CTIX Message System User's Manual Version 1.0

H. Conrad Cunningham, Michael E. Ehlers, and Kenneth C. Cox

This manual describes how to use the CTIX Message System for interprocess communication in a distributed application program. The CTIX Message System is a package of message-passing facilities developed by the Concurrent Systems Group of the Department of Computer Science at Washington University. It provides a process-to-process asynchronous, buffered communication medium. The package is implemented on a network of Convergent Technologies (CT) MiniFrame workstations. These workstations support the CTIX (the Ct's version of UNIX System V) operating system and the TCP/IP network protocols.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
CTIX MESSAGE SYSTEM USER'S MANUAL
Version 1.0

H. Conrad Cunningham, Michael E. Ehlers
and Kenneth C. Cox

WUCS-88-10

March 1988

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract
This manual describes how to use the CTIX Message System for interprocess communication in a
distributed application program. The CTIX Message System is a package of message-passing facili-
ties developed by the Concurrent Systems Group of the Department of Computer Science at Wash-
ington University. It provides a process-to-process, asynchronous, buffered communication medium.
The package is implemented on a network of Convergent Technologies (CT) MiniFrame worksta-
tions. These workstations support the CTIX (CT's version of UNIX System V) operating system and
the TCP/IP network protocols.
UNIX is a trademark of AT&T Bell Laboratories. Convergent Technologies, Convergent, MiniFrame Computer, MiniFrame, and CTIX are trademarks of Convergent Technologies, Inc.
## Contents

1 Overview .............................................................. 1

2 Message System Data Types .................................... 1
   2.1 Process Identifiers ........................................... 1
   2.2 Process Selectors .......................................... 2
   2.3 Message Keys ............................................... 2
   2.4 Message Key Selectors .................................... 3
   2.5 Message Components ..................................... 4

3 Message System Interface Functions ......................... 4
   3.1 Signing Onto the Message System ......................... 4
   3.2 Signing Off the Message System ......................... 5
   3.3 Sending a Message ....................................... 5
   3.4 Receiving a Message ..................................... 5
   3.5 Accessing the Current Input Message ................... 5
   3.6 Remote Process Startup ................................ 6
   3.7 Information Operations .................................. 6
   3.8 Miscellaneous Operations ................................. 7

4 Application Program Building .................................. 7
   4.1 Header Files .............................................. 7
   4.2 Compilation and Linking .................................. 7
   4.3 Cautions .................................................. 8

5 Acknowledgements ................................................ 8

A Copier Program Example ......................................... 9

B Message System Programmer's Manual ......................... 17
1 Overview

The CTIX Message System (MS) is a software package that provides simple facilities for the passing of messages among the processes of a distributed application program. The package executes on a network of Convergent Technologies (CT) MiniFrame workstations. These workstations support the CTIX (CT's version of UNIX System V) operating system and the TCP/IP network protocols. The Message System's user interface consists of a number of C-language functions (available via an object library) and source header files. Supported by a set of system processes distributed over the network, these functions provide a process-to-process, asynchronous, buffered communication medium.

The processor configuration is assumed to consist of a static, fully connected network of identical machines. Each node has a unique, statically assigned identifier. (Here "static" means the configuration and naming scheme do not change during an execution of a distributed user application.)

Although processes may be started dynamically, once started, they cannot move between nodes. Processes communicate via the MS by passing messages, distinct collections of data items. Each communicating process has a unique MS process identifier.

2 Message System Data Types

2.1 Process Identifiers

Every process that uses the Message System has a network-wide unique process identifier, an object of type ProcId.\(^1\) A process' ProcId is assigned by the MS when the process "signs on" to the message system. It remains constant during the session of MS usage, i.e., until the process "signs off" of the MS. If the process signs off and then signs on again, its process identifier may be different. A ProcId object is a triple of the form:

\[
(node, group, member)
\]

where each component is an integer of the proper type and range (given below). This triple is implemented as a C struct. Particular component values may be reserved for Message System usage.

- **node** is the identifier for the node (machine) upon which the process is resident. The values of these identifiers are statically associated with the nodes of the network.

  Representation: integer type NodeId  
  Range: \(1 \leq \text{node} \leq MAX\text{nodeId}\)

---

\(^1\)Object type and constant names used in this manual correspond to C typedef and #define names provided in the Message System header files.
group is the identifier for the group of processes of which the process is a
member. A group may consist of one or more processes. The value of
this component is selected by the user application process when it "signs
on" to the MS. All processes on the network which use the MS
concurrently must use group identifiers in a consistent manner.

Representation: integer type GroupId
User Range: \(1 \leq group \leq MAX\_GroupId\)

member is the identifier for the process as a member of the process group on the
node. The value of this component is assigned dynamically by the MS at
"sign on".

Representation: integer type MemberId
Range: \(1 \leq member \leq MAX\_MemberId \cup \{DONT\_CARE\}\)

The UNIX process identifier for a process, an object of type UnixId, is used by some of the
Message System functions. The UnixId for a process is sometimes needed so that UNIX system
operations can be applied to a process.

Representation: integer type UnixId

2.2 Process Selectors

A process selector, an object of type ProcSel, defines a selection predicate over ProcIds, i.e., it
defines a set of values of a ProcId that are acceptable for some purpose. A process selector is a
triple of the form:

\((node, group, member)\)

where each component is an integer of the proper type and range (given above) or DONT\_CARE.
A ProcSel may be used to express the set of processes from which messages may be received. To be
selected by a ProcSel, the values of the components of a process' ProcId must equal the values of
the components of the ProcSel. When a component of a ProcSel has the wildcard value
DONT\_CARE, then any valid value is acceptable to that component.

Representation: same as ProcId
Range: same as ProcId except value DONT\_CARE allowed for each component

2.3 Message Keys

A message key, an object of type MsgKey, is a user-selected tag value that can be attached to
messages in the Message System. Each message has a sequence of from zero to MAX\_num\_Keys
distinct keys. The keys within a message are indexed by integers beginning with 1.

\(^2\)The MS group concept is not related to the UNIX group concept.
Representation:  integer type \textit{MsgKey}

Range: \quad \textit{MINmsgKey} \leq \textit{MsgKey} \leq \textit{MAXmsgKey}

Keys may be used to encode information such as message formats, operation codes, priorities, transaction identifiers, sequence numbers, and so forth. For example, a user application may choose to use the first key to encode the type of a message, using key values drawn from a set of integer type codes. Application processes can then send messages with the first key set to a code representing the type of the message. Other application processes can request receipt of only messages having a certain type code for the first key.

2.4 Message Key Selectors

A message key selector, an object of type \textit{MsgKeySel}, defines a selection predicate over \textit{MsgKeys}. When applied to a \textit{MsgKey} value, a message key selector determines whether or not the key's value is acceptable. When a sequence of \textit{MsgKey Sel}s is applied to the sequence of \textit{MsgKeys} attached to a message, the message is acceptable only if all the key values are acceptable to the respective key selectors. If the sequence of message keys is longer than the sequence of key selectors, then the extra keys (with the highest indices) are not considered in the acceptability test. \textit{MsgKey Sel}s, along with \textit{ProcSel}s, are used by processes to select messages from their message queues.

A \textit{MsgKeySel} is a triple (implemented as a C struct) of the form:

\[
\langle \text{op}, \text{parm}_1, \text{parm}_2 \rangle
\]

where

\text{op} \quad \text{is an integer value denoting the selection (comparison) operation to be performed on the message Key.}

Representation:  integer type \textit{KeyOp}

Supported key selection operations include:

\begin{itemize}
  \item \textit{KeyEQ} \quad \text{message Key EQual to \text{parm}_1}
  \item \textit{KeyNEQ} \quad \text{Not EQual}
  \item \textit{KeyLEQ} \quad \text{Less or EQual}
  \item \textit{KeyGEQ} \quad \text{Greater or EQual}
  \item \textit{KeyIR} \quad \text{message Key In inclusive Range from \text{parm}_1 to \text{parm}_2}
  \item \textit{KeyNIR} \quad \text{Not In inclusive Range}
  \item \textit{KeyANY} \quad \text{message Key must be present (ANY value)}
  \item \textit{KeyNONE} \quad \text{message Key must be absent (NO value)}
  \item \textit{KeyNOP} \quad \text{always succeeds}
\end{itemize}

The \textit{KeyANY} and \textit{KeyNONE} operations are only meaningful when a sequence of message keys is being tested against a sequence of key selectors. \textit{KeyNOP} may also be useful in this context.

\text{parm}_i \quad \text{is a MsgKey value used as an operand in the selection operation designated by \text{op}.}

A function is provided which constructs \textit{MsgKey Sel}s from their components.
2.5 Message Components

A message is a logical collection of data items grouped together for transfer among processes via the Message System. A message is a tuple of the form:

\[ (\text{Source, Dest, NumKeys, KeyArray, LenData, DataBlock}) \]

where

- **Source** is the ProcId of the sending process.
- **Dest** is the ProcId of the destination process.
- **NumKeys** is the number of values in the **KeyArray**, i.e., the number of keys attached to the message.

  Representation: integer type **Counter**
  Range: \[ 0 \leq \text{NumKeys} \leq \text{MAXnumKeys} \]

- **KeyArray** is an array of length **NumKeys** whose elements are **MsgKeys**. The values are selected by the sending process.

- **LenData** is the number of bytes in the **DataBlock** component.

  Representation: integer type **Counter**
  Range: \[ 0 \leq \text{LenData} \leq \text{MAXlenData} \]

- **DataBlock** is a block of data bytes of length **LenData**.

  Representation: array of **char**

3 Message System Interface Functions

This section sketches the user application's interface to the Message System's facilities. Appendix A shows an example application program. Appendix B gives more detailed descriptions of each function. The user interface to the Message System includes several C functions. Unless otherwise specified, the functions return **SUCCESS** for successful completion and **FAILURE** if an error has occurred. In the case of an error, the global variable **ms_errno** is set to a nonzero error number to indicate the type of error that has occurred.

3.1 Signing Onto the Message System

**MSignOn(Group)** signs the calling process onto the Message System as a member of the process group **Group**. As a result, a ProcId and a message queue are allocated to the process. The process may then communicate with other such processes via the Message System. This call must precede calls of all other MS functions. (All processes on the network which use the MS concurrently must use group identifiers in a consistent manner.)
3.2 Signing Off the Message System

MSignOff() removes the calling process from the Message System. All messages queued for that process are destroyed and its ProcId is released. After signing off, a process may sign on again with another call to MSignOn.

MExit(n) performs an MSignoff() followed by an exit(n).

3.3 Sending a Message

MSend(Dest, LenData, DataBlock, NumKeys, KeyArray) constructs a message from the indicated message components and sends it to the Dest process’ message queue. If the member component of Dest is DONT.CARE, then the message is delivered to any one process in the specified group on the specified node. DONT.CARE values in the group and node fields are not supported. If NumKeys = 0, then KeyArray may be NULL.

3.4 Receiving a Message

MReceive(TimeOut, FromSel, NKeySels, KeySel) finds an acceptable message in the process’ queue, removes it from the queue, and places it into the calling process’ Current_Input_Message buffer. To be received, a message must be acceptable to all of the process (FromSel) and message key (KeySel) selectors specified in this call. If an acceptable message is not found in TimeOut seconds, then the call returns FAILURE. Special TimeOut values DONT_WAIT and WAIT_FOREVER are supported.

MCheck(TimeOut, FromSel, NKeySels, KeySel) checks for the availability of an acceptable message in the queue without receiving it. If an acceptable message is found, the function returns SUCCESS, otherwise it returns FAILURE and sets ms_errno appropriately.

MCount(MsgCount, FromSel, NKeySels, KeySel) counts the number of queued messages that satisfy the selectors and returns the count in MsgCount.

3.5 Accessing the Current Input Message

The functions described below provide access to the components of the Current_Input_Message, i.e., the last message received. These functions will return FAILURE if no message has been received, or if MClrCur has been executed since the last receive. Otherwise they will return SUCCESS and retrieve the specified component.

MClrCur() clears the Current_Input_Message buffer.

MSource(FromId) copies the ProcId for the Source process of the Current_Input_Message into FromId.

MDest(ToId) copies the ProcId for the Dest process of the Current_Input_Message into ToId.

MNumKeys(Nkeys) copies the NumKeys component of the Current_Input_Message into Nkeys.

MKey(KeyNum, KeyVal) copies the MsgKey value for KeyNumth element of KeyArray from the Current_Input_Message into KeyVal.
MLenData(Bytes) copies the LenData component of the CurrentInputMessage into Nbytes.

MDataBlock(MaxBlock, Block) copies the first min(MaxBlock, LenData) bytes of the DataBlock component of the CurrentInputMessage into Block.

3.6 Remote Process Startup

Processes using the Message System can start other processes on any node in the network configuration. To use the Message System each remotely started process must call MSignOn in the same way other processes do. The following functions are provided for remote process startup.

StartProcess(remNode, Program, ArgList, r_env, r_in, r_out, r_err) initiates (invokes) the execution of a process on remote node remNode. This function causes file Program on the remote node to be executed with UNIX command-line argument list ArgList, environment r_env, and standard input, output, and error channels directed to the files r_in, r_out, and r_err respectively. If r_env is NULL, then the environment of the calling process will be passed to the new process. If any of the i/o channel arguments are NULL, then that channel of the started process will be directed to "/dev/null". Note: When StartProcess returns, the request for startup has been sent, but the remote process has not necessarily been started.

GetInvoker(invokerId) returns the ProcId of the invoking process in InvokerId. If the process was not remotely started, then the call returns FAILURE; otherwise it returns SUCCESS.

3.7 Information Operations

The functions described below retrieve information maintained in the Message System data structures.

MyProcId(theProcId) retrieves the complete MS process identifier (theProcId) for the calling process.

ListProc(aProcSel, MaxCount, Count, ProcList) generates a list of ProcIds of processes that satisfy the process selector aProcSel and returns the number of entries in Count and the first min(MaxCount, Count) entries in the array ProcList. A "wildcard" value of DONT.CARE for the group or member components of aProcSel means any valid value will be accepted in the component. DONT.CARE is node supported for the node component. If MaxCount = 0, then ProcList may be NULL.

toUnixId(aProcId, theUnixId) returns, in theUnixId, the UNIX process identifier of the process which has MS process identifier aProcId.

toProcId(aUnixId, theProcId) returns, in theProcId, the MS process identifier of the process which has UNIX process identifier aUnixId.

toNodeid(nodename, nodeid) returns, in nodeid, the MS node identifier of the machine that has UNIX node (host) name nodename.

toNodeName(nodeid, nodename) returns, in nodename, the UNIX node (host) name of the machine that has MS node identifier nodeid.

mserror(str) prints, on stderr, the text string str followed by a description of the last MS error that has occurred.
3.8 Miscellaneous Operations

The functions described below provide various facilities that may be useful in application processes. Most of these functions return a value other than the SUCCESS/FAILURE code used by other MS functions.

UniqKey(NEWKEY) generates a network-wide unique MsgKey, places it in NewKey, and returns SUCCESS. If the key cannot be generated, FAILURE is returned and ms_errno is set appropriately.

aProcId(aNodeId, aGroupId, aMemberId) constructs and returns a ProcId or ProcSel value with the specified component values.

aKeySel(aKeyOp [aKey1 [aKey2]]) returns a MsgKeySel value with the specified component values. Key fields not needed by a particular selection operator may be omitted.

CompProcId(ProcId1, ProcId2) compares ProcId1 to ProcId2 and returns a negative value (LESS) if the first is less, zero (EQUAL) if equal, and a positive value (GREATER) if greater.

bad ProcId(id) returns 0 if id is a valid ProcId, i.e., if its components are within the valid ranges; otherwise, a nonzero error code is returned. (This call does NOT determine whether a process with that ProcId has actually signed onto the Message System.)

4 Application Program Building

4.1 Header Files

The Message System functions can be called from any CTIX C program. Various type, constant, error code, and data structure definitions must be included in each source file that accesses Message System facilities. This can be done with the statement:

#include (MS/user.h)

4.2 Compilation and Linking

The Message System functions have been made available as a system object library named libMS.a. Users can link the appropriate functions into their application programs by appending the option -lMS to the cc command used to (compile and) link the program. The command

cc userfile.c -lMS

compiles C source file userfile, links it with the MS library, and creates the executable program file, a.out.
4.3 Cautions

Since process identifiers are network-wide entities, users must be careful in their choice of group identifiers. All processes—belonging to all users—which use the Message System concurrently must use groups in a consistent manner. By carefully choosing the group identifiers used, several applications can be executed simultaneously.

The implementation of the Message System user library uses CTIX's shared memory and semaphore facilities. User processes (programs) which use the Message System can use shared memory and semaphores for other purposes as long as those uses do not interfere with Message System operation. (Contact the local Message System administrator for more information.)

The implementation of the Message System user library also uses the signals SIGUSR1 and SIGALRM and the CTIX process alarm clock. These facilities are not available to user processes which are signed onto the Message System.

The use of CTIX semaphores and the CTIX TCP/IP networking facilities (i.e., sockets) within the same program does not seem to work. Hence, user processes which use the Message System cannot use the TCP/IP facilities.

5 Acknowledgements

The Department of Computer Science at Washington University supported the development of the CTIX Message System. The goal of this work was to develop a simple message-passing system which can (1) facilitate instruction on distributed systems issues and (2) support experimental work in concurrency. We thank Dr. J. H. Cox, Jr., the department chairman, for his support of this project.

The CTIX Message System (Version 1.0) software was developed by the department's Concurrent Systems Group during the period from December 1987 through March 1988. Dr. Catalin Roman supervised the overall activities of the group and formulated the objectives of the project. The software package was designed and implemented by Conrad Cunningham, Mike Ehlers, Ken Cox, and Howard Lykins. Wei Chen participated in various discussions concerning the package's design. The workstations used for this work were donated to the University by Convergent Technologies, Inc., and made available to this project by the Engineering Computer Laboratory (ECL) and the Department of Computer Science. We thank ECL staff members Kevin Fenster, Bill Ross, and Peter McLain for their technical assistance with the CT workstations and the network facilities.
A  Copier Program Example

/* copier program - main process

   compile with: cc -o copy copy.c -lMS
   call with:    copy sourcename sourcefile desname desfile

   Invokes workers on remote systems sourcename and desname to do the copy. */

#include <MS/user.h>
define COPIER    "/users/mgsys/test/copworker"
define COPY_GROUP 1
define ID_KEY    (MINmsgKey)

static char data[MAXlenData];

main(argc,argv)
int argc;
char **argv;
{
   NodeId sendnode, recvnode;
   ProcId send_id, recv_id;
   MsgKey key[1];

   if (argc != 5) {
      printf("Illegal program use:\n\t");
      for ( ; argc-- > 0; argv++) printf("%s ",argv);
      printf("\n\t");
      exit(1);
   }

   /* Sign on to the MS as a member of group COPY_GROUP */
   MSsignOn(COPY_GROUP);

   /* Translate node names to node id's */
   toNodeId(argv[1], &sendnode); toNodeId(argv[3], &recvnode);

   /* Start the two worker processes */
   startup(sendnode, "-s", argv[2], &send_id);
   startup(recvnode, "-r", argv[4], &recv_id);

/* Send the ProcId of the receiver to the sender and vice-versa */

key[0] = ID_KEY;
if (!MSend(send_id, sizeof(ProcId), (char *)&recv_id, 1, key) ||
    !MSend(recv_id, sizeof(ProcId), (char *)&send_id, 1, key)) {
    merror("Starter: send failed");
    MExit(1);
}

/* Wait for sender and receiver to finish the copy */

if (!MReceive(WAIT_FOREVER, send_id, 0, (MsgKeySel *)NULL) ||
    !MReceive(WAIT_FOREVER, recv_id, 0, (MsgKeySel *)NULL)) {
    merror("Starter: checkback error");
    MExit(1);
}

/* exit from the message system */

MExit(0);
void startup(node, arg1, arg2, id)
    NodeId node;
    char *arg1,*arg2;
    ProcId *id;
{
    int strlen();
    /* Put arguments into data block */
    if (strlen(arg1) + strlen(arg2) + 2 > MAXlenData)
        {printf("Argument too long: \"%s\"\n", arg2);
            MSexit(1);
        }
    printf(data,"%s %s", arg1, arg2);
    /* Start COPIER on node, with arguments in data, current process' environment, and stdin, stderr, and stdout all attached to /dev/null */
    if (!StartProcess(node, COPIER, data, 0, "/dev/null", "/dev/null", "/dev/null") )
        {mserror("Starter: process startup failed");
            MSexit(1);
        }
    /* wait for callback with started process' ProcId */
    if (!MReceive(WAIT_FOREVER, aProcId(node, COPY_GROUP, DONT_CARE),
        0, (MsgKeySel *) NULL) ||
        !MSrc(source(id)) )
        {mserror("Starter: failed to get callback");
            MSexit(1);
        }
/*
   copier program - worker process
   invoked with either

   cpworker -s file   if it is preforming the read/send operation
   cpworker -r file   if it is preforming the write/receive operation

   compiled with

   cc -o cpworker cpworker.c -lMS

*/

#include <MS/user.h>
#include <fcntl.h>

#define COPY_GROUP 1
#define ID_KEY  (MInmsgKey)
#define DATA_KEY  (MInmsgKey+1)
#define EWD_KEY  (MInmsgKey+2)

static ProcId main_id, send_id, recv_id;
static char data[MAXlenData];

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

    if (argc != 3 || argv[1][0] != '-' ||
        (argv[1][1] != 's' && argv[1][1] != 'r')) {
        printf("Illegal program use:\n\t");
        for (; argc-- > 0; argv++) printf("%s ",argv);
        exit(1);
    }

    MSigOn(COPY_GROUP);

    if (argv[1][1] == 's')
        sender(argv[2]);
    else
        receiver(argv[2]);

    checkout();
    MSig(0);
}

void sender(filename)
  char *filename;
{
    int fd,r;
    MsgKey key[1];

    /* Checkin with whoever started this process */
    checkin(&send_id,&recv_id);

    /* Open the file to be sent */
    if ((fd = open(filename,O_RDONLY)) < 0) {
      printf("Sender: can't open %s\n",filename);
      MExit(1);
    }

    /* Send data in blocks of size MAXlenData with key DATA_KEY */
    key[0] = DATA_KEY;
    while ((r = read(fd,data,MAXlenData)) > 0) {
      if (!MSend(recv_id, r, data, 1, key)) {
        merror("Sender: send failed");
        MExit(1);
      }
    }

    /* Handshake with the receiver */
    if (!MReceive(WAIT_FOREVER, recv_id, 0, (MsgKeySel *)NULL)) {
      merror("Sender: handshake failed");
      MExit(1);
    }
}

/* Send final (empty) message with key END_KEY */

key[0] = END_KEY;
if (!MSend(recv_id, 0, (char *)NULL, 1, key)) {
  merror("Sender: send failed");
  MExit(1);
}
close(fd);
void receiver(filename)
    char *filename;
{
    int fd;
    Counter ct;
    MsgKey key;

    checkin(&recv_id,&send_id);

    if ((fd = open(filename,O_RDONLY)) >= 0) {
        printf("Receive: will not overwrite %s\n",filename);
        MSexit(1);
    } else {
        if ((fd = open(filename,O_WRONLY|O_CREAT,O644)) < 0) {
            printf("Receive: can't open %s\n",filename);
            MSexit(1);
        } else {
            for (;;) { /* get message from sender */
                if (!MReceive(WAIT_FOREVER, send_id, 0, (MsgKeySel *)NULL)) {
                    merror("Receive: receive failed");
                    MSexit(1);
                } else {
                    if (!MNumKeys(&ct) || ct != 1 || !MKey(i,&key) ||
                        (key != END_KEY && key != DATA_KEY)) {
                        printf("Receive: error in key\n");
                        MSexit(1);
                    } else {
                        if (key == END_KEY) break; /* all data has been sent */

                        /* write data to file */
                        if (!MLenData(&ct) || !MDataBlock(ct,data)) {
                            merror("Receive: error getting data");
                            MSexit(1);
                        } else {
                            if (write(fd,data,(unsigned)ct) != ct) {
                                printf("Receive: write error\n");
                                MSexit(1);
                            }
                        }
                        /* handshake with sender */
                        if (!MSend(send_id, 0, (char *)NULL, 0, (MsgKey *)NULL)) {
                            merror("Receive: handshake failed");
                            MSexit(1);
                        }
                    }
                }
            }
            close(fd);
        }
    }
}
void checkin(my_id, his_id)
    ProcId *my_id, *his_id;
{
    Counter n;
    MsgKey key;

    if (!MyProcId(my_id) || !GetInvoker(&main_id)) {
        merror("Can't locate process ids");
        MSexit(1);
    }

    /* Send my ProcId to the invoking process (copy) */

    if (!MSend(main_id, sizeof(ProcId), (char *)my_id, 0, (MsgKey *)NULL)) {
        merror("Can't send id to main");
        MSexit(1);
    }

    /* Get back the other worker's ProcId */

    if (!MReceive(WAIT_FOREVER, main_id, 0, (MsgKeySel *)NULL)) {
        merror("Receive error in checkin");
        MSexit(1);
    }

    if (!MNumKeys(&m) || n != 1 || MKey(i, &key) || key != ID_KEY) {
        printf("Key error in checkin\n");
        MSexit(1);
    }

    if (!MLenData(&n) || (int)n != sizeof(ProcId)) {
        printf("Data size error in checkin\n");
        MSexit(1);
    }

    if (!MDataBlock(n, (char *)his_id)) {
        printf("Data retrieve error in checkin\n");
        MSexit(1);
    }
}

void checkout()
{
    if (!MSend(main_id, 0, (char *)NULL, 0, (MsgKey *)NULL)) {
        merror("Can't send checkout to main");
        MSexit(1);
    }
}
B Message System Programmer’s Manual

This appendix consists of the CTIX Message System Programmer’s Manual. It provides specific documentation on each of the functions that form the Message System user library. These “man pages” are available on the CTIX systems via the man documentation system.
NAME
intro - introduction to the Message System functions

DESCRIPTION
The Message System is a mechanism for interprocess communication and process creation in a
distributed application program. The package consists of two parts, one is set of system
processes, server, executor, sender, and receiver, which should be running on each system; the
other is the set of functions to interface with the system processes to perform communication
and process creation. These functions are contained in the library /usr/lib/libMS.a, which
should be included when the program is loaded.

If the system programs are not running, they must be restarted (contact the local Message
System administrator).

FILES
/usr/lib/libMS.a the Message System library
/usr/include/MS/user.h definitions needed by user programs
/usr/include/MS/constants.h Message System constant definitions
/usr/include/MS/types.h Message System data types definitions
/usr/include/MS/errors.h definition of Message System errors

SEE ALSO
cc(1), ld(1)

LIST OF FUNCTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Appears on Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aKeySel</td>
<td>AKEYSEL.3MS</td>
<td>construct Message Key Selector</td>
</tr>
<tr>
<td>aProcId</td>
<td>APROCID.3MS</td>
<td>construct process identifier</td>
</tr>
<tr>
<td>badProcId</td>
<td>BAD_PROCID.3MS</td>
<td>check a process identifier</td>
</tr>
<tr>
<td>CompProcId</td>
<td>COMPPROC.3MS</td>
<td>compare two ProcIds</td>
</tr>
<tr>
<td>GetInvoker</td>
<td>GETINVOKER.3MS</td>
<td>get id of invoking process</td>
</tr>
<tr>
<td>ListProc</td>
<td>LISTPROC.3MS</td>
<td>list processes</td>
</tr>
<tr>
<td>MCheck</td>
<td>MRECEIVE.3MS</td>
<td>check for waiting messages</td>
</tr>
<tr>
<td>MClrCur</td>
<td>CURMSG.3MS</td>
<td>clear last message buffer</td>
</tr>
<tr>
<td>MCount</td>
<td>MRECEIVE.3MS</td>
<td>count waiting messages</td>
</tr>
<tr>
<td>mDataBlock</td>
<td>CURMSG.3MS</td>
<td>retrieve last message's data</td>
</tr>
<tr>
<td>MDest</td>
<td>CURMSG.3MS</td>
<td>retrieve last message's destination id</td>
</tr>
<tr>
<td>MKey</td>
<td>CURMSG.3MS</td>
<td>retrieve a key from last message</td>
</tr>
<tr>
<td>MLenData</td>
<td>CURMSG.3MS</td>
<td>retrieve last message's data length</td>
</tr>
<tr>
<td>MNumKeys</td>
<td>CURMSG.3MS</td>
<td>retrieve the key count in last message</td>
</tr>
<tr>
<td>MReceive</td>
<td>MRECEIVE.3MS</td>
<td>receive a message</td>
</tr>
<tr>
<td>MSend</td>
<td>MSEND.3MS</td>
<td>send a message</td>
</tr>
<tr>
<td>mSError</td>
<td>MERROR.3MS</td>
<td>print Message System error messages</td>
</tr>
<tr>
<td>MSexit</td>
<td>MSIGNON.3MS</td>
<td>leave Message System and exit</td>
</tr>
<tr>
<td>MSignOff</td>
<td>MSIGNON.3MS</td>
<td>leave Message System</td>
</tr>
<tr>
<td>MSignOn</td>
<td>MSIGNON.3MS</td>
<td>enter Message System</td>
</tr>
<tr>
<td>MSource</td>
<td>CURMSG.3MS</td>
<td>retrieve the sender id of last message</td>
</tr>
<tr>
<td>MyProcId</td>
<td>MYPROCID.3MS</td>
<td>process identifier of calling process</td>
</tr>
<tr>
<td>StartProcess</td>
<td>STARTPROCESS.3MS</td>
<td>start execution of a process</td>
</tr>
<tr>
<td>toNodeId</td>
<td>TONODEID.3MS</td>
<td>convert string to NodeId</td>
</tr>
<tr>
<td>toNodeName</td>
<td>TONODENAME.3MS</td>
<td>convert NodeId to string</td>
</tr>
<tr>
<td>toProcId</td>
<td>TOPROCID.3MS</td>
<td>convert UNIX pid to ProcId</td>
</tr>
<tr>
<td>toUnixId</td>
<td>TOUNIXID.3MS</td>
<td>convert ProcId to UNIX pid</td>
</tr>
<tr>
<td>UniqKey</td>
<td>UNIQKEY.3MS</td>
<td>generate unique message key</td>
</tr>
</tbody>
</table>

March 9, 1988
WARNINGS

The implementation of the Message System user library uses CTIX's shared memory and semaphore facilities. User processes (programs) which use the Message System can use shared memory and semaphores for other purposes as long as those uses do not interfere with Message System operation. Contact the local Message System administrator for more information.

The implementation of the Message System user library also uses the signals SIGUSR1 and SIGALRM and the CTIX process alarm clock. These facilities are not available to user processes which are signed onto the Message System. (See signal(2), alarm(2), and kill(2).)

The use of CTIX semaphores and the CTIX TCP/IP networking facilities (i.e., sockets) within the same program does not seem to work. Hence, user processes which use the Message system cannot use the TCP/IP facilities.
NAME
aKeySel - construct a Message Key Selector given component values

SYNOPSIS
#include <MS/user.h>

MsgKeySel aKeySel(aKeyOp, aKey1, aKey2)
KeyOp aKeyOp;
MsgKey aKey1;
MsgKey aKey2;

DESCRIPTION
aKeySel constructs a Message Key Selector (MsgKeySel) with the key selection operator
aKeyOp, the first message key operand aKey1, and the second message key operand aKey2. If
the operator does not make use of one (or both) of the operands, then the unneeded operand(s)
may be omitted. Unneeded fields of the constructed MsgKeySel are set to "null" values.

An aKeyOp with one of the following values has the indicated meaning:
[KeyEQ] message Key EQual to first operand.
[KeyNEQ] message Key Not EQual to first operand.
[KeyLEQ] message Key Less or EQual to first operand.
[KeyGEQ] message Key Greater or EQual to first operand.
[KeyIR] message Key In inclusive Range from first to second operands.
[KeyNIR] message Key Not In inclusive Range from first to second operands.
[KeyANY] message Key must be present with ANY value.
[KeyNONE] message Key must be absent (NO value).
[KeyNOP] always succeeds.

RETURN VALUE
aKeySel returns the constructed value.

ERRORS
No errors are reported. ms_errno is not changed.

SEE ALSO
intro(3MS), MReceive(3MS)
NAME
aProcId — construct a Message System process identifier given component values

SYNOPSIS
#include <MS/user.h>

ProcId aProcId(aNodeId, aGroupId, aMemberId)
NodeId aNodeId;
GroupId aGroupId;
MemberId aMemberId;

DESCRIPTION
aProcId constructs a Message System process identifier (ProcId) with node value aNodeId, group value aGroupId, and member value aMemberId.

RETURN VALUE
aProcId returns the constructed value.

ERRORS
No errors are reported. ms_errno is not changed.

SEE ALSO
intro(3MS), bad_ProcId(3MS), CompProcId(3MS), MReceive(3MS)
NAME
bad_ProcId — check a Message System process identifier for (syntactic) validity

SYNOPSIS
#include <MS/user.h>

int bad_ProcId(id)
ProcId id;

DESCRIPTION
bad_ProcId determines whether id is a valid Message System process identifier (or process selector), i.e., are the values of its fields within the valid ranges. The function does not determine whether a process with that ProcId is currently signed onto the Message System.

RETURN VALUE
If the argument is a syntactically valid ProcId, bad_ProcId returns 0; else it returns a nonzero error code. The value of ms_errno is not changed.

ERRORS
The value returned by bad_ProcId is one of the following if its argument is invalid:

[eBadNode] The node identifier is outside of the valid range.
[eBadGroup] The group identifier is outside of the valid range.
[eBadMember] The member identifier is outside of the valid range.

SEE ALSO
intro(3MS), aProcId(3MS), CompProcId(3MS)
NAME
CompProcId — compare two Message System process identifiers

SYNOPSIS
#include <MS/user.h>

    int CompProcId (id1, id2)
    ProcId id1;
    ProcId id2;

DESCRIPTION
CompProcId compares the two Message System process identifiers id1 and id2. The comparison is done lexicographically, comparing first on the node components, then the group, and finally the member. Components are compared using the normal integer ordering.

RETURN VALUE
CompProcId returns a negative value (constant LESS) if id1 < id2, zero (EQUAL) if id1 = id2, and a positive value (GREATER) if id1 > id2.

ERRORS
No errors are reported. msg_errno is not changed.

SEE ALSO
intro(3MS), aProcId(3MS), bad_ProcId(3MS)
CurMsg — access the last message received via the Message System

SYNOPSIS
#include <MS/user.h>

int MSource(FromId)
ProcId *FromId;

int MDest(ToId)
ProcId *ToId;

int MNumKeys(Nkeys)
Counter *Nkeys;

int MKey(KeyNum, KeyVal)
Counter KeyNum;
MsgKey *KeyVal;

int MLenData(Nbytes)
Counter *Nbytes;

int MDataBlock(MaxBlock, Block)
Counter MaxBlock;
char *Block;

int MClrCur()

DESCRIPTION
These routines access the data stored in the Current_Input_Message buffer by MReceive(3).

MSource retrieves the process id of the source of the message.

MDest retrieves the process id of the destination for the current message.

MNumKeys retrieves the number of keys in the current message.

MKey retrieves the key in position KeyNum of the KeyArray of the current message.

MLenData retrieves the length of the data field (in bytes) of the current message.

MDataBlock retrieves the first min(MaxBlock,LenData) bytes of the data field of the current message.

MClrCur clears the Current_Input_Message buffer.

RETURN VALUE
Upon successful completion, each routine returns a value of SUCCESS. Otherwise, a value of FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
The above routines will fail if the following occur:
[eNoCurMsg] Current input message buffer is empty.
[eKeyNotDefined] Requested message key not defined in current message (MKey only).

SEE ALSO
intro(3MS), MReceive(3MS), MSignOn(3MS)
NAME
GetInvoker — get process id of invoking process

SYNOPSIS
#include <MS/user.h>

int GetInvoker(id)
ProcId *id;

DESCRIPTION
GetInvoker retrieves the process id of the process that requested the startup of the calling process via the Message System.

RETURN VALUE
Upon successful completion of GetInvoker, a value of SUCCESS is returned. Otherwise, a value of FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
GetInvoker will fail if the following occurs:
[msNotRemStart] The calling process was not remotely started via the message system.

SEE ALSO
intro(3MS), StartProcess(3MS)
NAME
ListProc — list processes in the Message System

SYNOPSIS
#include <MS/user.h>

int ListProc(aProcSel, MaxCount, Count, procList)
ProcSel aProcSel;
Counter MaxCount;
Counter *Count;
ProcId *procList;

DESCRIPTION
ListProc produces a list of processes whose ids match aProcSel. (The node component of aProcSel must be fully specified, i.e., it cannot be the wildcard value DONT_CARE.) The ids are placed into the array procList; at most MaxCount ids are recorded in the list. Count records the actual number of processes found that match the given specification.

RETURN VALUE
Upon successful completion of ListProc, a value of SUCCESS is returned. Otherwise, a value of FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
Any error returned by MSend, MReceive, MDataBlock, or toNodeName may be returned by ListProc. The following errors may also occur:

[eProcSel]  Invalid process selector.
[eBadNode]  Invalid node for request.
[eNoMsgs]   The response reception timed out.

SEE ALSO
intro(3MS), CurMsg(3MS), MReceive(3MS), MSend(3MS), toNodeName(3MS), toUnixId(3MS)
NAME
MReceive — receive a message through Message System
MCheck — check if any messages are awaiting delivery
MCount — count the number of waiting messages

SYNOPSIS
#include <MS/user.h>

int MReceive(TimeOut, FromSel, NKeySels, KeySel)
Counter TimeOut;
ProcSel FromSel;
Counter NKeySels;
MsgKeySel *KeySel;

int MCheck(TimeOut, FromSel, NKeySels, KeySel)
Counter TimeOut;
ProcSel FromSel;
Counter NKeySels;
MsgKeySel *KeySel;

int MCount(msgCount, FromSel, NKeySels, KeySel)
Counter *msgCount;
ProcSel FromSel;
Counter NKeySels;
MsgKeySel *KeySel;

DESCRIPTION
MReceive receives a queued message from another process, dequeues the message, and places it into the calling process' Current_Input_Message buffer. (See CurMsg(3MS) for details of accessing information in this buffer.)

To be received, a message must be acceptable to all selectors specified in this call, i.e., satisfies the conjunction of the process selector and the message key selectors.

MCheck checks for the availability of a queued message without receiving the message.

MCount counts the number of queued messages that satisfy the selectors.

TimeOut is the integer number of seconds to wait for satisfaction of the request. A value of DONT_WAIT causes an error return if the receive cannot be satisfied immediately; a value of WAIT_FOREVER denotes an indefinite wait. Upon expiration of the timeout period an error value is returned.

FromSel is a process selector identifying an acceptable sending process of a message. A value of DONT_CARE for any component of FromSel means any valid value will be accepted in that component. (The constant process selector ANYPROC, which has DONT_CARE for all components, is provided for convenience.)

NKeySels is the number of values in the KeySel array, must be between 0 and MAXnumKeys, inclusive.

KeySel is an array of message key selectors identifying acceptable values for the KeyArray component of a message. Before a message can be accepted, all of the elements of KeySel must be satisfied. If the message's KeyArray has more elements than KeySel, then the additional key fields are ignored. If NKeySels is 0, then this argument may be NULL. (See aKeySel(3MS) for more information on message key selectors.)

MsgCount is number of queued messages that satisfy the selectors.
RETURN VALUE
Upon successful completion of \texttt{MReceive}, \texttt{MCheck}, or \texttt{MCount}, a value of \texttt{SUCCESS} is returned. Otherwise, a value of \texttt{FAILURE} is returned and \texttt{ms_errno} is set to indicate the error.

ERRORS
\texttt{MReceive}, \texttt{MCheck}, and \texttt{MCount} will fail if any of the following occur:

- \texttt{[eProcSel]} The process selector was invalid.
- \texttt{[eMaxKeys]} Message has too many keys.
- \texttt{[eBadTimeOut]} Invalid value specified for timeout.
- \texttt{[eNoSignOn]} Process not signed on to message system.
- \texttt{[eNoMsgs]} No satisfactory messages available.

SEE ALSO
\texttt{intro(3MS), aKeySel(3MS), aProcId(3MS), CurMsg(3MS), MSend(3MS)}
NAME

MSend — send a message through Message System

SYNOPSIS

#include <MS/user.h>

int MSend(Dest, LenData, DataBlock, NumKeys, KeyArray)
ProcId Dest;
Counter LenData;
char *DataBlock;
Counter NumKeys;
MsgKey *KeyArray;

DESCRIPTION

MSend constructs a message from the given arguments and sends it to the message queue for
process Dest. The destination process may dequeue the message with a subsequent call of
function MReceive. (Note: if the member component of Dest is DONT_CARE, then the message
is delivered to any one of the processes on the specified node and group.)

LenData is the number of bytes in the data portion of the message, while DataBlock is the
actual data. LenData must be between 0 and the defined constant MAXLenData; if LenData is 0,
argument DataBlock may be NULL.

NumKeys is the number of keys to be attached to the message, while KeyArray is the array of
keys. NumKeys must be between 0 and the defined constant MAXnumKeys; if NumKeys is 0,
argument KeyArray may be NULL.

RETURN VALUE

Upon successful completion of MSend, a value of SUCCESS is returned. Otherwise, a value of
FAILURE is returned and me_errno is set to indicate the error.

ERRORS

In the following list, errors marked with a star can only occur if the destination process is on
the same host as the calling process. MSend will fail if any of the following occur:

[eNoSignOn] — The process is not signed on.
[eProcId] — The destination ProcId is invalid.
[eMaxKeys] — The indicated NumKeys is greater than the system-imposed maximum.
[eMaxData] — The indicated LenData is greater than the system-imposed maximum.
[eSndHost] — The destination process host number is not valid.
[eSndProc]* — The destination process is not known.
[eSndQFull]* — The message could not be added to the destination process' message queue.
[eFailNotify]* — The destination process could not be sent the signal indicating a message was
available.

SEE ALSO

intro(3MS), MReceive(3MS)
NAME
  merror, ms_errno - Message System error messages

SYNOPSIS
  #include <MS/user.h>

  merror(str)
  char *str;

  int ms_errno;

DESCRIPTION
  merror produces a short error message on the standard error file describing the last error
  encountered during a call to the message system. First the argument string str is printed, then
  a colon, then the name of the error in square brackets, and finally the message associated with
  the error and a newline. The error is obtained from the global variable ms_errno, which is set
  upon encountering any error by the message system routines.

SEE ALSO
  intro(3MS), perror(3C), stdio(3S)
NAME
MSignOn, MSignOff, MSexit — enter or leave Message System

SYNOPSIS
#include <MS/user.h>

int MSignOn(group)
  GroupId group;

int MSignOff();

void MSexit(n)
  int n;

DESCRIPTION
MSignOn signs the calling process onto the Message System as a member of the given group. If
MSignOn is successful, the calling process receives a new ProcId and a message queue is created.
If the process is already signed on, MSignOn returns an error.

MSignOff removes the calling process from the Message System. Any messages queued for the
process are destroyed, its queue is deallocated, and its ProcId is released. After signing off, a
process may sign on again with another call to MSignOn.

MSexit is equivalent to MSignOff followed by exit(n).

RETURN VALUE
Upon successful completion of MSignOn or MSignOff, a value of SUCCESS is returned.
Otherwise, a value of FAILURE is returned and ms_errno is set to indicate the error. MSexit
does not return.

ERRORS
MSignOn will fail if any of the following occur:
[eMultiSignOn] The process is already signed on.
[eBadGroup] The given GroupId is outside the legal range of values.
[eProcTableFull] The Message System process table is full.
[eSharedMem] A shared memory lookup or attach failed.
[eSemaphore] A semaphore lookup failed.

MSignOff will fail if any of the following occur:
[eNoSignOn] The process is not signed on.
[eProcNotFound] The caller’s ProcId was not found in the Message System process table.

SEE ALSO
intro(3MS), MyProcId(3MS), exit(2)
NAME
MyProcId — return the Message System process identifier for the calling process

SYNOPSIS
#include <MS/user.h>

    int MyProcId(procId)
    ProcId *procId;

DESCRIPTION
MyProcId returns the calling process' Message System process identifier (ProcId) in argument procId.

RETURN VALUE
Upon successful completion of MyProcId, a value of SUCCESS is returned. Otherwise, a value of FAILURE is returned and MS_errno is set to indicate the error.

ERRORS
MyProcId will fail if one of the following occurs:
-[eNoSignOn] The process is not signed onto the Message System.

SEE ALSO
intro(3MS), MSignOn(3MS)
NAME
StartProcess — use Message System to start execution of a process

SYNOPSIS
#include <MS/user.h>

int StartProcess(node, prog, arglist, r_env, r_in, r_out, r_err)
NodeId node;
char *prog;
char *arglist;
char **r_env;
char *r_in, *r_out, *r_err;

DESCRIPTION
StartProcess initiates the execution of another process. This process may be started on any
node in the Message System, including the calling process' node. No confirmation of the remote
startup is given. (Such responsibilities are left to the application processes.)

The remote process is specified by the node identifier, node, of the machine on which it will
reside and the pathname, prog, of the program to execute. The pathname may either be a full
pathname (i.e., beginning with "/") or a partial pathname that will be prefixed by the default
location for processes under the Message System. Note that prog must be a pathname on node,
the executable file is not copied there by StartProcess.

arglist is a null terminated character string that will be parsed and then passed to the newly
created process as though it was the argument list on a shell-level command.

r_env is a pointer to an array of character pointers specifying the strings to be used as the
environment of the created process. If r_env is NULL, then the environment of the calling
process will be used as the environment on the new process. Note: Additional strings are added
to the environment of the started process to pass various information to it from the Message
System. At present, only the id of the process that requested the startup is included in this
manner.

r_in, r_out, and r_err are all null-terminated character strings that specify where the standard
file descriptors stdin, stdout, and stderr, respectively, are to be directed for the created process.
Any string that is empty, or found to be invalid at the remote node, will be replaced by
"/dev/null".

Note: To use the Message System, the remotely started process must sign onto the Message
System like any other process.

RETURN VALUE
StartProcess returns a value of SUCCESS if the startup request is successfully sent to the
Message System. Otherwise, a value of FAILURE is returned and ms_errno is set to indicate
the reason the send failed (see MSend(3MS)).

ERRORS
Any error returned by MSend may be returned by StartProcess.

SEE ALSO
intro(3MS), GetInvoker(3MS), MSend(3MS), MSignOn(3MS), environ(2), exec(2), stdio(3S)
NAME
toNodeId - convert a node name string into a Message System node identifier

SYNOPSIS
#include <MS/user.h>

    int toNodeId(nodename, nodeid)
    char *nodename;
    NodeId *nodeid;

DESCRIPTION
toNodeId returns, in argument nodeid, the Message System node identifier associated with the
node name string nodename.

RETURN VALUE
Upon successful completion of toNodeId, a value of SUCCESS is returned. Otherwise, a value of
FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
toNodeId will fail if one of the following occurs:

    [eNoConfig] The node configuration file cannot be accessed.
    [enodeName] Unknown node name.

SEE ALSO
    intro(3MS), MyProcId(3MS), toNodeName(3MS)
NAME
toNodeName — convert a Message System node identifier into a node name string

SYNOPSIS
#include <MS/user.h>

int toNodeName(nodeid,nodename)
NodeId nodeid;
char *nodename;

DESCRIPTION
toNodeName returns, in argument nodename, the UNIX node name string associated with the
Message System node identifier nodeid.

RETURN VALUE
Upon successful completion of toNodeName, a value of SUCCESS is returned. Otherwise, a
value of FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
toNodeName will fail if one of the following occurs:
[eNoConfig] The node configuration file cannot be accessed.
[eBadNode] Invalid node identifier.

SEE ALSO
intro(3MS), MyProcId(3MS), toNodeId(3MS), uname(2)
NAME
toProcId — converts a Message System process identifier to a UNIX process identifier

SYNOPSIS
#include <MS/user.h>

int toProcId(aUnixId, theProcId)
UnixId aUnixId;
ProcId *theProcId;

DESCRIPTION
toProcId returns, in argument theProcId, the Message System process identifier associated with
the process having UNIX process identifier aUnixId.

RETURN VALUE
Upon successful completion of toProcId, a value of SUCCESS is returned. Otherwise, a value of
FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
toProcId will fail if any of the following occur:
[enoSignOn] Process not signed on to message system.

SEE ALSO
intro(3MS), MyProcId(3MS), toUnixId(3MS)
NAME
toUnixId — convert a Message System process identifier to a UNIX process identifier

SYNOPSIS
#include <MS/user.h>

int toUnixId(aProcId, theUnixId)
    ProcId aProcId;
    UnixId *theUnixId;

DESCRIPTION
toUnixId returns, in argument theUnixId, the UNIX process identifier associated with the process
having Message System process identifier aProcId.

RETURN VALUE
Upon successful completion of toUnixId, a value of SUCCESS is returned. Otherwise, a value of
FAILURE is returned and ms_errno is set to indicate the error.

ERRORS
toUnixId will fail if any of the following occurs:

[aNoSignOn] Process not signed onto message system.
[aProcId] The aProcId argument is invalid.

SEE ALSO
intro(3MS), toProcId(3MS), getpid(2)
NAME

UniqKey — generate a Message System-wide unique message key

SYNOPSIS

#include <MS/user.h>

int UniqKey(NewKey)
MsgKey *NewKey;

DESCRIPTION

UniqKey generates a network-wide unique message key, NewKey.

RETURN VALUE

Upon successful completion of UniqKey, a value of SUCCESS is returned. Otherwise, a value of
FAILURE is returned and ma_errno is set to indicate the error.

ERRORS

UniqKey will fail if any of the following occur:

[eNoSignOn] The process is not signed on.

SEE ALSO

intro(3MS), MReceive(3MS), MSend(3MS)