale to standard scale */
    ret = ArmLib_OpenArm(&p);
    if (ret != 0) {
        fprintf(stderr, "pp: error opening arm\n");
        exit(1);
    }

    /* Initialize the parameter structures once, then reuse */
    ArmLib_I_ReadPosVel(&rpvInit);
    ArmLib_I_ReadPosVel(&rpv);

    /* Continue testing until broken */
    while(1) {
        tdelay.tv_sec = 1; tdelay.tv_usec = 0;
        select(0,NULL,NULL,NULL,&tdelay); /* sleep 1 second */

        /* Read a position from the arm */
        ret = ArmLib_ReadPosVel(&rpv);
        if (ret != 0) {
            fprintf(stderr, "pp: err reading pos\n");
            exit(1);
        }
    }
}


} /* Print out the values */
printf("T:\n");
for (i=0;i<4;i++) {
    for (j=0;j<4;j++)
        printf(" %8.4f", rpv.T[i][j]);
    printf("\n");
}
printf("\n");

} /* while */

/* Never reached due to while(1), but the right code */
ArmLib_I_CloseArm(&p2);
ret = ArmLib_CloseArm(&p2);
if (ret != 0) fprintf(stderr, "pp: error closing arm\n");
exit(1);

printf("pp: successfully ran test. Exiting...\n");

} /* main */
Program Using Two Arms

/*
 * twoarmtest.c -- program to demonstrate simultaneous
 * use of two arms.
 *
 * The two arms should behave as if they are at
 * opposite ends of a rubber band -- ie, they are pulled
 * towards each other (in center-of-workingspace
 * coordinate frame).
 *
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <errno.h>
#include <math.h>
#include "armlib.h"

int main() {
    int ret;
    struct ArmLib_P_OpenArm    oa;
    struct ArmLib_P_CloseArm   ca;
    struct ArmLib_P_ReadPosVel rpv0, rpv1;
    struct ArmLib_P_WriteForce wf;
    int done, i;
    fd_set rdfds;
    struct timeval tmout = {0,0};
    float p0x, p0y, p0z, p1x, p1y, p1z;
    float dx, dy, dz;
    float diffmag;
    float forcemult = 170;  /* Distance to force multiplier
                       (units = Netwtons/meter) */

    /* Open arms */
    ArmLib_I_OpenArm(&oa);
    oa.opennum = 0;
    oa.stdscale = 0;
    oa.readmode = ArmLib_Read_Mode_Latest_Unread;
    oa.whicharm = ""; /* Let ARM0 determine it */
    ret = ArmLib_OpenArm(&oa);
    if (ret != 0)
        {fprintf(stderr, "twoarm: err opening arm1\n"); exit(1);}

    ArmLib_I_OpenArm(&oa);
    oa.opennum = 1;
    oa.stdscale = 0;
oa.readmode = ArmLib_Read_Mode_Latest_Unread;
oa.whicharm = ""; /* Let ARM1 determine it */
oa.readmode = ArmLib_Read_Mode_Latest_Unread;
ret = ArmLib_OpenArm(&oa);
if (ret != 0)
    {fprintf(stderr, "twoarm: err opening arm2\n"); exit(1);}

printf("Successfully opened both arms.\n  "Hit any key to abort test.\n");

done = 0;
while (!done) {
    /* Read positions from both arms */
    ArmLib_I_ReadPosVel(&rpv0);
    rpv0.opennum = 0;
    ret = ArmLib_ReadPosVel(&rpv0);
    if (ret != 0)
        {fprintf(stderr,"twoarm: err read arm1\n"); exit(1);}

    ArmLib_I_ReadPosVel(&rpv1);
    rpv1.opennum = 1;
    ret = ArmLib_ReadPosVel(&rpv1);
    if (ret != 0)
        {fprintf(stderr,"twoarm: err read arm2\n"); exit(1);}

    p0x = rpv0.T[0][3];
    p0y = rpv0.T[1][3];
    p0z = rpv0.T[2][3];
    plx = rpv1.T[0][3];
    ply = rpv1.T[1][3];
    plz = rpv1.T[2][3];

    /* Differences */
    dx = p0x-plx;
    dy = p0y-plx;
    dz = p0z-plz;

    diffmag = sqrt( (double) (dx*dx + dy*dy + dz*dz));

    /* Send the force */
    ArmLib_I_WriteForce(&wf);
    wf.opennum = 0;
    wf.FVec[0] = -dx*forcemult;
    wf.FVec[1] = -dy*forcemult;
    wf.FVec[2] = -dz*forcemult;
    ret = ArmLib_WriteForce(&wf); /* Force to 1st arm */
    if (ret != 0)
        {printf("twoarm: error writing 1st arm\n"); exit(1);}
    ArmLib_I_WriteForce(&wf); /* Force to 2nd arm */
if (ret != 0)
    {printf("twoarm: error writing 2nd arm\n"); exit(1);} 

    /* Check for key press */
    FD_ZERO(&rdfds);
    FD_SET(0, &rdfds);
    done = select(FD_SETSIZE,(void*)&rd fds, NULL, NULL, &tmout);
    if (done == EINTR) done = 0;
}

ArmLib_I_CloseArm(&ca);
ca.opennum = 0;
ret = ArmLib_CloseArm(&ca);

ArmLib_I_CloseArm(&ca);
ca.opennum = 1;
ret = ArmLib_CloseArm(&ca);

} /* main */
Program Using Dials/Buttons

/*
 * Example program to show use of arm buttons and dials.
 * Requires UNC's adlib
 * (link w/ '-lad', 'man adlib' for info).
*/
#include "armlib.h"
#include "ad.h"     /* adlib include file */

main() {

struct ArmLib_P_OpenArm     popn;
struct ArmLib_P_CloseArm    pcls;
struct ArmLib_P_QueryConfig pcfg;
char *addevice;

/* Declarations used w/adlib */
ad_index adIndex;
ad_a_type aValues[AD_MAX_NUM_CHANNELS];
ad_d_type dValues[AD_MAX_NUM_CHANNELS];

/* Open the arm normally */
ArmLib_I_OpenArm(&popn);
if (ArmLib_OpenArm(&popn) != ArmLib_SUCCESS) exit(1);

/* Find out what AD device to use for this arm.
 * There's a default, which may be overridden by setting
 * the ARMAD0 environment variable */
ArmLib_I_QueryConfig(&pcfg);
ArmLib_QueryConfig(&pcfg);
addevice = pcfg.addevice;

/* do the AD open, using device gotten from QueryConfig */
adIndex = ad_open((void *) addevice, 0);
if (adIndex == AD_NULL_DEVICE) {
    fprintf(stderr, "Error opening dials/buttons\n");
    return(-1);
}

while(1) {

    /* Do the AD read */
ad_read(adIndex, aValues, dValues);

    /* UNC's conventions for arm thumb and finger switches */
    if (dValues[1] & 1) printf("Thumb switch is down\n");
    if (dValues[0] & 1) printf("Finger switch is down\n");

    /* the dial values are in the aValues[] array
    * Values are in the range [-0.5, 0.5]          */
printf("Dial #0 = %f\n", aValues[0]);

} /* Close AD device */
ad_close(adIndex);

/* Close arm */
ArmLib_I_CloseArm(&pcls);
ArmLib_CloseArm(&pcls);

}
IX. Backward Compatibility Library

When this new arm library was written, many applications already existed at UNC which depended on the old arm library's API. We thus created a set of backward compatibility routines to support these applications. The backward compatibility routines form a layer on top of the new arm library — they perform various unit and coordinate transformations, and call the appropriate routines from the new library.

New applications should avoid using these backward-compatibility routines, and should be written to use the new API whenever possible. Even old applications can benefit from being modified to use the new library directly. The rewriting should be straightforward, with the possible exception of the change in standard coordinate systems between the old and new API.

For detailed documentation on the old API, see the old arm library manual written by Russell Taylor.

Backward Compatible Routines:

Arm_Open
Arm_Close
Find_Handgrip_Orientation
Send_Handgrip_Forces
Arm_Write
Force_Transform
FOURMAT_Multiply
FOURMAT_Copy

[Deliberately non-functional — Gives error if called]

Backward Compatible Global Variables:

ArmLib_FOURMAT Handgrip [Different contents, but it exists]
ArmLib_FOURMAT Identity [Same contents as before]
int ARM_Finger_Switch
int ARM_Thumb_Switch
float ARM_Knob_Values[9]

Backward Compatible Include File:

armcalls.h
X. ArmLib Internals

Diagnostic Routines

Device-specific diagnostic routines which are integrated into the arm server may be invoked as follows:

- Phantom (from DOS):  phantom -diag
- ARM (from DOS): argonne -diag
- Sarocs (from VxWorks shell):  adiag()

The diagnostics system then presents the user with a menu from which various options may be chosen.

Separately packaged diagnostic routines are also available. Some of these are detailed in the appendicies.

Source Files

At UNC these files are currently located in /afs/cs.unc.edu/proj/grip/arm/armlib/src. Note that any files which are needed by the server built on the PC must conform to the PC file naming restriction of eight characters plus a three character extension (yuk). Also, some of these files which are specific to UNC are not included in the ftp distribution.

Main Library

- armlib.h Include file for user programs. Contains definitions for user-callable routines. Only used for client side. #includes armlib_c.h and com.h
- armlib_c.h "Internal" include file, common to both client and server. Contains most of the general definitions.
- al_types.h Type definitions (client & server)
- messages.h Error/Debug message definitions (client & server).
- com.h Definitions for server & client communications routines. Also: Message ID's and structures for client <-> server messages. (client & server)
- schedule.h Definitions for timed callback routines (server only)
- timeops.h Definitions for time-manipulation routines (client & server)

- armlib_c.c Generate routines, common to server & client.
- com.c General (client & server) communications routines They invoke routines from the appropriate specific comm package.
- armcalls.c Client side routines, including API routines.
- server.c Server side routines (this is the "main" server code).
- callback.c Server side callback routines.
- schedule.c Server side callback scheduler.
- timeops.c Time manipulation routines (client & server)
Specific Communication Packages

- com_mix.h: Definitions (client & server) for "MIX" communications package
- com_tcp.h: Definitions (client & server) for "TCP" communications package
- com_mix.c: "MIX" comm package: routines common to server and client
- scom_mix.c: "MIX" comm package: server only routines
- ccom_mix.c: "MIX" comm package: client only routines
- com_tcp.c: "TCP" comm package: routines common to server and client
- scom_tcp.c: "TCP" comm package: server only routines
- ccom_tcp.c: "TCP" comm package: client only routines
Specific Devices/Servers

phantom.c  Phantom: device-specific routines for Phantom.
argonne.c  ARM: device-specific routines for ARM.
a_adcvt.c  ARM: convert from raw AD readings to joint angles
sarcos.c  Sarcos: device-specific routines for Sarcos.

Compatibility Library

armcalls.h  #include file for old arm library applications
compat.c  compatibility routines (a "wrapper" around new armlib which provides old armlib API)

Makefiles

Makefile  makefile to build client library.  Also, "make sarcos" will invoke vwmakefile
vwmakefile  Build the Sarcos server (invoke via make sarcos)
realmake  Build the ARM or Phantom server.  Must be invoked (via NFS) from the PC.  There is a very short makefile on the PC which accomplishes this invocation (calling it with either make phantom or make argonne)
argonne.lnk  options for the PC link command.  Used by "realmake".  Be careful when modifying this file.  If you get the commas, etc. wrong it is very easy to overwrite one of the link libraries with the executable.  Aggh.
phantom.lnk  same as above, but for building Phantom server.

Low-Level Testing

sizeof.c  Used during development to print out sizes of various data types.
ft.c  Applies a user specified constant force to the arm until the program is killed.

High-Level Testing, Demo Applications

(At UNC these files are located in /afs/unc/proj/grip/arm/test, and have their own makefile.  In the distribution, they are located in the test subdirectory.)
apptest.c  Simple test application
printpos.c  Print out arm position.
printpos_compat.c  Print out arm position, but using old arm library API.  Used to test compatibility library.
speedtest.c  Tests synchronous and asynchronous read rates.
oldtest.c  Test force output using old arm library API on top of the new library.  Used to test compatibility library.
oldoldtest.c  Same test as above, except actually uses the old arm library.
twoarmtest.c  Open two arms simultaneously.
Installing ArmLib

Extracting files

You obtain the arm library via ftp as a tar file. cd to the directory into which you wish to install the software (/usr/local/lib/armlib is used as the example in this manual), and extract the tar file:

tar -xvf armlib.tar

This extraction operation will leave a Makefile, .c and .h files in the installation directory, and will create a test subdirectory with a Makefile and .c files for some test programs.

Client-Side arm library

You will probably have to make some changes to the file armlib_c.c. The array ArmLib_X_Typeinfo[] contains information about each arm. Once piece of information is the name of the server machine. At UNC our phantom server is named "phantom". If the PC serving your phantom has a different TCP/IP name, then you’ll have to change this array. If you’re using a device type other than a Phantom you will have significantly more work to do. See the "Separation of Device-Dependent Routines" subsection of the "ArmLib organization" section of this manual for more details.

The .o and .a files for each architecture are stored in subdirectories of the top level armlib directory. If you installed armlib in /usr/local/lib/armlib at your site, then by default the .o and .a files will be stored in /usr/local/lib/armlib/bin. The makefile can be easily modified to store .o and .a files for different architectures in different subdirectories. To build the library, cd to /usr/local/lib/armlib and type make. The .o files are not needed after the make is complete and may be deleted to reclaim disk space.

Phantom Server

The most complex part of the installation is the installation of the software on the server. For a PC serving a Phantom, the following steps are required:

a) Buy a PHANToM, a PC, and an Ethernet card. Install DOS and the PHANToM software.
b) Buy a copy of Borland C/C++ and install it. Other C compilers might work, but we haven’t tested them.
c) Buy a copy of SunSelect’s PC-NFS and the PC-NFS programmer’s toolkit. Do NOT buy PC-NFS PRO, as it will not work with the programmer’s toolkit.
d) Install PC-NFS and the PC-NFS programmer’s toolkit, including the libraries for Borland C/C++.
e) Configure PC-NFS with all of the appropriate IP addresses, gateways, etc.
f) Copy all of the .c and .h files from the unix directory over to the PC (using ftp or whatever). Create a PC_DOS subdirectory.
g) Type “make -f reallmake phantom” in the directory containing the .c and .h files. Hopefully there are no problems with this step.
h) Reset the origin of the Phantom device as described in the phantom documentation. This must be done after every power-up of the phantom controller box.

i) Run the arm server by typing “phantom”.
   We recommend that you create a small batch file to rerun it every time it exits, since it will exit when any sort of error occurs.
   Then run your program on the client.

j) You can also run our diagnostic routines by typing “phantom -diag”

   If you plan to make any sort of significant changes to the armlib code, you will probably want to avoid having to recopy the files from unix to the PC every time you recompile the server (or worse, risk having the client and server files become inconsistent). At UNC we use PC-NFS to mount the unix partition containing our source code, and we have a small makefile on the PC which cd’s to the NFS mounted partition and invokes “realmake” to build the server (that’s how “realmake” got its name). Thus we keep all of the source code in one place.
Recompiling Test Applications

The test applications for the arm library are located at UNC in
/afs/unc/proj/grip/arm/armlib/test. In the distribution they are located in the test
subdirectory. They can be built for any architecture by typing make while in this directory.
The executables are placed in the appropriate architecture-specific subdirectory.

ArmLib Organization

Client/Server Model

The arm library is organized using a client-server model. The user's application resides
on the client machine, along with the client half of the arm library. The server half of the
arm library resides on the machine which controls the arm hardware. There is a separate
server machine for each force-feedback device.

The user's application makes calls to the client half of the arm library, which in turn
communicates (currently via TCP/UDP over ethernet) to the server half of the arm library.
Almost all of the actual "work" is done by the server half of the library. The server takes
care of the transformations from world space to joint space and vice-versa.

The file armcalls.c provides the "top-level" routines for the client side of the arm
library, while the file server.c provides the "top-level" routines for the server side of the
library.

Many utility and communications routines are used by both the client and the server.
The description of files in the "files" section of this manual indicates which files contain
routines used by both the client and the server.

Advantages of Client/Server Model:

1) The computational load can be spread among two machines. The client machine is free
to use its computational resources for graphics and modelling work, while the server
machine handles the arm library computations. The server may devote 100% of its time
to servicing the arm, leading to better control over the timing of position reads and force
writes.

2) Due to other constraints, the application may be required to run on a specific machine.
For example, to obtain good graphics performance applications must often run on the
graphics machine (an SGI, Pixel-Planes 5, etc.). Without the use of a client-server
model, the graphics machine would have to be directly connected to the force-feedback
arm hardware. The client-server model avoids this problem by allowing any machine
to be used as the client.

This flexibility is especially important in a research environment. For a "production"
VR system, it is much more likely that a single (possibly multi-processor) machine
would be used for controlling all I/O, including graphics and force-feedback. This
library could be easily adapted for such a machine by replacing the TCP/UDP
communications with shared-memory communications. See the later section on
communications routines for details.
3) The client-server model allows the library to easily support the use of different arms, or even serveral arms at once, from a single client machine.

Disadvantages of Client/Server Model:

1) Communications throughput and latency can be a problem.

2) Information is tightly compartmentalized between the client and server. Careful thought is required to make sure that the application has all of the information it needs.

**Separation of Device-Dependent Routines**

One of the goals when designing this arm library was to carefully compartmentalize device-dependent parts of the library. The device-specific routines, whose names begin with ArmLib_XA_ are located in separate files. sarcos.c contains the routines for the sarcos arm, argonne.c contains the routines for the ARM, and phantom.c contains the routines for the Phantom. For the ARM, some additional support routines (called from routines in argonne.c) are contained in a_adcvt.c. When building a server, the appropriate device-specific files must be compiled and linked in. The standard device-specific routines are the following:

**ArmLib_XA_gettimeofday(struct timeval *)**

Return time with precision equal to or better than 100 msec. Accuracy to real-world time not necessary.

**ArmLib_XA_ArmInit(void)**

Perform any device-dependent initialization which is required at server startup, including, if necessary, initialization of the XA_gettimeofday clock.

**ArmLib_XA_ArmStart(void)**

Perform any device-specific initialization which is required at ArmOpen time.

**ArmLib_XA_ArmFinish(void)**

Perform any device-specific cleanup which is required at ArmClose time.

**ArmLib_XA_ArmExit(void)**

Perform any device-specific cleanup which is required at server shutdown.

**ArmLib_XA_GetJointInfo(ArmLib_X_JointInfoArray)**

Stick alpha, theta, d, a, sin(alpha), cos(alpha) for every joint in the arm into the JointInfo array. (Matrix can be optionally used instead for any joint). In essence, this routine returns information about the physical configuration of the arm.

**ArmLib_XA_UpdateJointInfo(ArmLib_XA_JointInfoArray)**

Update theta and (optionally) theta_vel for each joint in the arm. In essence, this routine returns information about the arm's current
position and velocity.

\texttt{ArmLib_XA_WriteJointTorque(ArmLib_JOINTVEC)}

Cause the arm to output a set of torques to its joints. This is how forces are produced at the device-dependent level.

\texttt{ArmLib_XA_Diagnostics(void)}

Device-dependent diagnostics routine which can be invoked to perform tests. These tests are run on the server machine. A user-driven menu system is appropriate.

The device-specific routines for the Sarcos arm rely on some Sarcos-supplied routines. Due to the on-the-fly linking provided by VxWorks (the operating system which runs on the Sarcos controller), these routines do not need to be linked in at compile time. These routines are documented in the Sarcos Arm manuals. We also have the source code to these routines, and we have made some changes to this source code from the original Sarcos distribution (although Sarcos has since incorporated some of these changes into their newer software distributions).

In addition to allowing for device-specific routines, mostly for reading positions and writing forces, the arm library also has an array, \texttt{ArmLib_X_typeinfo}, which contains various pieces of important device-specific information about each arm type. This array is declared in \texttt{armlib_c.h}, and initialized in \texttt{armlib_c.c}.

Thus, in order to add a new arm type, the following steps would be required:

1) Increase \texttt{ArmLib_X_NUM_ARM_TYPES} by 1, and add a \texttt{#define} for the new arm (\texttt{#define ArmLib_X_Armytype_**** n}). These symbols are defined in \texttt{armlib_c.h}.

2) Create an additional entry in \texttt{ArmLib_X_typeinfo} for the new arm.

3) Create a file containing the device-specific routines as described above. The necessary routines are the ones which are called from \texttt{server.c}. The existing routines for the ARM, Phantom and Sarcos arm can be used as models.

4) Many of the armlib files (\texttt{server.c} in particular) have some \texttt{#ifdef}'s to isolate additional dependencies, most of which are not strictly device-related but are instead server operating system related. These \texttt{#ifdef}'s would have to be adapted for use with the new device and/or server operating system.

5) Some new elements in the \texttt{ArmLib_X_typeinfo} array structure might need to be added, to cover device-dependencies which we didn't think of in the initial arm library design. Some additional code might have to be added to other parts of the arm library as well. If we did our job well, the work in this step should be minimal.

\textbf{Separation of Communications Routines}
In designing the arm library, we made a concerted effort to isolate the client-server communications routines from the rest of the code and provide the capability to use multiple types of client-server communication. All communications calls go to routines in `com.c`. These routines "dispatch" the call to the appropriate communication-medium specific routine. In theory, each arm type can use a different means of client-server communication.

We initially supported both a TCP-only, and TCP/UDP mix for communications. The TCP-only routines have fallen into disrepair, especially with the addition of asynchronous communications, which can't be easily supported by the TCP-only communications medium.

The biggest advantage of this careful isolation of communications routines is that it should be fairly easy to add a new method of communication. The one that we have most specifically in mind is shared-memory communication, which could be used if the client and server halves of the library both run on the same machine.

Steps for adding a new communications medium:

1) Write the necessary routines, modelling them along the lines of those in `com_mix.c`, `scom_mix.c`, and `ccom_mix.c`.

2) Modify `com.c` so that it calls these new routines when appropriate.
XI. Notes, Bugs, etc.

1) Position prediction (simple or in conjunction with Kalman filtering) is not currently implemented, although the hooks for it are present throughout the code and API.

2) The TCP-only communications package probably no longer functions properly. It did at one time, but we did not maintain it as development progressed. It most certainly does not support the asychronous read modes.

3) Velocity is not currently supported fully — the ARM server doesn't calibrate its velocity data, and the general code doesn't compute the velocities. Also, the handling of the velocity/novelty flag isn't really correct in the server code. Finally, was I correct in documenting OrientVel?

4) I don't believe that the velocity is scaled when the arm is opened in scaled mode. The velocity should probably be scaled along with the position. Need to think about this.

5) Joint angles aren't sufficient - eg STM code. Need to either
   a) Transmit entire 4x4 matrix for each joint, or a subset of this matrix
   b) Have a way of fetching D-H info, then computing 4x4 matrix on client.
   In any case, the server doesn't even fully support joint angles now.

6) The "mix" communications mode uses UDP for some stuff. It assumes that UDP packets are delivered in order, which is not guaranteed by the UDP protocol. On a LAN, such as Ethernet, this shouldn't be a problem, but the code should probably eventually be fixed so that it doesn't depend on in-order delivery.

7) The force output of the Sarcos arm is currently clipped to a value below the maximum which is deliverable by the hardware. This was done so that applications such as docker, which request unreasonably large forces, don't hurt the user. The forces are clipped in a manner which preserves their direction. In the long run, a better solution needs to be found for this issue, since hard surfaces require large forces for short periods of time.

8) Large, sudden forces on the Sarcos arm would probably cause the safety system to trip out, so something will have to be done about this problem before good hard surfaces can be achieved.

9) The code for handling static friction on surfaces is currently somewhat broken. The units for the kS, dSnap, kStick parameters don't make sense. We will clean up this code and clarify the usage of these parameters in a future version of the library. In the meantime, very acceptable results can be achieved by just fiddling with the parameter values.
Appendix A — Phantom Joint Axes

The diagram below gives the Denavit-Hartenberg coordinate frames for the Phantom. These are the coordinate frames used internally by the arm library.
Note: 1) Joint rotations are about the Z0-Z2 axes. Sign of rotation is determined by right-hand rule.
2) Xt, Yt, Zt give the tool frame for tool #0 (NOT #1). They are not shown in the top view to avoid clutter.
3) Measurements were made at UNC and are not "official" dimensions.

SIDE VIEW — First two joint angles at 0°, last one at 90°

TOP VIEW — First two joint angles at 0°, last one at 90°
Appendix B — Additional Phantom Server Information

Motor Overload Protection

The peak power output of the phantom motors is substantially greater than the sustainable continuous power output. If large forces are requested for several seconds then it is possible to burn out the motors. In order to protect the motors against this danger, the phantom server has code which periodically runs to compute an estimate of the motor temperatures. If the estimated temperature is too high then the force output will be cut off until the motor temperature drops to an acceptable level.

Thus, the application writer should not need to worry about damaging the phantom motors - the server code should be able to protect them. (Note that we at UNC take NO RESPONSIBILITY for any failure of this code to protect your motors! However, that's what we use at UNC for our phantom, and it seems to work fine).

Maximum Force Clipping

The phantom server has the capability to limit the maximum output force. If it does so it will reduce all forces by the same factor, so that force direction is preserved. The server will print "X"s on the PC console to indicate that such a force reduction is in progress. The purpose of this capability is to reduce the likelihood that a sudden force will jerk the phantom loose from the user's finger, potentially hurting the user or damaging the device. Of course, when this capability is enabled hard surface quality may be somewhat reduced.