Project Report
ATC-284

WSP Utility Libraries

O. J. Newell

23 October 2000

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

Prepared for the Federal Aviation Administration,
Washington, D.C. 20591

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	23 October 2000	Project Report

4. TITLE AND SUBTITLE

WSP Utility Libraries

5. FUNDING NUMBERS

C — F19628-00-C-0002

6. AUTHOR(S)

O. Newell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Lincoln Laboratory, MIT
244 Wood Street
Lexington, MA 02420-9108

8. PERFORMING ORGANIZATION REPORT NUMBER

ATC-284

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of Transportation
Federal Aviation Administration
AND-420
800 Independence Ave., S.W.
Washington, DC 20591

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

This report is based upon studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, under Air Force contract F19628-00-C-0002.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

This document is available to the public through the National Technical Information Service, Springfield, VA 22161

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

The ASR-9 Weather Systems Processor (WSP) augments the weather detection capability of existing ASR-9 radars to include low-level wind shear warnings, storm cell tracking and prediction, and improved immunity to false weather echoes due to anomalous propagation (AP). To economically develop and field an operational system at the 34 WSP sites, the FAA is pursuing a strategy that leverages the software written during the 10-year R&D phase of the project. To that end, the software developed at Lincoln Laboratory has been 'hardened' to ensure reliable, continuous operation, and has been ported to a 'Phase II' prototype built around the latest generation of COTS hardware.

A significant number of the hardened software modules are being used in the production version of the WSP with only minor modifications. Included as part of the software are a number of lower-level utility libraries to provide basic services such as memory management and network communications. This document provides a detailed description of these common utility libraries.

14. SUBJECT TERMS

Airport Surveillance Radar	Radar data
Wind Shear	Microburst
Gust Front	

15. NUMBER OF PAGES
60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unclassified

ABSTRACT

The ASR-9 Weather Systems Processor (WSP) augments the weather detection capability of existing ASR-9 radars to include low-level wind shear warnings, storm cell tracking and prediction, and improved immunity to false weather echoes due to anomalous propagation (AP). To economically develop and field an operational system at the 34 WSP sites, the FAA is pursuing a strategy that leverages the software written during the 10-year R&D phase of the project. To that end, the software developed at Lincoln Laboratory has been 'hardened' to ensure reliable, continuous operation, and has been ported to a 'Phase II' prototype built around the latest generation of COTS hardware.

A significant number of the hardened software modules are being used in the production version of the WSP with only minor modifications. Included as part of the software are a number of lower-level utility libraries to provide basic services such as memory management and network communication. This document provides a detailed description of these common utility libraries.

TABLE OF CONTENTS

Abstract	iii
List Of Illustrations	vii
1. INTRODUCTION	1
2. CORE UTILITY LIBRARIES	3
2.1 Memory Management	3
2.2 Linked Lists	5
2.3 Time Handling	7
2.4 Message Logging	7
3. SERVER-CLIENT NETWORK COMMUNICATIONS LIBRARY	11
3.1 Server Client Sample Programs	13
3.2 Message Transport Example	15
3.3 SCLite Implementation	18
APPENDIX A: CORE UTILITY LIBRARY REFERENCE	23
A.1 Memory Management Library Reference	23
A.2 Linked List Reference	27
A.3 Time Class Reference	32
A.4 Message Logging Library Reference	34
APPENDIX B: SERVER CLIENT NETWORKING LIBRARY REFERENCE	39
GLOSSARY	51
REFERENCES	53

LIST OF ILLUSTRATIONS

Figure No. **Page**

1. Server-Client Transport Modes 11
2. Server-Client Communications Layers 12
3. TCP Implementation of Server-Client Data Transport Mode 19
4. Server-Client UDP Implementation 20
5. TCP Implementation of Server-Client Message Transport Mode 21

1. INTRODUCTION

The ASR-9 Weather Systems Processor (WSP) augments the weather detection capability of existing ASR-9 radars to include low-level wind shear warnings, storm cell tracking and prediction, and improved immunity to false weather echoes due to anomalous propagation (AP). To economically develop and field an operational system at the 34 WSP sites, the FAA is pursuing a strategy that leverages the software written during the 10-year R&D phase of the project. To that end, the software developed at Lincoln Laboratory has been 'hardened' to ensure reliable, continuous operation, and has been ported to a 'Phase II' prototype built around the latest generation of COTS hardware.

A significant number of the hardened software modules are being used in the production version of the WSP with only minor modifications. Included as part of the software are a number of lower-level utility libraries that are shared by many of the higher-level weather detection algorithms. These libraries supply basic functionality such as linked list and time manipulation routines, as well as more advanced facilities such as a custom, high-performance memory manager. A comprehensive network communications facility is also provided.

The software described here stands on its own, and knowledge of the WSP system as a whole, while beneficial, is not assumed. Interested readers should refer to [1] for a comprehensive overview of the WSP software environment.

2. CORE UTILITY LIBRARIES

2.1 MEMORY MANAGEMENT

The widespread use of dynamic memory allocation in a real-time application can sometimes be problematic. The basic problem is that the system-supplied memory allocation routines (*malloc(), free()*) are not normally very efficient, and constant allocation/deallocation of memory blocks can lead to fragmentation of the memory heap and eventual exhaustion of the memory resource. One solution is to largely avoid dynamic memory allocation following system startup, allocating enough memory up-front to handle the maximum expected system loading. While this strategy can work fairly well for simpler applications, it tends to break down in larger, more complex applications such as the WSP's weather detection algorithms. In the more complex cases, the common use of a large number of temporary working buffers for images and other data simply makes preallocation of all necessary working arrays impractical with respect to both resource efficiency and programming convenience.

Studies of memory usage by the applications running in the WSP system have shown that the size distribution of allocated memory blocks tends to be quantized as opposed to continuous. In other words, the applications tend to use blocks of memory of certain sizes, allocating and freeing the memory blocks over and over again. This fact can be exploited by a custom memory manager that takes advantage of the quantized nature of allocation requests and maintains multiple memory pools, each pool consisting of fixed-size blocks that are used to handle requests for a distinct size range. The use of fixed block sizes within each individual memory pool guarantees that no fragmentation occurs within each pool. Each pool is organized as a simple stack of free memory blocks, resulting in fast and predicable response times for allocations and deallocations.

The number and size distribution of the separate memory pools affects the overall efficiency of the memory manager. Use of a small step size for successive pools (e.g. 32 bytes) limits the number of wasted bytes for each allocation to 31 bytes. On the other hand, using such a small step size over the whole range of expected request sizes (0-512K) results in an unacceptably high number of separate pools from a pool management standpoint. To address this issue, the WSP memory manager splits the pools up into 'small' and 'large' categories. The 'small' category covers requests from 0-4096 bytes, and uses a step size of 32 bytes. This results in efficient use of memory when large numbers of relatively small objects are allocated. The 'large' category covers request from 4096-512Kbytes, and uses a step size of 1024 bytes, keeping the number of separate pools required to cover the desired size range to a manageable number (508). Note that the category boundaries and step sizes are currently hard coded, but could easily be set up as runtime parameters should the memory manager be used in another application with differing allocation requirements.

Use of the memory manager in a C program is identical to the use of the normal *malloc()* and *free()* routines, with the exception the custom memory manager routines being called *Malloc()* and *Free()* to differentiate them from the system-supplied routines. Use of the memory manager in a C++ program is even more transparent, since the linking in of the custom memory manager automatically overrides the system-supplied *new/delete* routines. Additional functions are provided in the library to support analysis of memory usage and debugging. The *MemStats()* function prints out the number of allocated blocks for each memory pool that is in use, as well as a summary of total memory usage. The *MemStatsDiff()* function is similar, but prints out the differences in memory utilization between multiple calls to the function, allowing a programmer to instrument a programmer and isolate memory-related problems. Lastly, the *MemExit()* function frees up all memory being used by the custom memory manger back to the system. This is useful when using a third-party memory analysis tool (such as Purify) to track down memory leaks, since such tools typically expect all memory to be explicitly freed prior to program termination.

The following example illustrates the use of the above functions.

```
#include <Mem.h>

int main( int argc, char *argv[] )
{
  char *buf1;
  char *buf2;
  char *buf3;

  /* Allocate three buffers of varying sizes */
  buf1 = Malloc(32);
  buf2 = Malloc(2048);
  buf3 = Malloc(8192);

  /* Print out summary usage statistics */
  MemStats( stdout );

  /* Initial call to MemStatsDiff() (saves current usage state) */
  MemStatsDiff( stdout );

  /* Free up one of the buffers and print out new memory usage changes */
  Free( buf2 );
  MemStatsDiff( stdout );

  /* Free up all memory cached by custom memory manager back to system */
  MemExit();

  exit(0);
}
```

When compiled and run, the program produces the following memory usage output.

```
MemStats:
 Blocksize   Total     Used     Free
 ---------------------------------------------------
    32         1         1        0
    2048       1         1        0
    8192       1         1        0

 Total Heap Size (Never DEcreases):  10272  Used: 10272  Free: 0

MemStatsDiff:              OLD                          NEW
 Blocksize   Total     Used     Free     Total     Used     Free
 ---------------------------------------------------------------
    32         0         0        0        1         1        0
    2048       0         0        0        1         1        0
    8192       0         0        0        1         1        0

 Total Heap Size (Never DEcreases):  10272  Used: 10272  Free: 0

MemStatsDiff:              OLD                          NEW
 Blocksize   Total     Used     Free     Total     Used     Free
 ---------------------------------------------------------------
    2048       1         1        0        1         0        1

 Total Heap Size (Never DEcreases):  10272  Used: 8224  Free: 2048
```

See appendix section A.1 for detailed description of all functions included with the memory management library.

2.2 LINKED LISTS

There are two types of list classes provided by the utilities library, an intrusive linked list class and a non-intrusive linked list class. The intrusive linked list class requires each stored object to be defined with a linked list pointer structure as it's first element. The intrusive class is intended for cases where speed is important, and each item may reside on only a single list at any given time. The non-intrusive list, as it's name suggests, does not require the objects themselves to contain any linked-list related fields, as the linked list pointer information is managed using separate, dynamically allocated linked list node structures. This allows for more flexibility (objects can reside on multiple lists), but has somewhat slower performance. In general, unless large numbers of objects must be visited very rapidly, the non-intrusive list is a better choice, and is the list class used most frequently by the WSP software.

The following two examples illustrate the use of the intrusive and non-intrusive linked list classes.

```
//
// Test program for intrusive linked list class
//

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#include <LLDList.h>

// Test object for list insertion.  Must have list link as first  element of structure.

typedef struct
{
  LLDListLink link;
  int         value;
} TestObj;


int main( int argc, char *argv[] )
{
  int i;
  LLDList list;
  TestObj  objArray[10];

  // Put 10 items on the list
  for( i = 0 ; i < 10 ; i++ )
  {
    objArray[i].value = i;
    LLDList.PutTail( &objArray[i] );
  }

  // Iterate through the list, printing out each entry

  TestObj *obj = (TestObj *)LLDList.FindFirst();

  while( obj != NULL )
```

```
  {
   printf("List Elem %d: Addr: 0x%x  Value: %d\n", n, (unsigned int)obj,
        obj->value );
   n++;
   obj = (TestObj *)LLDList.FindNext( obj );
  }
}


//
//  Test program for intrusive linked list class
//
#include <LLDList.h>

//
// Test object for list insertion.  Doesn't require any linked list
// pointers when a non-intrusive list is used
//

typedef struct
{
 int        value;
} TestObj;


int main( int argc, char *argv[] )
{
 int i;
 LLNIDList      list;
 LLNIDListCursor listCursor;
 TestObj        objArray[10];

 // Put 10 items on the list
 for( i = 0 ; i < 10 ; i++ )
 {
  objArray[i].value = i;
  LLNIDList.PutTail( &objArray[i] );
 }

 // Iterate through the list, printing out each entry.  When a non-intrusive
 // linked list is used, objects may reside on multiple lists, so a separate
 // list cursor is used to maintain the list position

 TestObj *obj = (TestObj *)LLDList.FindFirst( cursor );

 while( obj != NULL )
 {
  printf("List Elem %d: Addr: 0x%x  Value: %d\n", n, (unsigned int)obj,
       obj->value );
  n++;
  obj = (TestObj *)LLDList.FindNext( cursor );
 }

}
```

See appendix section A.2 for a detailed description of all functions provided by the linked list classes.

2.3 TIME HANDLING

Time information is used extensively throughout the WSP software. The set of most frequently used time manipulation routines (time differencing, adding, etc...) have been encapsulated in a C++ class for programming convenience. The class supports overloaded versions of the addition and subtraction operators to increase the readability of time manipulation code, as well as an overloaded print method for use with the C++ *cout* operator. The example below illustrates the straightforward use of the class to compute the difference between two times and print the results.

```
//
// Time class test program
//

#include <LLTime.h>

int main( int argc, char *argv[] )
{
 // Construct two time objects, initialized to be 10 seconds apart
 LLTime t1( 4, 1, 1997, 10, 20, 30 ); //  4/1/97-10:20:30
 LLTime t2( 4, 1, 1997, 10, 20, 40 ); //  4/1/97-10:20:40

 // Print out the two times
 cout << "t1 = " << t1 << " t2 = " << t2 << endl;

 // Compute the difference in seconds and print it
 int timeDiff = t2 - t1 ;
 cout << "t2 - t1 = " << timeDiff << " seconds " << endl;

 // Add 10 seconds to t1 and print result
 t1 += 10;
 cout << "t1 += 10 = " << t1 << endl;

}
```

When compiled and run, the program produces the following memory usage output.

```
t1 = 04/01/1997-10:20:30 t2 = 04/01/1997-10:20:40
t2 - t1 = 10 seconds
t1 += 10 = 04/01/1997-10:20:40
```

See appendix section A.3 for a complete description of all functionality provided by the time class.

2.4 MESSAGE LOGGING

The logging of diagnostic and error messages is a common requirement of most medium to large scale software systems, especially during the software development and debugging phase. The messages can typically be split into four classes, informational, warning, error, and debug. Depending on circumstance, it is often desirable to have flexibility with regard to whether the output is directed to a terminal window, a file, or simply discarded. The UNIX ***syslog*** facility provides most of the necessary features, but is somewhat limited in its ability to support the multiple separate logs required by the WSP software. The WSP message logging library, modelled after the ***syslog*** facility, provides the required functionality.

From an algorithm's point of view, the facility is quite simple. A call such as:

```
Log( LOG_INFO, "Info msg %d\n", 1 );
```

outputs the following message to the current logging destination:

```
Nov 18 20:30:46  [INFO]  Info msg 1
```

All messages are prefixed with a time tag, and the message 'class', which corresponds to the first argument passed to the Log() function. Available basic message classes are **LOG_INFO, LOG_WARN, LOG_ERR, and LOG_DBG.** In addition to the default debug level (1), 3 additional levels of debug are supported (**LOG_DBG2, LOG_DBG3, LOG_DBG4**) to allow for finer control of the amount of output produced by a running program. Debug messages can be enabled on a per file or per function basis using the logging configuration file. Other features of the logging implementation include selective output to stdout/stderr and/or disk files, fixed limits on log file size, and automatic creation of backup log files across multiple program runs and/or multiple days.

The logging behavior is controlled using a configuration file, normally read once at program startup. An example configuration file is shown below: Note that the configuration file supports UNIX shell-style comments (leading '#').

```
#==========================================================================
#
# File:  log.conf
#
# Logging facility configuration file.
#
#==========================================================================


# Output LOG_INFO message to file 'testlog.YYMMDD-HHMMSS' AND stdout
info    testlog.%T,stdout

# All other message just go to log file.  Files can be different for each message type, but are
# typically set to the same file.
warn    testlog.%T
err     testlog.%T
debug   testlog.%T


#==========================================================================
# stdout/stderr can be redirected to a logging file (including one of
# the ones specified above).
#==========================================================================
#

stdout  testlog.%T
stderr  testlog.%T


#==========================================================================
#
# Debug message control.  Allows specification of debug level and scope
#
# Debugging level can range from 1 to 4, with 4 being the most detailed
```

```
# (more output). The default level is 1. Debug scope can range from a single function
# to all files in the application.

#==========================================================================

debuglevel      2

debugEnableFile     logtest1.C

#==========================================================================
#
# To save backup copies of logfiles across multiple program runs or
# multiple days, specify a non-zero value for numBackups
#
#==========================================================================

numBackups      2

#==========================================================================
#
# Daily backup setting. Setting this value to 'true' or 'yes' allows
# separate logfiles to be maintained for each 24-hour period.
#
#==========================================================================

dailyBackup true

#==========================================================================
#
# Max history size in bytes. If logging output exceeds this amount,
# the current file is made into a backup (number of backups controlled
# by numBackups parameter) and a new file is started.
#
#==========================================================================

maxHistorySize   100000
```

The following sample program illustrates the initialization and use of the logging package. Assuming the configuration file shown above is used, the output of the program will be written to stdout and the disk file 'testlog.YYMMDD-HHMMSS'.

```
#include <stdio.h>

#include <Log.h>

int main( int argc, char *argv[] )
{
 int i;

 if( LogOpen( "log.conf" ) < 0 )
 {
  fprintf( stderr, "Error opening log\n");
  exit(-1);
 }
```

```
    Log( LOG_INFO, "Hello %d (info)\n", i);

    Log( LOG_WARN, "Hello (warn)\n");
    Log( LOG_ERR, "Hello (err)\n");

    Log( LOG_DBG2, "Hello (dbg2)\n");

    printf("normal printf\n");

    LogClose();

}
```

Program output (disk file and stdout):

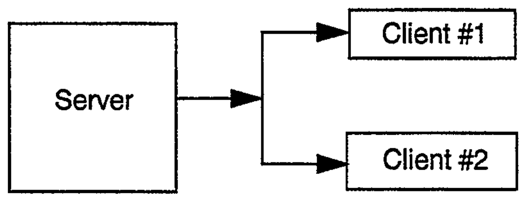
```
Dec 09 13:00:31 [INFO] Hello 1 (info)
Dec 09 13:00:31 [WARN] Hello (warn)
Dec 09 13:00:31 [ERR] Hello (err)
Dec 09 13:00:31 [DBG] Hello (dbg2)
normal printf
```

See appendix section A.4 for a detailed description of the functionality provided by the logging library.

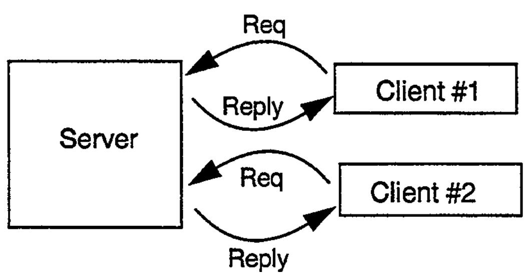
3. SERVER-CLIENT NETWORK COMMUNICATIONS LIBRARY

The server-client communications library (*SCLite)* provides a high-level set of functions for inter-process communication using UNIX sockets. Using SCLite, processes may communicate with each other using either TCP or UDP sockets while remaining largely isolated from low-level details such as read/write time-outs due to a network outage.

Two major modes of server-client communication are supported. The data transport mode (Figure 1.a) allows a process to act as a data server to one or more data clients. This is by far the most common usage in the context of Lincoln Laboratory weather algorithm software. The message transport mode (Figure 1.b) represents the more traditional usage of the term server-client, where a server process listens for requests from one or more clients, and replies to each request individually.



a) Server-Client Data Transport Mode



b) Server-Client Message Transport Mode

Figure 1. Server-Client Transport Modes

SCLite currently supports two underlying communications protocols. The TCP protocol is normally used for low-to-medium bandwidth connections where guaranteed delivery of each data packet is critical. TCP, a connection- based protocol, requires a separate data transmission to each connected client. For higher bandwidth connections (sustained data rate > 100K/sec.), the connectionless UDP protocol is supported as a lower-overhead (albeit less reliable) alternative, since it supports true Ethernet broadcasts and does not require a separate data transmission for each connected client. Note that the UDP protocol is only supported when using the package in the data transport mode, since the message transport mode relies on a reliable connection. The two protocols are implemented using a layered approach, illustrated in Figure 2.

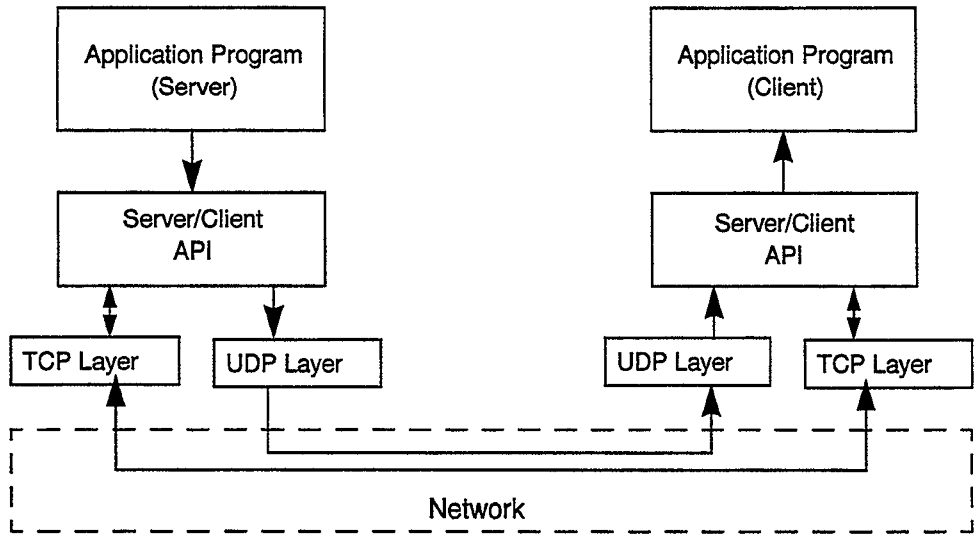


Figure 2. *Server-Client Communications Layers*

A key element of the server-client package is the *configuration* file -- an ASCII file containing a list of stream names with associated protocol information. The file allows for the protocol and port information for a given data or control streams to be reconfigured without having to modify the processes making use of the stream (other than restarting them with the new configuration file). A sample configuration file is shown below:

```
#
#   Sample server-client configuration file
#
#   Format of each entry:
#
#   <streamName> <serverHost> <protocol> <port> [protocolOptions]
#
#   Field definitions:
#
#   streamName     ASCII name for the stream. 32 characters max.
#
#   serverHost     Hostname of server. The reserved word 'local' specifies that the server
#                  is running on the local host.
#
#   protocol       Protocol for service. Available protocols are TCP and UDP.
#
#   port           TCP or UDP port number for service.
#
#  [protocolOptions]
#
#   -queueSize <size>Queue size in bytes for server (TCP protocol) or client (UDP protocol)
#
# Stream entry for transmitting data from TCP port 6661 on host 'frederick' to client applications
```

```
#
# Stream entry for transmitting data on TCP port 6663 on local host to client applications running on
#  local host
#
stream2    local        TCP    6663
#
# Stream entry for transmitting data on UDP port 6664 on frederick to client applications running on
#  frederick or other LAN-accessible hosts.
#
stream3    frederick    UDP    6663
```

The configuration file is specified when the stream is opened in one of two ways. If a filename is passed to the *SCOpen()* function, the specified filename will be used. Otherwise, the filename contained in the environment variable **SC_CONFIG_FILE** is used. Note that for backwards compatibility with an older version of the package, the environment variable **ALG_SERVICES** will be used if **SC_CONFIG_FILE** does not exist.

3.1 SERVER CLIENT SAMPLE PROGRAMS

Simple examples server-client data transport and message transport applications are shown below. These and other examples can be found in the sclite test directory (currently resides in /sw/share/sclite/test/src).

3.1.1 Data Transport Example

This example consists of a pair of programs. The data server, sc_write.c, sits in a loop and transmits a data packet filled with a ramp at one second intervals. The data client, sc_read.c, reads the incoming packets and prints out a message for every packet received. Note that no user-level connection management is required. Whenever the server program calls *SCSend()*, the SCLite package checks for new connections and cleans up after any connections that were broken since the last call to *SCSend()*. Similarly, whenever the client calls *SCRecv()*, a connection attempt is made if the server-client stream is not already in the connected state.

For simplicity, status/error handling is largely omitted from these examples. The example programs in the sclite/test/src directory illustrate typical status/error handling sequences.

Configuration file:

```
/*
 *   File: testConfig
 *
 *   Test program configuration file specifying parameters for single test stream
 */
teststream     localhost    TCP   8000
```

Server program:

```
/*
 *   File: sc_write.c
 *
 *   Test program to write a server_client data stream.
 */

#include <stdio.h>
```

```
#include <ServerClient.h>

void main( int argc, char *argv[] )
{
    int             i ;
    SCStream  *sp ;
    int             buf[BUFWORDS] ;

    if( ( sp = SCOpen( "teststream", SC_WRONLY, "testConfig" ) ) == NULL )
    {
            fprintf(stderr,"(scwrite) Failed to establish server service - exiting\n");
            exit( 2 );
    }

    /* Fill buffer with ramp */
    for( i=0 ; i < BUFWORDS ; i++ )
            buf[i] = i;

    for( i = 0 ; i < 100000 ; i++ )
    {
            sleep( 1 );
            printf("Sending rec %d, %d bytes\n", i, BUFWORDS*sizeof(int) );
            SCSend( sp, buf, BUFWORDS*sizeof(int), SC_NOTIMEOUT );
    }

    SCClose( sp );
}
```

Client program:

```
/*
 * File: scread.c
 *
 * Test program to read a server_client data stream.
 *
 */
#include <stdio.h>
#include <ServerClient.h>

#define MAX_WORDS (8192)

void main( int argc, char *argv[] )
{
    int             i, recLen;
    int             recCount = 0;
    int             buf[MAX_WORDS] ;
    SCStream  *sp;

    /*
     * Open stream in read-only mode (implies data client usage).  Default is to block on
     * calls to SCRecv().
     */
    if( ( sp = SCOpen( "teststream", SC_RDONLY, "testConfigFile" ) ) == NULL )
    {
            fprintf(stderr,"(scread) Failed to establish client service - exiting\n");
            exit( 2 );
```

```
    }

    while( 1 )
    {
        /*
        * Wait for incoming data, and perform connect/reconnect sequence as needed
        */
        if( (recLen = SCRecv( sp, buf, MAX_WORDS*sizeof(int), SC_NOTIMEOUT ) ) > 0 )
        {
            printf("Received rec %d:  %d bytes:", recCount, recLen);

            /* Print out 1st 4 words */
            printf("words[0-3]:  %d %d %d %d\n", buf[0], buf[1], buf[2], buf[3] );
            recCount++;
        }
        else if ( recLen == SC_NOCONNECTION )
        {
            printf("sc_read:  not connected to server - sleeping 1 second\n");
            sleep(1);
        }
        else
        {
            /*
             * When client configured as blocking, any return status other than a data packet
             * packet length or a NOCONNECTION indication is unexpected
             */
            printf("sc_read: Unexpected status from SCRecv (%d) \n", recLen );
            exit(-1);
        }
    }
}
```

3.2 MESSAGE TRANSPORT EXAMPLE

This example consists of a pair of programs, scserver.c and scclient.c. In the message transport mode, the server process waits for incoming messages from one or more clients, and, when a message is received, responds with a reply message. The configuration file is the same as used in the previous example. Note that the server program uses *SCRecvFrom()* and *SCSendTo()* in order to associate a unique client id with each data transfer.

This example also introduces the use of connect/disconnect handler functions. Since the connect/disconnect sequence is handled largely within the context of *SCSend()*/*SCRecv()* calls, the connect/disconnect handlers act can be used as notifiers to a server-client application that a new connection has occurred. A typical connect handler may output some fixed start-up information to the newly connected client. A typical disconnect handler may release any server resources that have been allocated to a particular client, such as an 'ownership' token of some shared resource.

Server program:

```
/*
 *   File: scserver.c
 *
 *   Simple server test program.  Waits for requests from clients and sends replies.
```

```
*/

#include <stdio.h>
#include <ServerClient.h>

#define RECV_BUF_WORDS   8192
#define REPLY_BUF_WORDS  1024
#define RECV_TIMEOUT     2000   /* 2 sec */

/*
 * Handlers for new connections or broken connections.  These versions only
 * output a simple print message
 */
void ConnectHandler( SCStream *sp, int clientId, void *arg )
{
    printf("ConnectHandler: clientId: %d  arg: 0x%x\n",
      clientId, (unsigned int)arg );
}

void DisconnectHandler( SCStream *sp, int clientId, void *arg )
{
    printf("DisconnectHandler: clientId: %d  arg: 0x%x\n",
      clientId, (unsigned int)arg );
}

void main( int argc, char *argv[] )
{
 int       i, recvBytes, clientId;
 int       recvBuf[RECV_BUF_WORDS] ;
 int       replyBuf[REPLY_BUF_WORDS] ;
 SCStream  *sp ;

 /* Open stream in r/w mode, specifying that this is the server side */
 if( ( sp = SCOpen( "teststream" , SC_RDWR | SC_SERVER,
             "testConfig" ) ) == NULL )
 {
  fprintf(stderr,"Error opening stream 'teststream'\n");
  exit( -1 );
 }

 SCSetConnectHandler( sp, ConnectHandler, (void *)0 );
 SCSetDisconnectHandler( sp, DisconnectHandler, (void *)0 );

 /*  Put a simple ramp in the reply buffer */
 for( i = 0 ; i < REPLY_BUF_WORDS ; i++ )
  replyBuf[i] = i;

 while( 1 )
 {
  /*
   * Wait for client request
   */
  if( (recvBytes = SCRecvFrom( sp, recvBuf, MAX_BUF_WORDS*sizeof(int),
                              RECV_TIMEOUT, &clientId )) < 0 )
  {
   printf("Error (%d)\n", bytes );
  }
  else if( bytes == 0 )
  {
```

```
    printf("Timeout\n" );
  }
  else
  {
    printf("Received msg of %d bytes, sending reply to client %d\n",
        recvBytes, clientId );
    SCSendTo( sp, replyBuf, REPLY_BUF_WORDS, SC_NOTIMEOUT, clientId );
  }
 }

 SCClose( sp );
}
```

Client program:

```
/*
 *   File: scclient.c
 *
 *   Simple client test program.  Sends requests to server processes
 *  (see scserver.c) and waits for replies.
 */

#include <stdio.h>
#include <ServerClient.h>

#define CONNECT_TIMEOUT     60000  /* 60 sec */
#define SEND_TIMEOUT         2000  /* 2 sec */
#define RECV_TIMEOUT         8000  /* 8 sec */
#define SEND_BUF_WORDS       1024
#define REPLY_BUF_MAX_WORDS  8192

void main( int argc, char *argv[] )
{
 int      i, bytes, bytesSent;
 int      sendBuf[SEND_BUF_WORDS] ;
 int      replyBuf[MAX_BUF_WORDS] ;
 SCStream  *sp ;

 /*  Put a simple ramp in the buffer */
 for( i = 0 ; i < SEND_BUF_WORDS ; i++ )
   sendBuf[i] = i;

 /* Open stream in r/w mode, specifying that this is the client side */
 if( ( sp = SCOpen( "teststream" , SC_RDWR I SC_CLIENT,
                "testConfig" ) ) == NULL )
 {
   fprintf(stderr,"Failed to establish server service - exiting\n");
   exit(-1);
 }

 /* Wait for initial connection (example of SCConnect() call usage) */
 while( ! (status = SCConnect( sp, CONNECT_TIMEOUT )) )
   printf("Waiting for connection\n");

 while( 1 )
 {
   if( (status = SCSend( sp, sendBuf, SEND_BUF_WORDS*sizeof(int), SEND_TIMEOUT )) > 0 )
```

```
    {
      /* Successfully sent message - wait for reply */
      if( (replyBytes = SCRecv( sp, replyBuf, REPLY_BUF_MAX_WORDS,  RECV_TIMEOUT )) > 0 )
        printf("Received reply of %d bytes \n", replyBytes );
      else if( replyBytes == 0 )
        printf("SCRecv: Timeout\n" );
      else if( replyBytes == SC_NOCONNECTION )
        printf("SCRecv: No connection\n");
      else
        printf("SCRecv: Unknown error, return status = (%d)\n", replyBytes );
    }
    else if( status == SC_NOCONNECTION )
    {
      printf("SCSend: No connection\n" );
      break;
    }
    else if( status == 0 )
    {
      printf("SCSend: Timeout\n" );
      break;
    }

    sleep( 1 );
  }

  SCClose( sp );
}
```

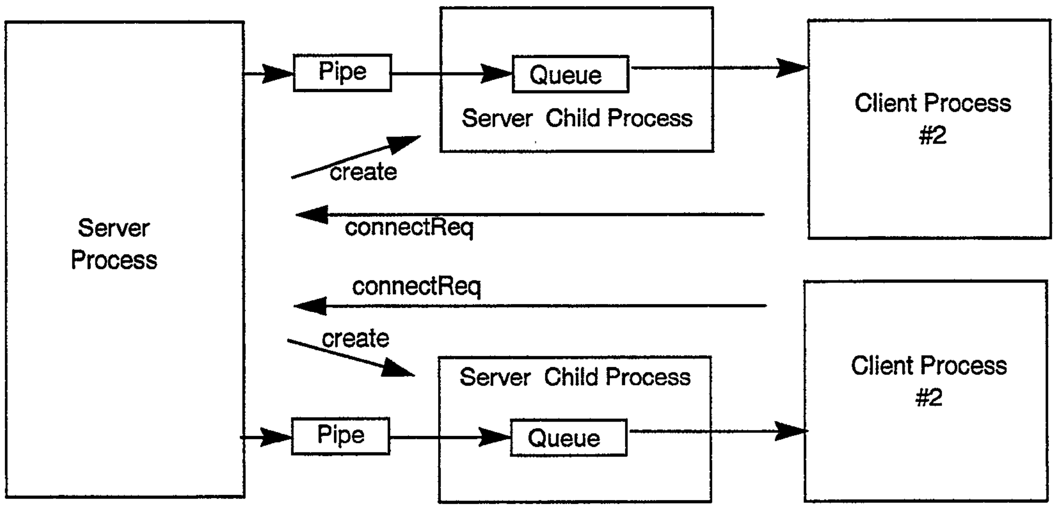
3.3 SCLITE IMPLEMENTATION

This section provides implementation details of the TCP and UDP data and message transport mechanisms. Knowledge regarding the implementation is of generally useful when design server-client programs using the SCLite package, especially when assessing design trade-offs.

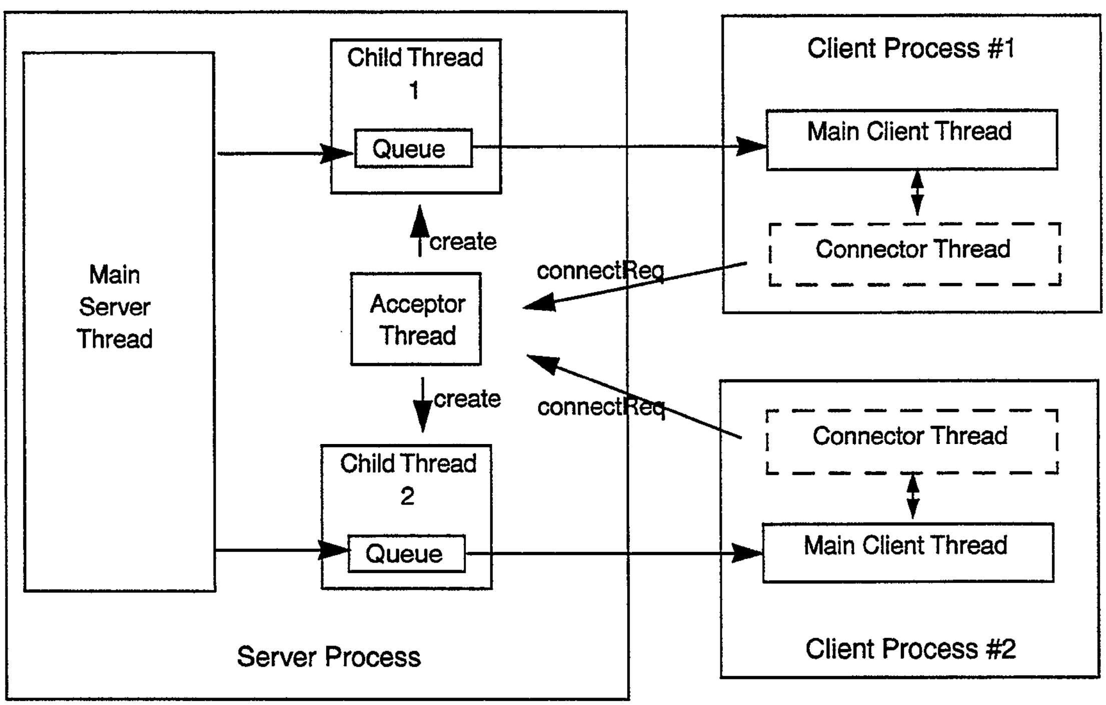
3.3.1 TCP Implementation - Data Transport Mode

When a TCP/IP-based data transport server is connected to multiple clients, each client may be connected via network paths of varying speeds. A client on the same physical machine, for example, utilizes the 'loopback' network interface (very fast), while another client may be connected via a 56 or even 19.2 Kilobaud dial-up PPP network connection. Simply transmitting data to each client in a serial fashion in such a configuration would result in an overall latency for each 'send' operation equal to the sum of all the transmission delays for each device. To avoid this latency, a separate child process (or thread) is created for each connection to provide a more concurrent serving of the data. To further reduce the latency from the perspective of the server process, each child process is equipped with a buffering mechanism. This allows the server to quickly resume its normal processing once the data has been transferred to the child process (fast).

Threaded and non-threaded variants of the TCP data transport protocol are provided by the SCLite package. The non-threaded version is the default, since the threaded version is not yet supported on all target platforms. When available, the threaded variant is preferred, since use of dedicated threads to perform the socket accept/connect sequence better insulates application programs from internal UNIX connection timeouts when attempting to connect/reconnect a socket during a network outage. The major functional blocks of the threaded and non-threaded implementations are shown in Figure 3.



a) Non-Threaded Implementation



b) Threaded Implementation

Figure 3. ***TCP Implementation of Server-Client Data Transport Mode***

3.3.2 TCP Implementation of Message Transport Mode

The message transport implementation is somewhat simpler than the data transport implementation, since there is no need for any buffering above and beyond that provided by the TCP network layer. The basic architecture is shown in Figure 5. As with the data transport implementation, both threaded and non-threaded variations are provided.

Server-client programs using this message transport facility will most commonly utilize a simple request/reply protocol, with a single reply being sent for every incoming request message. The implementation is, however, capable of supporting more complex schemes (such as allowing a server process to periodically send a message to one or more clients without receiving any incoming request). Basically, once the communications link is established, it possesses all the properties of the (underlying) full-duplex UNIX socket.

3.3.3 UDP Implementation of Data Transport Mode

The UDP-based data transport implementation utilizes the internet broadcast mechanism to allow multiple clients to 'listen' to a single data transmission on a single network. In this case, the assumption is made that the actual latency due to the transmission of each UDP packet is minimal, and no buffering is performed on the server side. Instead, buffering is performed on the client side, preventing the loss of data if a client is busy processing data when new data arrives at the network interface. Once again, a separate child process is used to implement the buffering mechanism, although shared memory is used for child-parent communications in place of a UNIX pipe for efficiency reasons (the code was inherited from an application requiring high bandwidth). The UDP design is illustrated in Figure 4.

The UDP protocol places a 1472 byte limit on the length of individual data packets. SCLite performs automatic packet fragmentation on the server side and defragmentation on the client side to allow for transparent transfers of larger packets. The implementation currently does not support detection/retransmission of missing packets. If missing packets are a significant problem, either the network itself should be redesigned to reduce them to an absolute minimum, or use of the TCP protocol should be considered.

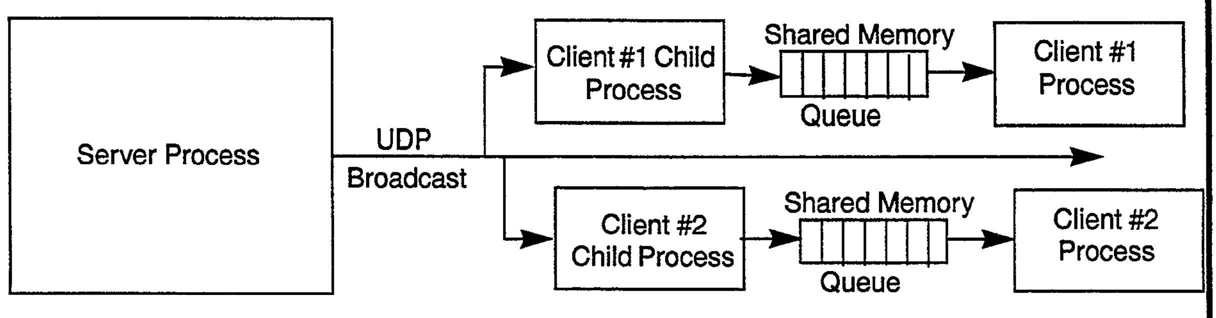
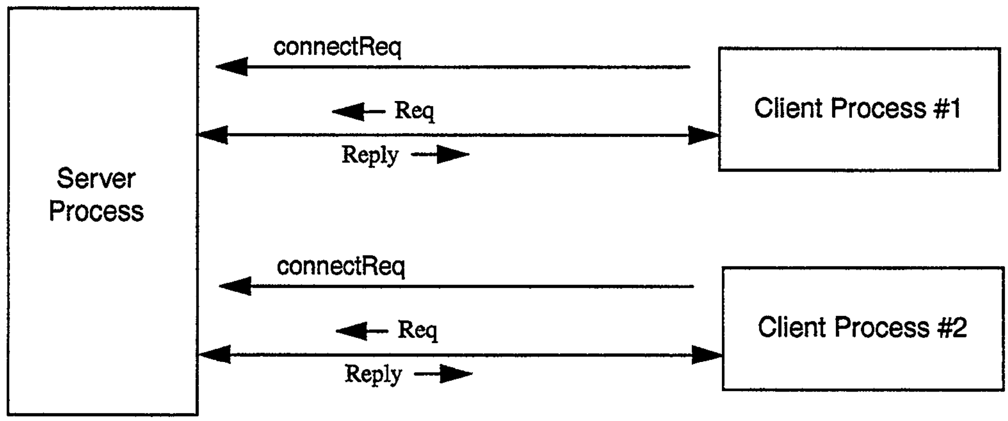
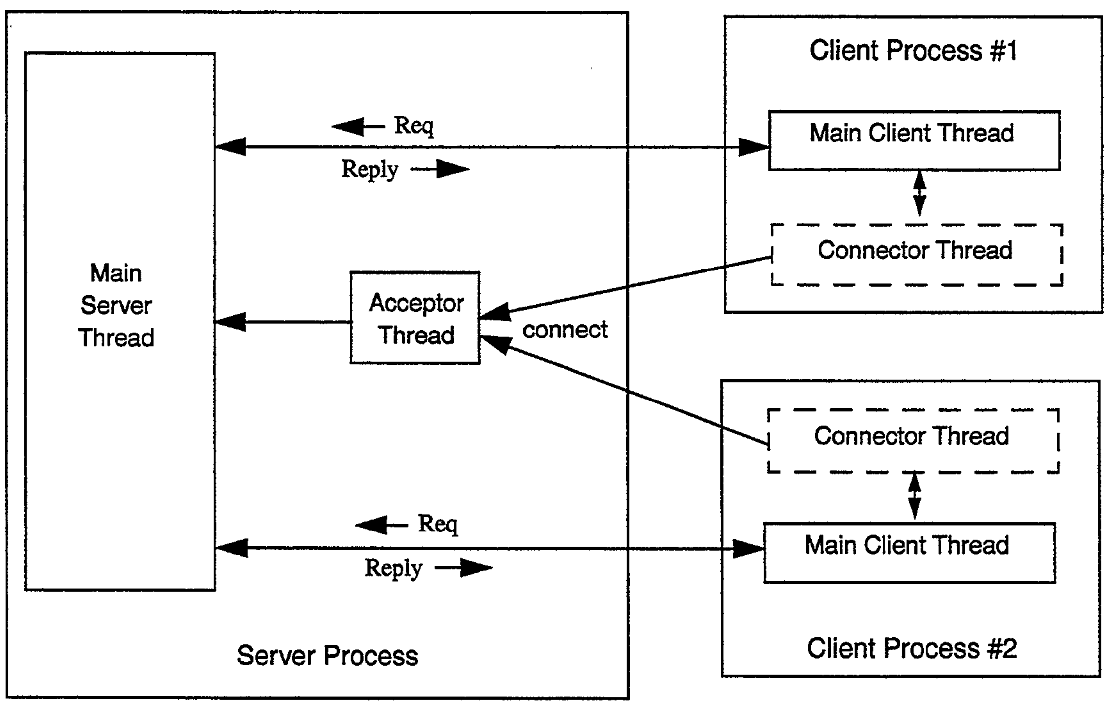


Figure 4. Server-Client UDP Implementation



b) Non-Threaded Implementation



b) Threaded Implementation

Figure 5. TCP Implementation of Server-Client Message Transport Mode

APPENDIX A
CORE UTILITY LIBRARY REFERENCE

A.1 MEMORY MANAGEMENT LIBRARY REFERENCE

This section provides detailed descriptions of the functions provided by the custom WSP memory manager. These functions are made available when an application is linked with the memory management library (-lmem).

Malloc()

Name

Malloc()
Custom memory manager wrapper for system malloc().

Synopsis

#include <Mem.h>

*void *Malloc(size_t sz);*

Description

Allocates a block of (no smaller than) *sz* bytes according to the WSP custom memory manger scheme. For example, if the small block size for the custom memory manger is 32 bytes, and a request for 50 bytes is made, then a block of 64 bytes will be returned. As described in the overview to the *mem* library, this approach is used to prevent memory fragmentation.

This is a 'fast' malloc which will simply return the first free block of the appropriate size from the custom memory manger's cache of blocks. If no blocks of the appropriate size are available, a new one will be allocated and returned via system *malloc*(). If the requested *sz* is larger than the maximum block size for the custom memory manager, the new block will be allocated via system *malloc*(). Note that these large blocks are 'unmanaged', i.e. the custom memory manage does not maintain a cache of these extremely large blocks; it is assumed that there will be few, if any, such requests over the lifetime of any of the WSP algorithms.

Returns

Pointer to the new memory block. If memory is exhausted, prints a message and calls *MemStats*() and *MemStatsDiff*() to report memory manager state.

See Also

MemStats(), MemStatsDiff()

Free()

Name

Free()
Custom memory manager wrapper for system free().

Synopsis

#include <Mem.h>

*void *Free(void *m);*

Description

Frees a block of memory according to the WSP custom memory manger scheme.

This is a 'fast' free which simply places the block on the appropriate stack of free (i.e. available for use) blocks, for later use by the memory manager. If the size of the block is greater than the maximum blocksize of the memory manager, it is explicitly freed by system *free*(). It is assumed that there will be few, if any, such large blocks needed over the lifetime of any of the WSP algorithms.

Returns

Nothing

Note

Note that via the WSP scheme, the block size is maintained internally in each block. This size is set to a key value upon freeing of the block. In this way, checks for multiple frees can be readily accomplished. If the block size is set to this key value, and Free() is then called for that block, it is probable that a double free is being performed on that block. In this case a message is printed and a segmentation fault is forced to allow for easier debugging.

MemStats()

Name

MemStats()
Utility for printing out custom memory manager statistics.

Synopsis

#include <Mem.h>

*void MemStats(FILE *fp);*

Description

Utility for printing out custom memory manager statistics. For each block size used by the manager, prints out the block size, total blocks allocated at that size, number of blocks at that size currently in use, and number of blocks at that size currently on the free stack. Note that the information is printed out only for block sizes currently in use (e.g. if no blocks of size 512 have been requested, no info for that size block will be printed). After printing this information, the total heap size that has been allocated at the time of the call to *MemStats*() is printed, followed by the total heap size currently in use, and finally the total heap size on the free stacks.

Returns

Nothing

See Also

MemStatsDiff(), which prints out the change in memory manager statistics since the last call to *MemStatsDiffer*().

MemStatsDiff()

Name

MemStatsDiff()

Utility for printing out custom memory manager statistics. Prints out the change in memory manager statistics since the last time MemStatsDiff() was called.

Synopsis

#include <Mem.h>

*void MemStatsDiff(FILE *fp);*

Description

Utility for printing out changes in custom memory manager statistics between subsequent calls to *MemStatsDiff*(). On the first call, this routine prints out the full set of memory manager statistics as printed by *MemStats*(). Memory statistics from that time are saved for comparison to the current statistics the next time *MemStatsDiff*() is called. On subseqent calls, statistics are only printed for the block sizes that have witnessed allocation/deallocation activity.

Returns

Nothing

MemExit()

Name

MemExit()

Function to free all unused memory chunks (allocated and maintained by the custom memory manager) back to operating system.

Synopsis

#include <Mem.h>

void MemExit(void);

Description

Function to free all unused memory chunks (allocated and maintained by the custom memory manager) back to operating system. This is useful when using tools like Purify that would report the memory as leaked if this wasn't done. It should be the VERY LAST call made before exiting.

Returns

Nothing

new()

Name

new()

Definition of 'new' which overrides system 'new'; allows WSP custom memory management to be used transparently from WSP C++ algorithms.

Synopsis

#include <Mem.h>

*void * operator new (size_t sz);*
*void * operator new [] (size_t sz);*

Description

Definition of 'new' which overrides system 'new'; allows WSP custom memory management to be used transparently from WSP C++ algorithms. Essentially just calls *Malloc*() from the custom memory library.

Returns

Pointer to the new memory block. If memory is exhausted, prints a message and exits.

See Also

Malloc() from the mem library.

delete()

Name

delete()
Definition of 'delete' which overrides system 'delete'; allows WSP custom memory management to be used transparently from WSP C++ algorithms.

Synopsis

#include <Mem.h>

*void operator delete (void *m);*
*void operator delete [] (void *m);*

Description

Definition of 'delete' which overrides system 'delete'; allows WSP custom memory management to be used transparently from WSP C++ algorithms. Essentially just calls *Free*() from the custom memory library.

See Also

Free() from the mem library.

A.2 LINKED LIST REFERENCE

This section provides detailed descriptions of the linked list classes provided with the WSP utility library. These classes are made available when an application is linked with the general utility library (-lllutil).

LLDList

Name

class LLDList

Synopsis

#include <LLDlist.h>

Description

Provides methods for managing doubly linked lists of application objects. Objects are required to start with a *LLDListLink* structure.

Constructors

LLDList()

Destructors

~LLDList()
Deletes just the list object (*LLDList*) itself. Does not delete nodes on the list. These must be cleaned up by the application.

Public member functions

*void *GetHead()*
Gets the first object on the list, while simultaneously removing it from the list. All relevant list constructs are updated to reflect the modified list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty.

*void PutHead(void *)*
Prepends an object to the head of the list, updating all relevant list constructs.

*void *GetTail()*
Gets the last object on the list, while simultaneously removing it from the list. All relevant list constructs are updated to reflect the modified list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty.

*void PutTail(void *)*
Appends an object to the end of the list, updating all relevant list constructs.

*void *FindFirst()*
Finds the first object on the list. This is the object located at the list head. Does not remove object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty.

*void *FindNext(void *obj)*

Finds the object on the list following the one pointed to by *obj*. Does not remove the object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if there are no objects following *obj* on the list (i.e. *obj* is at the list tail).

*void   *FindLast()*
Finds the last object on the list. This is the object located at the list tail. Does not remove object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty.

*void   *FindPrev(void *obj)*
Finds the object on the list preceding the one pointed to by *obj*. Does not remove the object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if there are no objects preceding *obj* on the list (i.e. *obj* is at the list head).

*void   *FindIndexed(short index)*
Finds an object on the list based on its index location. The index is referenced to the head of the list. The head is considered index zero. Does not remove the object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list has too few objects (i.e. list size < (index +1)).

*void   InsertBefore(void *next, void *obj)*
Insert an object *obj* before the object pointed to by *next*. If *next* is NULL, insert the object at the end (tail) of the list.

*void   InsertAfter(void *prev, void *obj)*
Insert an object *obj* after the object pointed to by *prev*. If *prev* is NULL, insert the object at the front (head) of the list.

*void   *Remove(void *obj)*
Removes the object pointed to by *obj* from the list. Note that *obj* is not deallocated; it is up to the application to do this. Returns A pointer to next object on list, or NULL if the object deleted was the last on the list.

int   NumElements()
Returns the number of elements on the list.

int   IsEmpty()
Determine if list is empty. If it is, return TRUE (1), otherwise return FALSE (0)

void   Reverse()
Reverses the order of objects on a list.

LLNIDList

Name

class LLNIDList

Synopsis

#include <LLNIDList.h>

Hierarchy

LLNIDListLink->LLNIDList

Description

Provides methods for managing non-intrusive (NI) doubly linked lists of application objects. Objects are not required to contain any particular linking structure. By contrast, the *LLDList* class requires that list objects begin with an *LLDListLink* structure.

Component Structures

```
typedef struct LLNIDListNodeStruct LLNIDListNode;

struct LLNIDListNodeStruct
{
        LLNIDListNode *prevFree;
        LLNIDListNode *next;
        LLNIDListNode *prev;
        void             *obj;    /* Pointer to object stored on list. */
};

typedef struct LLNIDListNodeStruct  LLNIDListLink;
```

nstructors

LLNIDList()

Destructors

~LLNIDList()

Assignment operators

LLNIDList& operator += (LLNIDList &rhs)

Concatenates the nodes of two *LLNIDLists* by appending the nodes of *rhs* onto the end of the list of nodes for *lhs*. The input *rhs* is unmodified.

Other operators

LLNIDList operator + (LLNIDList &lhs, LLNIDList &rhs)
Returns a new *LLNIDList* which is the oncatenation of *lhs* and *rhs*. The elementes of *lhs* are positioned first in the new list, followed by those of *rhs*. Note there are then multiple copies of pointers to the list nodes; the nodes themselves are not duplicated.

friend ostream& operator << (ostream& os, LLNIDList& list)
Output operator for *LLNIDList*. The number of nodes is first printed, then for each node, the node number followed by the address of the node.

Public member functions

int NumElements()
Returns the number of nodes in the *LLNIDList*.

*void *GetHead()*
Gets the first object on the list, while simultaneously removing it from the list. All relevant list constructs are updated to reflect the modified list. The object is

returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty.

*void PutHead(void *)*
Prepends an object to the head of the list, updating all relevant list constructs.

*void *GetTail()*
Gets the last object on the list, while simultaneously removing it from the list. All relevant list constructs are updated to reflect the modified list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty.

*void PutTail(void *)*
Appends an object to the end of the list, updating all relevant list constructs.

*void *FindFirst(LLNIDListCursor &cursor)*
Finds the first object on the list. This is the object located at the list head. Does not remove object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty. Also, input *cursor*, which is passed by reference, is set appropriately (pointing to the head of the list) for use with subsequent *FindNext*(), *FindPrev*(), etc. calls.

*void *FindNext(LLNIDListCursor &cursor)*
Finds the object on the list following the one pointed to by *cursor*. Does not remove the object from the list. Updates *cursor* to node found. Returns a pointer to the object following *cursor*, or NULL if *cursor* pointed to the last object on the list. The object is returned as a void pointer and must be cast to the correct type by the application.Updates *cursor* value to point to the object found.

*void *FindLast(LLNIDListCursor &cursor)*
Finds the last object on the list. This is the object located at the list tail. Does not remove object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list is empty. Also, input *cursor*, which is passed by reference, is set appropriately (pointing to the tail of the list) for use with subsequent *FindNext*(), *FindPrev*(), etc. calls.

*void *FindPrev(LLNIDListCursor &cursor)*
Finds the object on the list preceding the one pointed to by *cursor*. Does not remove the object from the list. Updates *cursor* to node found. Returns a pointer to the object preceding *cursor*, or NULL if *cursor* pointed to the first object on the list. The object is returned as a void pointer and must be cast to the correct type by the application.Updates *cursor* value to point to the object found.

*void *FindIndexed(LLNIDListCursor &cursor, int index)*
Finds an object on the list based on its index location. The index is referenced to the head of the list. The head is considered index zero. Does not remove the object from the list. The object is returned as a void pointer and must be cast to the correct type by the application. NULL is returned if the list has too few

objects (i.e. list size < (index +1)). Also *cursor* is updated to point to the object found.

*void InsertBefore(LLNIDListCursor &cursor, void *obj)*
Insert an object before the object pointed to by *cursor*. The *cursor* must point to a valid list node.

*void InsertAfter(LLNIDListCursor &cursor, void *obj)*
Insert an object after the object pointed to by cursor. The cursor must point to a valid list node.

*void *Remove(LLNIDListCursor &cursor)*
Delete the link for the object at the current *cursor* postion from the list. Returns the object whose link was just deleted and updates the *cursor* to the next object on the list. Returns NULL if the *cursor* is positioned at the end of the list (NULL). The object is returned as a void pointer and must be cast to the correct type by the application. Prior to calling this function, the *cursor* must have been assigned a valid value by performing a call to one of the Find()/Get() functions.

int IsEmpty()
Determine if list is empty. If it is, return TRUE (1), otherwise return FALSE (0)

void Clear()
Removes all objects from the list. Note that the nodes are deallocated via calls to *delete.*

A.3 TIME CLASS REFERENCE

This section provides detailed descriptions of the time class provided with the WSP utility library. These classes are made available when an application is linked with the general utility library (-lllutil).

LLTime

Name

class LLTime

Synopsis

#include <LLTime.h>

Hierarchy

LLTime

Description

Supports date/time calculations. All times packaged into *LLTime* objects are interpreted as GMT.

Constructors

LLTime()
Default constructor; returns the time of the birth of UNIX (1/1/1970 00:00:00).

LLTime(const LLTime &from)
Copy constructor; a new *LLTime* object is created with all fields (month, day, etc.) copied from the supplied *LLTime* object.

LLTime(const short month, const short day, const short year,
const short hour, const short min, const short sec)
Creates a new *LLTime* object with the specified month, day, etc.

Destructors

~LLTime()

Assignment operators

LLTime& operator = (const LLTime &)
Assign all fields of lhs *LLTime* object to those of rhs.

int operator == (const LLTime &)
Checks whether all fields of lhs *LLTime* object are equal to corresponding fields of rhs *LLTime* object.

int operator < (LLTime &)
Determines whether lhs *LLTime* object is chronologically before rhs *LLTime* object. Returns 1 if true, 0 otherwise.

int operator > (LLTime &)
Determines whether lhs *LLTime* object is chronologically after rhs *LLTime* object. Returns 1 if true, 0 otherwise.

int operator - (LLTime &) *// Difference in seconds, lhs - rhs*

LLTime operator + (int) // *Add 'int' seconds*
void operator += (int) // *Add 'int' seconds*

Access functions

The following functions return the indicated field of the *LLTime* object.

short getYear(void)
short getMonth(void)
short getDay(void)
short getHour(void)
short getMinute(void)
short getSecond(void)

time_t getSeconds(void)
Convert *LLTime* object to time since Epoch in seconds; returns this value.

int getJulianDay(void)
Returns the day of the year (0 through 365 or 366 for leap years) corresponding to the time indicated in the *LLTime* object. Accounts for leap years, etc.

The following functions set the indicated field of the LLTime object.
void setYear(const short y)
void setMonth(const short m)
void setDay(const short d)
void setHour(const short h)
void setMinute(const short m)
void setSecond(const short s)

// Set all fields at once.
void set(short mo, short d, short y, short h, short m, short s)

void setToGM(time_t t) // *Set using time in seconds since epoch*
void setToPresentGM(void) // *Set to current Greenwich Mean Time*

Public data members

short month
short day
short year
short hour
short minute
short second

Related global functions

friend ostream& operator << (ostream&, const LLTime &)
Prints out an *LLTime* object, in form year, month, day, hour, minute, second.

A.4 MESSAGE LOGGING LIBRARY REFERENCE

This section provides detailed descriptions of the message logging functions provided with the WSP utility library. These classes are made available when an application is linked with the general utility library (-lllutil).

LogOpen()

Name

LogOpen()
Open a message logging file.

Synopsis

#include <Log.h>

*int LogOpen(char *configFile)*

Description

Opens up one or more destinations for logging utility output; log outputs are as specified in the *configFile*. This function must be called prior to any calls to the *Log()* function.

A commented example of a valid configuration file is shown below. All valid options are discussed in the comments provided.

```
#================================================================
#
# File: log.conf
#
# Logging facility configuration file.
#
#
#================================================================

#================================================================
#
# Output streams for the main 4 logging classes. Output can be to
# a file and/or stdout/stderr. If both a file and stdout/stderr are used,
# specify like 'testlog,stdout' (no spaces allowed)
#
# The same file can be specified for multiple logging classes.
# To discard all messages for a given logging class, specify /dev/null
# as the output stream.
#
# A '%T' included anywhere in the filename will expand to the time
# that the log file is opened. The timestring looks like: 'YYMMDD-HHMMSS'
#
# In conjuction with the 'dailyBackup' and 'nBackups' options (see below),
```

```
# this feature can be used to create daily logfiles with meaningful names.
#
#=====================================================================

# Output LOG_INFO message to file 'testlog.YYMMDD-HHMMSS' AND stdout
info    testlog.%T,stdout

# All other message just go to log file. Files can be different for each message type, but
# are typically set to the same file.
warn    testlog.%T
err     testlog.%T
debug   testlog.%T


#=====================================================================
#
# stdout/stderr can be redirected to a logging file (including one of
# the ones specified above). This is handy if libraries external to
# an application make use of stdout/stderr (bypassing the Log() function
# library). As with the 4 message 'classes' described above, stdout/stderr
# may also be redirected to /dev/null
#=====================================================================
#

stdout  testlog.%T
stderr  testlog.%T


#=====================================================================
#
# Debug message control. Allows specification of debug level and scope
#
# Debugging level can range from 1 to 4, with 4 being the most detailed
# (more output). The default level is 1.
#
# If debugging enabled by specifying output stream other than /dev/null,
# debugging messages from all modules in the application will be output
# by default (global scope). The output can be made more selective using
# two commands (in combination or individually)
#
#    debugEnableFile <file> [startLine] [stopLine]
#
#    debugEnableFunction <functionName>
#
# If [startLine] and [stopLine] are not specified for the file version,
# debugging messages are enabled for the entire file. Up to 20 of each
# 'rule' can be specified at one time.
#
# The matching process is a simple substring match for both the
# <file> and <functionName> fields, so, for example, a function name of
```

```
# "Gust" would match any function prototype containing the word "Gust"
# (In the case of C++, even if the "Gust" string is part of one of the
# function arguments)
#
#=================================================================

debuglevel        2

debugEnableFile      logtest1.C

# Sample syntax to enable debugging for all member functions of class 'Dummy'
#debugEnableFunction  Dummy::


#=================================================================
#
# To save backup copies of logfiles across multiple program runs or
# multiple days, specify a non-zero value for numBackups
#
#=================================================================
numBackups        2

#=================================================================
#
# Daily backup setting.  Setting this value to 'true' or 'yes' allows
# separate logfiles to be maintained for each 24-hour period.  The
# current file becomes a backup file, which is kept around until the
# 'numBackups' value is exceeded.
#
# The default is to do the switchover at 00:00:00 local time. If the
# additional argument 'GMT' is supplied, the switchover will occur at
# 00:00:00 GMT
#
# If no 'dailyBackup' setting is specified (commented out or missing),
# creation of backup files is controlled solely by 'maxHistorySize'.
#
#=================================================================

dailyBackup true
#dailyBackup true GMT


#=================================================================
#
# Max history size in bytes.  If logging output exceeds this amount,
# the current file is made into a backup (number of backups controlled
# by above parameter) and a new file is started.
#
#=================================================================
```

maxHistorySize 100000

Returns

0 is log was successfully opened; -1 otherwise.

Log()

Name

Log()
Log a message.

Synopsis

#include <Log.h>

*int Log(int loggingClass, char *fmt, /* args*/ ...)*

Description

Writes a printf-style message to the message log, using the specified logging class. Valid values for *loggingClass* are LOG_INFO, LOG_WARN, LOG_ERR, LOG_DBG, LOG_DBG2, LOG_DBG3, and LOG_DBG4. The LOG_INFO class is intended for informational messages that are the result of normal program operation. The LOG_WARN class is intended for warning messages that are indicative of a non-fatal error condition. The LOG_ERR class is intended for serious errors which possibly require immediate attention. The four LOG_DBG classes are intended for debug messages of varying detail, ranging from LOG_DBG typically being used for high-level debugging messages (function entry/exit) and the other LOG_DBG<X> classes being used for messages providing successively greater detail. As described in the *LogOpen()* description, the output for the various classes can be individually controlled via the configuration file.

Note that the function prototype in Log.h does not match the function signature shown above. This is due to the fact the *loggingClass* argument is actually a concatenation of several arguments, and the LOG_<X> class definitions are actually macros that pass multiple arguments. This is done to transparently provide line number, file, and function name information to the logging function.

Returns

0 is log was successfully opened; -1 otherwise.

LogClose()

Name

LogClose()
Close a message logging file.

Synopsis

#include <Log.h>

void LogClose(void)

Description

Close a message logging file. Flushes all log output buffers.

Returns

Nothing

APPENDIX B
SERVER CLIENT NETWORKING LIBRARY REFERENCE

This appendix provides detailed descriptions of the functions provided with the WSP server-client network communications library. These functions are made available when an application is linked with the normal or multi-threaded variants of the library (-lsclite or -lscliteMT).

SCClose()

Name

SCClose()
Close a server-client stream.

Synopsis

#include <ServerClient.h>

*int SCClose(SCStream *sp);*

Arguments

sp	Pointer to open server-client stream.

Description

SCClose() closes down the specified server-client stream and frees up any associated resources.

Returns

Zero on success, or -1 if the stream pointer is invalid (not initialized properly or already closed).

See Also

SCOpen()

SCConnect()

Name

SCConnect()
(Re)connect a client.

Synopsis

#include <ServerClient.h>

int SCConnect(SCStream sp, int timeout);*

Arguments

sp	Pointer to server-client stream object.
timeout	Timeout value for connection attempt, in milliseconds.

Description

SCConnect() connects a client to a server. Calling this function is not strictly necessary, as each call to *SCRecv()* will attempt to (re)connect an unconnected client stream, but it can come in handy in certain situations.

Returns TRUE if the connection was established, or FALSE if timeout occurred.

See Also *SCOpen()*

SCOpen()

Name

SCOpen()
Open a server-client stream.

Synopsis

#include <ServerClient.h>

*SCStream *SCOpen(char *serviceEntry, int flags, char *configFile);*

Arguments

serviceEntry	The service entry can take on one of two forms. The most common form is a simple service name, such as 'datastream1'. If so specified, additional connection-related information (hostname, port number, etc...) is derived from the line in the server-client configuration file with the matching service name. Alternatively, the serviceEntry string may contain all the required connection information, in the same format as the configuration file. This can be useful when a program wants to dynamically generate it's own stream configuration. For example, a serviceEntry of "datastream1 juliet TCP 8900" specifies a stream name of 'datastream1' on host 'juliet' using the TCP protocol and port number 8900.
flags	Flags bitmask controlling stream r/w mode and other options. Flags values are described below.
SC_WRONLY	Data server mode. The stream is configured as a unidirectional output stream, serving data to one or more clients.
SC_RDONLY	Data client mode. The stream is configured as a unidirectional input stream, accepting data from a stream configured as a data server.
(SC_RDWR | SC_SERVER)	Message server mode. The stream is configured as a bidirectional message-oriented server connection, suitable for receiving messages from one or more message-oriented clients and transmitting a reply.
(SC_RDWR |	Message client mode. The stream is configured as a

SC_CLIENT)	bidirectional message-oriented client connection, suitable for sending messages from a message-oriented server.
SC_NONBLOCK	Non-blocking mode. Calls to *SCSend()* or *SCRecv()* will not-block if the send queue backs up or the receive queue is empty, but will instead return immediately with appropriate status.
SC_RAW	Raw IO Mode. Application 'sees' Server-Client headers. Useful for porting programs which used the old Server-Client package, where the protocol headers were always visible to the application.
configFile	Name of configuration file containing list of stream names with corresponding port numbers and protocols. If specified as NULL, the configuration file specified in environmental variable **SC_CONFIG_FILE** will be used. If the environment variable **SC_CONFIG_FILE** does not exist, a check is made for the existence of a second environment variable, **ALG_SERVICES** (to support an older version of the server-client library)

Description

SCOpen() opens a server-client stream for write-only, read-only, or read/write access. It should be noted that the stream is **not** in a connected state upon return from this routine. This is due to the dynamic nature of the connections. From the server's perspective, there may not be any clients ready to connect at startup, and it is not desirable to block in *SCOpen()* waiting for a client to request a connection. The converse is true for the case of a client -- the corresponding data server may not be running and it is desirable to avoid blocking in the *SCOpen()* call for any given stream since the process may be a client of other, existing, servers.
In general, the management of connections is handled internally to the *SCSend()/ SCSendTo() and SCRecv()/SCRecvFrom()* calls. During each call to *SCSend()*, the stream is checked to see if any new clients are requesting a connection. If so, the connection is made and added to the servers list of connected clients. During each call to *SCRecv()*, a check is made to see if the client is connected to a server, and if not, an attempt is made to (re)establish the connection. If the user *does* wish to block a program functioning as a client until a server responds, the *SCConnect()* call can be used.

Returns

Pointer to opened server-client stream, or NULL if open failed.

Environment Variables

SC_CONFIG_FILE (preferred) or **ALG_SERVICES** (legacy backward compatibility). One of these environment variables must point to a file containing a description of the server-client SCStream linkages used by the application.

See Also

SCClose()

SCRecv()

Name

SCRecv()
Receive data via a Server-Client stream.

Synopsis

#include <ServerClient.h>

*int SCRecv(SCStream *sp, void *data, int maxBytes, int timeout);*

Arguments

sp	Pointer to open server-client stream.
data	Pointer to data buffer for receipt of data.
maxBytes	Maximum allowed length for arriving data packet (*sizeof(data)*). Packets larger than this value will be discarded, and the return value will be set to **SC_OVERFLOW** to indicate the condition. Normally, this argument will be specified to allow for the largest expected data packet.
timeout	When the stream is configured as blocking, this value specifies a timeout value in milliseconds. A value of -1 (or **SC_NOTIMEOUT**), indicates that there is no timeout, and the call should block forever waiting for data. This argument is ignored when the stream is configured as non-blocking.

Description

SCRecv() receives a packet of data from a Server-Client stream.

Returns

Number of bytes received, 0 if no data was available, **SC_ERROR** (-1) if an unknown error occurred, **SC_NOCONNECTION** (-2) if the client is not currently connected to a server, or **SC_OVERFLOW** (-3) if a data packet was discarded because it was larger than the buffer passed to *SCRecv()*.

See Also

SCOpen(1), SCSend(1)

SCRecvFrom()

Name

SCRecvFrom()
Receive data via a Server-Client stream.

Synopsis

#include <ServerClient.h>

*int SCRecvFrom(SCStream *sp, void *data, int maxBytes, int timeout,* *int *clientId);*

Arguments

sp	Pointer to open server-client stream.
data	Pointer to data buffer for receipt of data.
maxBytes	Maximum allowed length for arriving data packet (*sizeof(data)*). Packets larger than this value will be discarded, and the return value will be set to **SC_OVERFLOW** to indicate the condition. Normally, this argument will be specified to allow for the largest expected data packet.
timeout	When the stream is configured as blocking, this value specifies a timeout value in milliseconds. A value of -1 (or **SC_NOTIMEOUT**), indicates that there is no timeout, and the call should block forever waiting for data. This argument is ignored when the stream is configured as non-blocking.

Description

SCRecvFrom() receives a packet of data from a Server-Client stream, returning the clientId of the sender. This routine is normally used in conjunction with SCSendTo() to implement a server process that waits for requests from multiple clients, services the request, and returns a result.sends a packet of data to all clients currently connected to the specified data stream.

Returns

Number of bytes received, 0 if no data was available, **SC_ERROR** (-1) if an unknown error occurred, **SC_NOCONNECTION** (-2) if the client is not currently connected to a server, or **SC_OVERFLOW** (-3) if a data packet was discarded because it was larger than the buffer passed to *SCRecv()*.

See Also

SCOpen(), SCSendTo(1)

SCSelect()

Name

SCSelect()
Send data via a Server-Client stream.

Synopsis

#include <ServerClient.h>

*int SCSelect(SCStream *streamSet[], int nStreams, int timeout,*
*SCStream *readySet[]);*

Arguments

streamSet	Set of input streams to check for I/O readiness.

nStreams	Number of streams in stream set
timeout	Timeout value in milliseconds. A value of **SC_NOTIMEOUT (-1)** indicates that the call should block indefinitely waiting for I/O.
readySet	Set of streams with pending I/O

Description

Wait (block) for input to become available on at least one of the streams in the specified set. Connection attempts are made periodically for any streams in the set that are in the unconnected state (either never connected or a connection was dropped). The connect attempt interval for each stream can be controlled by the call to SCSetConnectInterval() (default value = 5 sec)

This routine is normally used by processes that read data from multiple data streams, to avoid polling each stream separately. A call to SCSelect(), followed by a SCRecv() call for each ready stream is used instead.

Returns

Number of streams in readySet. A value of indicates that a timeout occurred prior to any streams being ready.

See Also

SCOpen(1), SCRecv()

SCSend()

Name

SCSend()
Send data via a Server-Client stream.

Synopsis

#include <ServerClient.h>

*int SCSend(SCStream *sp, void *data, int bytes, int timeout);*

Arguments

sp	Pointer to open server-client stream object.
data	Pointer to data buffer to send.
bytes	Number of bytes to send
timeout	When the stream is configured as blocking (i.e., *SCOpen()* default; **SC_NONBLOCK** was *not* specified), this value specifies a timeout value in milliseconds. A value of -1 (or **SC_NOTIMEOUT**) indicates that there is no timeout, and the call should block forever on a full client queue. This argument is ignored when the stream is configured as non-blocking. (NOTE: This argument is not yet fully supported, and is intended as a placeholder for a future release.

Description

SCSend() sends a packet of data to all clients currently connected to the specified data stream. When using the TCP protocol in combination with non-blocking mode, a full queue for ANY of the connected clients will result in a return value of 0 bytes. When using the UDP (broadcast) protocol, the returned byte count will always equal the amount specified in the *SCSend()* call, since the transmitter has no way of knowing if any clients are actually receiving the data.

Returns

Number of bytes sent, 0 if server stream is configured as non-blocking and data was not successfully transmitted due to a full client queue, **SC_NOCONNECTION** if no clients are currently connected (TCP only), or **SC_ERROR** (-1) if an (unknown) error occurred.

See Also

SCOpen(1), SCRecv(1)

SCSendTo()

Name

SCSendTo()
Send data to specified client via a Server-Client stream.

Synopsis

#include <ServerClient.h>

*int SCSendTo(SCStream *sp, void *data, int bytes, int timeout, int clientId);*

Arguments

sp	Pointer to open server-client stream object.
data	Pointer to data buffer to send.
bytes	Number of bytes to send
timeout	When the stream is configured as blocking (i.e., *SCOpen()* default; **SC_NONBLOCK** was *not* specified), this value specifies a timeout value in milliseconds. A value of -1 (or **SC_NOTIMEOUT**) indicates that there is no timeout, and the call should block forever on a full client queue. This argument is ignored when the stream is configured as non-blocking. (NOTE: TIMEOUT NOT YET SUPPORTED FOR *SCSend()*)
clientId	Client Id, as obtained from SCRecvFrom(). If specified as SC_ALL_CLIENTS, the data packet will be written to all clients. If specified as negated version of a clientId, data is sent to all client *except* the specified client.

Description

SCSendTo() sends data to the specified client. This routine is normally used in conjunction with SCRecvFrom() to implement a server process that waits for requests from multiple clients, services the request, and returns a result.sends a packet of data to all clients currently connected to the specified data stream.

Returns

Number of bytes sent, 0 if server stream is configured as non-blocking and data was not successfully transmitted due to a full client queue, **SC_NOCONNECTION** if no clients are currently connected (TCP only), or **SC_ERROR** (-1) if an (unknown) error occurred.

See Also
SCOpen(1), SCRecvFrom(1)

SCSetBlocking(), SCSetQueueSize()

Name

SCSetBlocking(), SCSetQueueSize()
Set functions for server-client streams.

Synopsis

#include <ServerClient.h>

void SCSetBlocking(SCStream sp, int blockFlag);*

void SCSetQueueSize(SCStream sp, int queueSize);*

Arguments

sp	Pointer to open server-client stream object.
blockFlag	TRUE to enable blocking, or FALSE to disable blocking.
queueSize	Size for server or client queue, in bytes.

Description

SCSetBlocking() sets a server-client stream to blocking or non-blocking. Typically the blocking mode will be set by the call to *SCOpen()*, but it is sometimes handy to temporarily switch modes back and forth in complex applications.

NOTE: Non-blocking mode is not yet supported for streams configured as servers (**SC_WRONLY**) -- only clients.

SCSetQueueSize() sets the queue size for the specified server-client stream The queue size defaults to 512K (Large, but necessary for backward compatibility with sc_pac applications and config files). A more flexible approach to setting the queue size is to use the *-queueSize* option in the server-client configuration file. See the *SCLite* library documentation for more details.

Returns

Nothing.

See Also

SCOpen()

Name

SCSetConnectHandler(), SCSetDisconnectHandler()
Set handler functions for connection state changes

Synopsis

```
#include <ServerClient.h>

void SCSetConnectHandler( SCServer *sp,
                void ( *handler )( SCServer *sp, int clientId, void *arg ),
                void *handlerArg );

void SCSetDisconnectHandler( SCServer *sp,
                void ( *handler )( SCServer *sp, int clientId, void *arg ),
                void *handlerArg );
```

Arguments

sp	Pointer to active server object.
handler	Handler function to call when new connection or broken connection occurs. When invoked, the handler's first two arguments are the pointer to the stream with a new or broken connection, and the unique clientId for the new or broken connection. The third argument will be the handler argument passed by the user to the corresponding Set() function (see below)
handlerArg	Argument to pass to handler function when it is invoked.

Description

SCSetConnectHandler() and *SCSetDisconnectHandler()* install a handler function to be called when a new client connects to, or existing client disconnects from, a server stream. New connection/ broken connection processing is normall performed upon entry to each *SCSend()* or *SCRecvFrom()* call (which call depending on whether the server is configured as a data server or a message server). The handler is invoked at user-level (not via a signal handler), allowing for safe use of all system functions (*malloc()*, *free()*, etc...).

A typical use of the connect handler is to send additional 'startup' information to newly connected clients, by invoking a separate call to *SCSend()* within the handler function. The handler invocation code is set up to expect this usage -- it guarantees that an *SCSend()* called from within a handler will NOT process any additional incoming connections (and possibly recursively invoke the connection handler). It is also guaranteed that the data transmission triggered by the handler's *SCSend()* will occur *prior* to the data transmission for the original *SCSend()*, allowing for the 'startup' information to be transmitted to the client before any other data is set.

A typical use of the disconnect handler is to free up any shared resources that are 'owned' by the client with the matching clientId. This helps to prevent possible program deadlocks when a network connection becomes unreliable for any reason.

Note: These calls are only currently valid for servers using the TCP protocol. (Since UDP is connectionless, they don't apply).

Returns Nothing.

See Also *SCSend(), SCRecvFrom()*

SCSetRecType(), SCGetRecType(), SCSetHdrType()

Name ***SCSetRecType(), SCGetRecType(), SCSetHdrType()***
sc_pac library compatibility functions.

Synopsis *#include <ServerClient.h>*

*void SCSetRecType(SCStream *sp, unsigned short recType);*

unsigned short SCGetRecType(SCStream sp);*

*void SCSetHdrType(SCStream *sp, int hdrType);*

Arguments

sp	Pointer to open server-client stream object.
recType	Record type of the server-client protocol header.
hdrType	**SC_HEADER_16BYTE** (default) to transmit 16-byte headers, **SC_HEADER_8BYTE** to transmit 8-byte headers. When using the (old) 8-byte headers, the maximum record size than can be transmitted is 128 KBytes.

Description In typical usage, server-client protocol headers are transparent to the application. The headers (containing record length, type, sequence # info) are prepended to the user's data buffer when sending records and stripped off on the other end prior to being 'seen' by a client process. This was not the case with an older version of the communication library, *sc_pac*, where the headers were visible at the application level, and, in fact, used to encapsulate record type information. To support I/O from/to these applications, a mechanism is needed to get/set the record type of the protocol header. *SCGetRecType()* and *SCSetRecType()* provide this mechanism. In addition, to support *really* old *sc_pac* applications, the function *SCSetHdrType()* is supplied to allow the user to control whether the protocol headers transmitted are the new 16-byte variety (default) or the older 8-byte variety (128K max record length).

When sending data, a call to *SCSetRecType()* should precede a corresponding call to *SCSend()*. If only one record type is being used, a single call to *SCSetRec-*

Type() following the call to *SCOpen()* will result in *all* transmitted records being tagged with the specified record type. When receiving data, a call to *SCGetRecType()* following a successful call to *SCRecv()* will retrieve the record type of the record just received.

An alternative method of achieving backward compatibility is to use the **SC_RAW** mode flag to make the headers visible to the application. This may have some uses, but it is not recommended, since the raw header information utilizes big-endian byte ordering, and the application will not be portable to a little-endian (Intel) machine.

Returns

SCGetRecType() returns the record type contained in the header block of the last record received. *SCSetHdrType()* and *SCSetRecType()* return nothing.

See Also

SCOpen()

GLOSSARY

AP	Anomalous Propagation
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
FAA	Federal Aviation Administration
LAN	Local Area Network
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol

REFERENCES

1. Newell, O.J., "ASR-9 Weather Systems Processor Software Overview", MIT Lincoln Laboratory, Project Report ATC-264, 20 October 2000.