A Guide to Programming with MP
Version 1.1.2

Simon Gray

Department of Mathematics and Computer Science
Kent State University
Kent, OH 44242-4001, USA
sgray@mcs.kent.edu

June 9, 1997

Abstract

This report describes a C implementation of version 1.1.2 of the Multi Protocol (MP) [7] [8] and is intended as a guide for programmers wishing to use MP in their own applications. MP is designed for integrating symbolic, numeric, graphical, document processing, and other software tools for scientific computation in different computing paradigms (point to point, distributed, parallel, software bus). The protocol specifies a set of basic types, a mechanism (annotated parse trees) for sending structured data (such as expression trees or arrays, lists, etc.), annotations to provide supplemental information, and various optimization techniques. In a layer above the data exchange portion of the protocol, MP supports collections of definitions for mathematical symbols (operators and symbolic constants) and annotations in *dictionaries*. The C implementation defines a mapping of MP data types to C data types and a set of routines for transmitting data and performing some standard type conversions. MP is independent of the mechanism used to deliver the data. The current implementation works with files, sockets, ToolBus, and PVM. Examples illustrating how a sender and receiver can be connected and exchange annotated parse trees are included.

MP version 1.1.2: Multi Protocol
Kent State University, Kent, OH
Authors: S. Gray, N. Kajler, P. Wang

NOTICE

Independent packages used by MP have their own copyright:

- GNU MP: multiple precision integer and real arithmetic in MP is provided by the GNU Multi Precision library under the GNU General Public License. GMP is available from `prep.ai.mit.edu` and many other ftp servers as gmp-2.0.2.tar.gz or the latest version.
- XDR: eXternal Data Representation software, Copyright (C) 1984, Sun Microsystems, Inc.

MP is available via anonymous ftp from `ftp.mcs.kent.edu:/dist/MP/`. The latest information on MP is always available from `http://SymbolicNet.mcs.kent.edu/areas/protocols/mp.html`. If you obtain MP, please send a short note containing your full name and address to `mp@mcs.kent.edu`. This allows us to keep track of MP's users and to notify you of new versions, bug fixes, updated documentation, etc. Please send any comments or bug reports to `mp@mcs.kent.edu`.

If you use MP as part of a project and/or publish a result that was partly obtained using MP, please cite MP. We would also greatly appreciate it if you would inform us about such a project/paper.

MP has received support from a number of sources, including ARO, NSF, and ICM/Kent.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document describes libMP, a library of C routines implementing an Application Programmer Interface (API) for the Multi Protocol (MP) specification. This chapter gives an overview of MP. The basics of creating a connection between independent packages are covered in chapter 2 and the basics of sending and receiving data as MP Trees are presented in chapter 3. Advanced topics are discussed in chapter 4.

1.1 Motivation and Overview

There is a growing body of mathematical software tools to aid scientists and engineers in finding solution(s) to problems. These tools may be general purpose such as Maple and Mathematica for computer algebra or specialized such as Camal for general relativity and Singular for commutative algebra and algebraic geometry. For the most part, these tools are standalone and offer little support for interaction with other tools. Problem-solving, however, frequently requires the services of tools with different areas of expertise. The Multi Protocol is intended to support the efficient exchange of "scientific" data in a system that provides interactions among a collection of scientific tools whose data needs are diverse: symbolic, numeric, and character-based.

The Multi Project at Kent State is part of an ongoing research effort into the integration of software tools for scientific computing. Three dimensions of the problem of tool integration are readily identifiable: data integration, control integration, and user interface integration [10, 11]. Data integration involves the exchange of data between separate tools, including the definition of a mechanism allowing the tools to agree on the format and meaning of the transmitted data. Control integration concerns the establishment, management, and coordination of inter-tool communications. Finally, the aim of user interface integration is to provide the user with a logical and consistent style of interaction with the entire system. A key philosophy of our project has been to view the dimensions of tool integration as *individual* problems to be solved *separately*, recognizing that each of these problems may have several solutions. The focus of our first efforts has been on data integration and has produced the Multi Protocol, a specification for encoding mathematical expressions for efficient communication among scientific computing systems. A library of C routines based on the MP specification has been implemented and is described here.

The major design and implementation decisions made in MP are given in [6] and [8] (respectively versions 0.5 and 1.0). MP defines a set of basic types and a mechanism for constructing structured data. Numeric data (fixed and arbitrary precision floats and integers) are transmitted in a binary format (2's complement, IEEE float, etc.). Expressions are represented as linearized, annotated parse trees (MP Trees) and are transmitted as a sequence of *node* and *annotation packets*, where each node packet transmits a node from the expression's parse tree. The node packet has fields giving the *type* of the data carried in the packet, the number of arguments (for operators) that follow, the number of annotations, some semantic information, and the data. Operators and constants which occur frequently have an optimized encoding and are known as "common". Various optimizations are supported to minimize the overhead incurred by the node packets, allowing the sender to transmit blocks containing "raw" (untyped) data. A datum sent without any type information attached is called a *data limb*.

Annotations efficiently carry additional information which is either supplementary and can be safely ignored by the receiver, or contains information essential to the proper decoding of the data. Each annotation carries a set of flags which tell the receiver whether it can safely ignore the annotation's content or not.

In a layer above the data exchange portion of the protocol, MP supports collections of definitions for annotations and mathematical symbols (operators and symbolic constants) in *dictionaries*. Dictionaries address the problem of application heterogeneity by supplying a standardized representation and semantics for mathematical objects. They are identified within packets through a dictionary tag field. Applications that communicate according to definitions provided in dictionaries do not need to have direct knowledge of each other, promoting a "plug-and-play" style of interoperation at the *application* level. A dictionary for polynomials has been built on top of MP in this manner [1]. Another, more general, library for manipulating MP Trees is also available [2].

Figure 1.1 illustrates the correspondence between an expression to be communicated and its abstract syntax tree (subscripted material represents an annotation).

$$\left(\frac{a(x2 - x1)}{x2 - x1} \right)_{\text{(comment gradient)}}$$

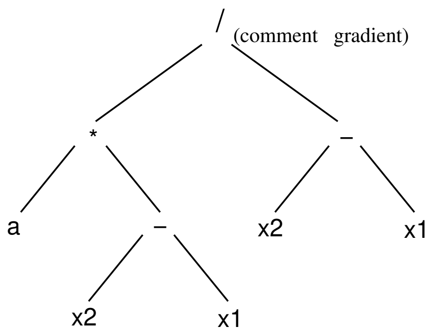

Figure 1.1: Sample expression

1.2 Communication Between Applications

It is not possible to anticipate all the scenarios in which someone might want to transmit mathematical data, so the implementation of MP is independent of the underlying communication mechanism (referred to here as a "device") used to transmit data between applications. Instead, MP data is written to and read from buffers attached to a *link*. A link is created within an *environment* that defines and provides the resources to a link, including interfaces to the supported transport devices. Data is moved between MP buffer-space and the transport device through routines provided by an abstract device interface. This approach provides a great deal of flexibility, allowing researchers to provide the kind of communication most appropriate for the task at hand and also allowing new communication packages to be adopted with minimum effort. Currently there is support for communication using files, TCP sockets, ToolBus [4] (a software bus package), and PVM [12] (a package for parallel distributed computation). Users can write their own device interfaces to support other transports which can be added to an environment and then used to open a link. It is also possible to send and receive MP encoded expressions using MIME [5].

1.3 The Structure of libMP

The MP library, libMP, is implemented in a layered fashion as shown in figure 1.2. The highest level provides routines for sending and receiving data in *packets*. The level below that provides routines for communicating *data limbs*, i.e. data that has no extra overhead and which can be used to construct and send (or receive and deconstruct) objects in a piecemeal fashion (that is, the packet header and packet body can be processed separately). This is especially useful on the receiving side when the type of the incoming packet may not be known in advance and it is necessary to retrieve the packet header independently of the packet body. The buffering level knows nothing about packets or annotated trees. It simply provides a buffer where packets can be written (sending side) and read (receiving side), routines to put/get basic data elements, and access to an underlying device for transmitting the data (the device level).

Packet level	MP_Put<*type*>Packet() MP_Get<*type*>Packet()
Data level	IMP_Put<*type*>() IMP_Get<*type*>()
Byte level	IMP_PutLong() IMP_PutBytes() IMP_GetLong() IMP_GetBytes()
Buffer level	Abstract Device Interface
Device level	data delivery mechanism (file, socket, ToolBus, PVM, ...)

Figure 1.2: MP implementation layers

1.4 Conventions

Within this document the typewriter font is used for code, data types, and routines.

Within the code the following conventions are used:

- all MP-related data types and constants begin with MP_
- subwords are distinguished by capitalizing the first letter of the subword
  Example: MP_PutAnnotationPacket()
  Example: MP_Uint32Type
- typedefs for static types end in _t
  Example: MP_Link_t
- typedefs for pointer types end in _pt
  Example: MP_Link_pt
- top-level MP routines and node packets for an MP type begin with MP_
  Example: MP_PutSint32Packet()
  Example: MP_Sint32
- middle-level MP routines and data limbs for an MP typebegin with IMP_
  Example: IMP_PutSint32()
  Example: IMP_Sint32
- several of the MP types have abbreviated names:
  - Cmt – Common Meta Type
  - Cop – Common Operator
  - Mop – Meta Operator

  - `Cmop` – Common Meta Operator

- A C define that identifies the definition of symbolic constants, operators, annotations, meta types and meta operators given in a dictionary use the following convention: MP_<type><dictionary><value>. For example, `MP_CmtProtoIMP_Sint32` specifies the Common Meta Type (Cmt) definition given in the prototype dictionary (Proto) for a signed 32-bit integer data limb (IMP_Sint32).

- The values for `MP_Success` and `MP_Failure` are consistent with C's encoding of true and false, allowing the programmer to test for success/failure in the usual manner. For example,

```
if (!MP_PutSint32Packet(link, n, 0))
  MP_PrintError(link);
```

Portions of the MP API are actually macros containing calls to functions. These macros are defined in the relevant header files.

1.5 Reporting Results

As the library and the protocol are still experimental, we would greatly appreciate learning of your experiences: how you used the library, what you found worthwhile, what you had to change, any bugs you found. Please send bug reports, inquiries, and experiences to `mp@mcs.kent.edu`.

When reporting bugs, please supply as much information as possible, including the architecture, operating system, and compiler used, output from the MP log file, and any other useful information you can think of. In cases like this, more detail is better than less.

If you use MP as part of a significant project and/or publish a result that was partly obtained using MP, please cite MP. We would greatly appreciate it if you could inform us about such a project or paper. Updates on MP are available from `http://SymbolicNet.mcs.kent.edu/areas/protocols/mp.html`.

Chapter 2

Making a Connection

A *link* is simply an endpoint for communication. It may be connected to a file (unidirectional data transfer) or to links in other applications (bidirectional data transfer). The steps to send/receive MP data via a link are

1. Initialize an MP environment (`MP_InitializeEnv()`)
2. Open a link for sending/receiving data (`MP_OpenLink()`)
3. Process data (e.g., make requests/receive results (sender/client) – handle requests (receiver/server))
4. Close the link (`MP_CloseLink()`)
5. Release the environment (`MP_ReleaseEnv()`)

2.1 The MP Environment

The MP environment maintains a set of resources to be shared by a collection of links created within an application. Typically an application will need only a single environment. Multiple environments would only be necessary if an application needs to support links that have different buffering requirements or different arbitrary precision number formats (see chapter 4 on Advanced Topics).

```
MP_Env_pt MP_AllocateEnv (void)
MP_Env_pt MP_InitializeEnv (MP_Env_pt env)
void      MP_ReleaseEnv (MP_Env_pt env)
```

Table 2.1: MP environment routines

An environment may be allocated in two ways. First, through a call to `MP_AllocateEnv()`. If the call fails, NULL is returned, otherwise a pointer to a partially initialized environment structure (`MP_Env_pt`) is returned. Environment options may be reset at this point (see section 4.1 for more details). This must be followed by a call to `MP_InitializeEnv()` to complete the initialization. If the argument to `MP_InitializeEnv()` is NULL, an environment will be allocated *and* completely initialized. Alternatively, if the argument is not NULL, it is assumed that the argument was previously allocated through a call to `MP_AllocateEnv()`. In either case, the routine returns a pointer to a completely initialized environment structure on success, and NULL on failure. If enabled, a log file will be opened as part of the creation of an environment. Initially, the log will contain information about the environment's characteristics (see section 4.2.2 for more information on logging). An environment is released with `MP_ReleaseEnv()`. There should be no further activity on any links associated with an environment that has been released.

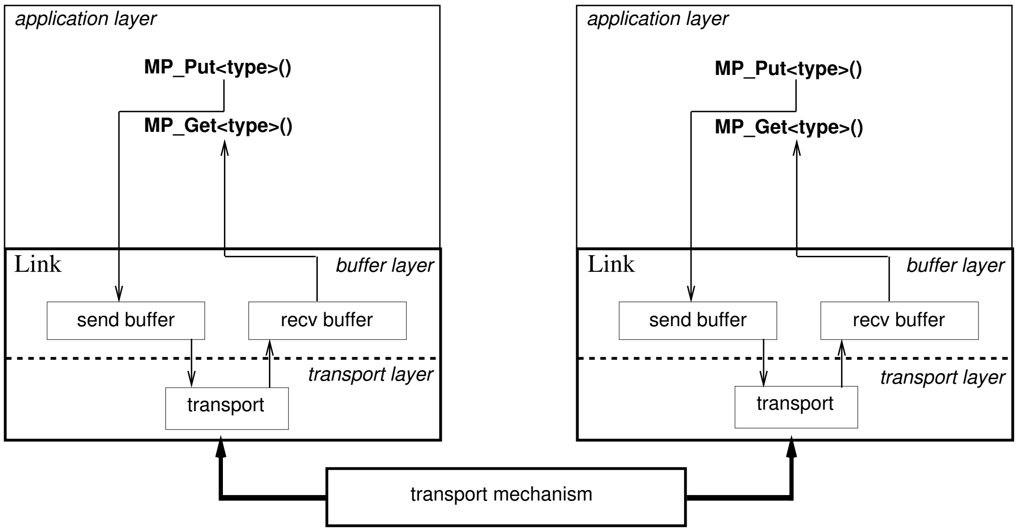

Figure 2.1: Link connection organization

2.2 MP Links

Abstractly, a link is an endpoint for communication with other applications. An application sends data to and receives data from a link. As illustrated in figure 2.1, a link has two layers: a buffering layer which contains a set of send and receive buffers, and a transport layer that manages communication with the underlying transport device. The buffers are implemented as a linked list of buffer segments. The application programmer will have no direct interaction with the transport layer and is only involved in transports at the time a link is created and a specific transport device is bound to the link. Similarly, the application programmer's only concern with the buffering layer is to customize the size of the buffers or to change the time at which data is moved between transport buffer space and the MP buffers. Typically the programmer will use the default values.

The transport device is responsible for the actual transmission of data between applications. MP assumes as little as possible about the exact nature of this transport, preferring to support a diverse collection of data delivery devices. In this way, a variety of communication paradigms can be used to send MP data. Currently files, TCP sockets, ToolBus, and PVM are supported.[1] At the time a link is "opened", a particular transport device is bound to the link. Other transports may be used by first adding them to the environment (see section 4.1.2 in the chapter on Advanced Topics).

Information is exchanged between applications in the form of *messages*. A message is composed of one or more *message fragments* , where each fragment corresponds to the data in a single buffer segment. The buffering layer lies between the MP application interface and the transport device and is where messages are assembled and disassembled. The programmer's view is that each *put* and *get*, whether at the node or data packet level, accesses a link connecting the application with one or more other applications, as determined by the underlying transport device. In reality, all accesses are to one of two buffers: *put*s access a link's send buffer and *get*s the receive buffer. The contents of a buffer are written to the underlying data delivery device by the transport-specific write routine. Similarly, a transport-specific read routine is responsible for loading data from the transport device into the link's receive buffer.

A link is created and initialized with a call to `MP_OpenLink()` which returns a pointer to an MP link structure (`MP_Link_pt`) on success and NULL on failure. The link inherits attributes from an MP environment.

[1] The FILE and TCP devices are automatically included in the MP library. The PVM and TB devices must be added to an environment at execution time with a call to `MP_AddEnvTranspDevice()` before a PVM- or TB-link can be opened. See section 4.1.2 for information on adding devices to an environment and the INSTALL file in the MP distribution for instructions on compiling the necessary support routines.

```
MP_Link_pt MP_OpenLink  (MP_Env_pt env, int argc, char **argv)
void       MP_CloseLink (MP_Link_pt link)
```

Table 2.2: MP link routines

Some of these link attributes are resettable by the user. Remember that a call to `MP_InitializeEnv()` must have been made *prior* to any call to `MP_OpenLink()`.

Arguments passed in the `argv` vector identify the transport device to use, what mode to open it in, etc. These are shown in table 2.3. The arguments may either be given as options on the command line when the application is launched or hardcoded within the application.

The following examples illustrate setting the arguments from the command line. The arguments “-infile" and “-outfile" are needed by the example applications and are not required by `MP_OpenLink()`. The first example starts application `recv-tree` which will open a link using the TCP transport device and launch a remote application, `send-tree`, on host eeyore with argument "-infile ex1.dat".

```
pooh{12}%recv-tree -MPtransp TCP -MPmode launch -MPapplication "send-tree
-infile ex1.dat" -MPhost eeyore
```

To manually establish a connection between two applications on separate machines, you must start one in “listen” mode and *then* its partner in “connect” mode. For example:

```
pooh{13}%recv-tree -MPtransp TCP -MPmode listen -MPport 1045 -outfile
ex1.out
```

On host Pooh, application `recv-tree` opens a TCP link that will listen for connections on port 1045. Now we run `send-tree` which also opens a TCP link which will attempt to connect to port 1045 on Pooh.

```
owl{25}%send-tree -MPtransp TCP -MPmode connect -MPhost pooh -MPport 1045
-infile ex1.dat
```

To open a link to write to a file:

```
owl{2}%send-tree -MPtransp FILE -MPmode write -MPfile ex1.mp-infile ex1.dat
```

To open a link to read from a file:

```
owl{3}%recv-tree -MPtransp FILE -MPmode read -MPfile ex1.mp-outfile ex1.out
```

Option	Legal Values	Used with
-MPtransp	FILE, TCP, PVM, TB	always
-MPmode	listen, connect, launch, fork	-MPtransp TCP
	read, write	-MPtransp FILE
-MPapplication	< application name >	-MPmode launch
-MPhost	< hostname >	-MPmode launch
	< hostname >	-MPmode connect
-MPport	< port number >	-MPmode connect
	< port number >	-MPmode listen
-MPfile	< filename >	-MPtransp FILE

Table 2.3: Arguments to `MP_OpenLink()`

Here we use a `.mp` extension to indicate a file containing MP data, but this is not necessary. Alternatively, one might hardcode the arguments within the body of the program as follows:

```
char *argvec[]={"send-tree", "-MPtransp", "FILE", "-MPmode", "write",
               "-MPfile", "MP-ex1.mp", "-infile", "ex1.dat", NULL};

  env = MP_InitializeEnv(NULL);

  if ((link = MP_OpenLink(env, 9, argvec)) == NULL) {
    MP_ReleaseEnv(env);
    exit(-1);
    }
  . . .
```

There are several things to note about table 2.3.

1. Launching remote applications is done using the remote shell command, so the appropriate `rsh/remsh` conventions must be observed. In paricular, the `.rhosts` file has to have the appropriate entries – see the man page for `rsh` (or, `remsh`) for details.

2. The argument to `-MPapplication` must be given in double quotes (`" "`) so that it is treated as a single argument to the remote application and not additional options to the launching application. This argument will be passed to rsh, so must include the name of the remote application to be launched as well as any *additional* command line options you want to provide the remote application. The MP software will add other command line options telling the remote application to which port and on which host to make the TCP connection. More precisely, the remote application will be called with the additional command line arguments:
   `-MPtransp TCP -MPmode connect -MPhost` <hostname> `-MPport` <port number>
   where <hostname> is replaced by the name of the host on which the client is running and <port number> is replaced by the port number at which the client is "listening". An application should pass these arguments to the call to `MP_OpenLink()`.

3. When using a port number in connect mode, it is assumed that an application is *already* running on the given host and is listening on the specified port. Opening a link in connect mode is a blocking operation, i.e., control will resume to the program only after the connection has been established.

4. For listen mode, an attempt will be made to use the port specified, but if it is already in use, the next available port will be used. If events are being logged, the port number can be taken from the log file once the application has successfully bound to a port number.

5. For fork mode, the running program is forked, and a TCP connection is established with the forked child. If the connection fails, the forked child exits and the `NULL` link is returned to the parent. The command `MP_GetLinkStatus()` with the argument `MP_LinkIsParent` (resp. `MP_LinkIsChild`) can be used to determine which side of the link is the parent (resp. child) side.

6. The order of the arguments does not matter.

Also, note in the hardcoded example above that the name of the application (`send-tree`) is given as the first entry in `argvec`. This is not necessary and is included only for logging purposes (as part of `MP_OpenLink()`'s initialization sequence).

A link is closed with a call to `MP_CloseLink()`, after which data may not be written to, nor read from, the link.

2.3 Example: link communication via the FILE and TCP devices

When communicating using files, the sending side must open the file for writing and the receiving side for reading. The communicating programs have the responsibility of coordinating access to the file.

When using TCP sockets in launch mode, the connection is established for you. The alternative is to have the two communicating applications use the connect and listen modes. One side must take the "server" role, creating a socket and listening for a request to connect, while the other side must take the role of a "client", creating a socket, identifying the address of the "server", and making the connection. Note that the listening application should be running (and listening) *before* the connecting application is run.

The example code below may be used with the FILE transport or the TCP transport. The applications use command line arguments to determine the particular transport. These programs may be found in the `examples` directory of the MP source distribution. The steps indicated in the comments correspond to the outline presented at the beginning of this chapter.

```
/* send-tree.c -
      read expressions in a textual, stylized MP format
      from a file and send them as MP Trees to the link */
#include "MP.h"

int main(int argc, char *argv[])
{
  char        *ifname = NULL;
  MP_Link_pt link    = NULL;
  MP_Env_pt  env     = NULL;

  if ((env = MP_InitializeEnv(NULL)) == NULL) {          /* STEP 1 */
    fprintf(stderr, "%s: MP_InitializeEnv() failed!\n", argv[0]);
    exit(1);
    }

  if ((link = MP_OpenLink(env, argc, argv)) == NULL) { /* STEP 2 */
    MP_ReleaseEnv(env);   exit(1);
    }

  if ((ifname = IMP_GetCmdlineArg(argc, argv, "-infile")) == NULL)  {
    MP_LogEvent(link, MP_ERROR_EVENT, "missing -infile argument");
    exit(1);
    }

  imp_file2tree(link, ifname);                             /* STEP 3 */

  MP_CloseLink(link);                                      /* STEP 4 */
  MP_ReleaseEnv(env);                                      /* STEP 5 */
  exit(0);
}
```

```
/* recv-tree.c - receive MP Trees from the link and write them
                 to a file in a stylized, textual format. */
#include "MP.h"
#include "node.c"
main(int argc, char *argv[])
{ char       *ofname = NULL;
  int        more = 0, n = -1;
  node_t     *root = NULL;
  MP_Link_pt link = NULL;
  MP_Env_pt  env  = NULL;

  if ((env = MP_InitializeEnv(NULL)) == NULL) exit(1);
  if ((link = MP_OpenLink(env, argc, argv)) == NULL) {
    MP_ReleaseEnv(env);  exit(1);
    }
  if ((ofname = IMP_GetCmdlineArg(argc, argv, "-outfile") == NULL) {
    MP_LogEvent(link, MP_ERROR_EVENT,
                "missing or bad -outfile argument\n");
    exit(1);
    }
  fd = fopen(ofname, "w");
  printf("server's annotated tree(s):\n");
  do {
    if (MP_TestEofMsg(link)) MP_SkipMsg(link);  /* msg alignment */
    printf("\n\n%-20s %-5s %-30s %-7s %-9s\n", "Type", "Dict",
            "Value", "#Annots", "#Children");
    root = m_get_node(link);
    more = process_request(root);
    free_tree(root);
    root = NULL;
    free_label_table();
  } while (more);
  fclose(fd);
  MP_CloseLink(link);
  MP_ReleaseEnv(env);
}
```

A sample call using launch mode looks like:

```
Pooh{16}% recv-tree -MPtransp TCP -MPmode launch -MPhost eeyore
-MPapplication "send-tree -infile expr"
```

Alternatively, using the listen and connect modes with the "server" running on Pooh and the "client" on Piglet, we would first start the server and have it listen for a connection:

```
Pooh{27}% recv-tree -MPtransp TCP -MPmode listen -MPport 1045 -outfile
results
```

Then we could start the "client" on another machine:

```
Piglet{57}% send-tree -MPtransp TCP -MPmode connect -MPhost Pooh -MPport
1045 -infile expr
```

Chapter 3

Sending & Receiving Data in MP

3.1 MP - C Data Types

Table 3.1 defines the mapping of MP basic types to C types. As indicated in the table, most MP types have natural counterparts in C. For the string-based types, the length can be discovered using normal string-handling operations. The size of the machine precision numeric types is generally determined by its type (e.g., MP_Sint32Type is a C long - four bytes)[1] The raw type, on the other hand, requires different treatment. Unlike the string-based types, the amount of data to be handled must be supplied by the application program. The put and get routines which handle the raw type take a length argument (in bytes) and a pointer to an array of char (uninterpreted bytes). The default representation and supporting code for arbitrary precision numbers are from the GNU Multiple Precision package. Resetting the large integer/real package is described in section 4.1.3. Fixed precision integers are transmitted in 2's complement form. The representation for fixed precision reals is quietly negotiated at link connection time, with preference given to the native format.[2] There should be no loss of precision when transmitting reals when two sides are using the native format, however occasionally one side may be forced to convert between its native format and the MP standard format (IEEE 754 [9]), in which case there may be some loss of precision. The log file will tell you which format is native and which format is actually used by the link.

3.2 Packets

Recall that data is sent in messages, where each message typically contains an MP Tree (a linearized version of the annotated parse tree). MP Trees are composed of a sequence of *node packets*, *annotation packets*, and, possibly, *data limbs*.

3.2.1 node packet

A node packet corresponds to a node in a parse tree. Interior nodes of the tree are always operators with arguments, while the leaves of the tree will be non-operator types or an operator without arguments. Each node packet has five fields:

node type Identifies the type of data carried in the node packet. This will always be an MP type as listed in the first column of table 3.1.

[1] At the moment, libMP is not 64-bit clean, i.e. will not work on machines for which `sizeof(long) == 8` (such as the DEC ALPHA). This will change in the next version of libMP.

[2] A limited number of formats are currently implemented. More will become available as MP is ported to machines using non-IEEE formats.

MP Basic Type	MP-C typedef	Description
MP_StringType		The library routines expect
MP_IdentifierType		C strings (null-terminated
MP_OperatorType		character strings) for these
MP_MetaType		types and will prepend the
MP_MetaOperatorType		4-byte length field.
MP_ConstantType		
MP_RawType		unsigned long int array of char
MP_BooleanType	MP_Boolean_t	unsigned char
MP_CommonMetaType	MP_Common_t	unsigned char
MP_CommonMetaOperatorType	MP_Common_t	unsigned char
MP_CommonOperatorType	MP_Common_t	unsigned char
MP_CommonConstantType	MP_Common_t	unsigned char
MP_CommonLatinIdentifierType	MP_Common_t	unsigned char
MP_CommonGreekIdentifierType	MP_Common_t	unsigned char
MP_Sint8Type	MP_Sint8_t	signed char
MP_Uint8Type	MP_Uint8_t	unsigned char
MP_Sint32Type	MP_Sint32_t	signed long int
MP_Uint32Type	MP_Uint32_t	unsigned long int
MP_Real32Type	MP_Real32_t	float
MP_Real64Type	MP_Real64_t	double
MP_ApIntType	MP_ApInt_t	void *
MP_ApRealType	MP_ApReat_t	void *

Table 3.1: MP - C data types

data The actual data value (e.g., 7, sin, PI).

dictionary tag Gives the numeric tag of the dictionary in which the node's value is defined. This field is ignored for those types whose values are not described in dictionaries.

number of annotations Each node of the tree may be annotated with supplemental information. This field specifies the number of such annotations attached to the node.

number of arguments This field is only valid for operator types and gives the number of operands the operator takes. This field is ignored for non-operator types.

A node packet may be viewed as having two parts: a header and a body. Typically the value field is found in the body and everything else is contained in the header. For the "common" types (MP_Sint8Type, MP_Uint8Type, MP_CommonMetaType, MP_CommonOperatorType, MP_CommonConstantType, MP_CommonLatinIdentifierType, MP_CommonGreekIdentifierType, MP_BooleanType), the value field is moved into the header, providing a more compact representation. Hence, for efficiency, the common types should be used whenever possible. Of these, the common operator, common meta, and common constant types are integer *encodings* of data. The mapping from integer to operator/meta type/constant is given in a dictionary (see section 3.3).

The MP API provides a set of routines to put and get packets for each of the MP types. For example, the routine MP_PutSint32Packet() puts a single node packet containing a signed 32 bit integer to a link and MP_GetCommonOperatorPacket() gets from a link a node packet containing a common operator.

3.2.2 data limbs

A data limb contains data *only*; there are no additional fields. In particular, there is no type field to identify the type of the data. Data limbs are used as an optimization either when the sender and receiver have agreed in advance on the type and structure of the data to be exchanged or that information is provided elsewhere, such as in a prototype annotation (see section 4.3). Most packet-level routines have data-limb level equivalents. For example, `IMP_PutSint32()` puts a signed 32-bit integer value to the link and `IMP_GetReal64()` gets a 64-bit value from the link which it interprets as a 64-bit real number. The common types don't have corresponding IMP_Put routines.

3.2.3 annotation packet

Any node of the tree may have zero or more annotations attached to it. Generally an annotation is a piece of information relevant to a node's data, but is not itself an operator or argument to an operator. MP provides a set of protocol-defined annotations and users may supply their own. The decision to attach an annotation is made by the sending tool. Annotations may

- be purely incidental and of no consequence to the correct interpretation of the node's data. For example, `Comment ''This solution works best''`.
- provide additional interpretive information that may be useful to have, but is not critical to the data's interpretation. With the *unit of measurement* annotation, for instance, a user can specify that the value sent is in feet, miles, joules, liters, etc.
- be required for correct decoding of data. For instance, MP uses the *prototype* annotation to specify the format of a special block of data. Without the prototype, the block of data is a meaningless collection of bytes. The same applies to MP-defined annotations used for optimizing purposes including *label*, *reference*, *store*, *stored*, and *retrieve*. Such annotations are tagged with the *required* bit set to 1 (see table 3.2 and below).

An annotation packet has three fields:

annotation type An integer identifying the annotation type (e.g., `MP_AnnotProtoPrototype`).

dictionary tag Gives the numeric tag of the dictionary in which the annotation's type and its characteristics are defined. The tag value indicates whether the annotation is MP- or user-defined (see the next section).

flags Three flags further establish an annotation's meaning and importance:

`MP_AnnotRequired` indicates that the annotation carries some information essential for properly decoding the data stream and/or understanding the data's meaning. A tool receiving an expression tagged with a required annotation should be able to process the annotation. If it cannot, generally it should return an error and/or skip the subexpression tagged with this annotation. The receiver may, however, attempt to process the data, but does so at it its own risk. Conversely, when the flag is cleared, the annotation is said to be supplemental. There should be no harm in disregarding a supplemental annotation.

`MP_AnnotValuated` indicates that the annotation has an argument which is given as an MP Tree. The argument must immediately follow the annotation in the data stream. By contrast, a simple annotation takes no argument.

`MP_AnnotTreeScope` determines whether the annotation applies only to the node to which it is attached or to the entire subtree rooted at that node.

Annotation Flag	Cleared (0)	Set (1)
`MP_AnnotRequired`	supplemental	required
`MP_AnnotValuated`	simple	valuated
`MP_AnnotTreeScope`	node	(sub)tree

Table 3.2: MP annotation flag values

3.3 MP Dictionaries

MP supports collections of definitions of mathematical symbols (symbolic constants and operators) and annotations in dictionaries. A dictionary is an offline, human-readable document containing a list of items together with their meaning. The definition may be formal or informal, but must be sufficiently precise for a reader to understand it. Each dictionary has a character string name which uniquely identifies it to users (e.g., PolynomialDictionary). When used in association with an MP Tree for data exchange, a dictionary is specified by a tag number which uniquely identifies it to other processes. There are 256 dictionary tags available that can be encoded within the node packet header (2^{16} for annotations). They are distributed as shown in table 3.3.

Tag	Define	Description
0	`MP_BasicDict`	MP-defined “basic” dictionary
1...127		less-basic MP-defined dictionaries
128	`MP_NoSemanticsDict`	no semantics at all
129	`MP_ReceiverDict`	semantics of the receiver
130	`MP_UnavailSemanticsDict`	semantics exist but are not available via a dictionary
131	`MP_StringNameDict`	dict name supplied as a string in the packet’s value field
132...154		reserved for future extensions
155	`MP_LocalDict`	for dictionary development/local use
156...254		user-defined “basic” dictionaries
255	`MP_ExtraTagDict`	other dictionaries (read extra tag field)

Table 3.3: Dictionary tags

Dictionaries for particularly common constants, annotations, and operators are defined by MP. The entries in the MP-provided dictionaries are implemented as C “defines”. The complete hierarchy of MP-defined dictionaries is not yet specified. At the moment, there are the following preliminary MP dictionaries:

Tag	Define	Header file	Defines values for
0	`MP_BasicDict`	`MP_BasicDict.h`	“Basic” constructs like $+$, $-$, $*$, $/$, sin, etc
2	`MP_MpDict`	`MP_MpDict.h`	Internal MP use like negotiations
6	`MP_ProtoDict`	`MP_PrototDict.h`	communication of Prototyped MP trees
10	`MP_PolyDict`	`MP_PolyDict.h`	communication of polynomial data
11	`MP_MatrixDict`	`MP_MatrixDict.h`	Linear Algebra
12	`MP_NumberDict`	`MP_NumberDict.h`	Number Theory

Table 3.4: Existing Preliminary MP dictionaries

The respective header files can be found in the `MP/h` directory of the MP distribution.

Some of the tags are explained in more detail below. The details of how the tag is handled internally by

MP are not important for the programmer to know. To use a dictionary, the programmer needs to have access to the dictionary source and understand its contents.

128 The node's value has no meaning for the sender and the receiver should attach no local meaning to it.

129 The sender is not attaching any specific semantics to the node and is relying on the receiver to interpret it according to local semantics. This is especially useful where the sender has defined a function in the receiver, hence there can be no definition for the function in a statically defined dictionary.

130 The semantics of the node's value exists but is not available to the receiver (that is, the receiving process has no possibility of being able to use the node's meaning, however it could retrieve that meaning from the sender).

131 This tag indicates that the actual dictionary used is given as a string as *prefix of* the packet's *value*. For example, "PolynomialDictionary::Add".

155 Reserved to people developing/using a local dictionary. This provides a tag that is guaranteed to be available (at least locally) for experimentation.

255 indicates to the MP software that another field containing the tag number of a dictionary immediately follows the packet header and must be read in to determine which dictionary defines the node's value. The division of these extended tags between MP and non-MP dictionaries will be determined later when (if) they are needed.

3.4 MP Trees

An MP Tree is a linearized version of an annotated parse tree is formed from a prefix parse of the tree. Node packets are distinguished from annotation packets by their position in the stream of packets. This ordering is well-defined since every tree must begin with a node packet, every node specifies the number of annotation packets associated with it, annotations immediately follow the node with which they are associated, and the operands of operator nodes follow, in order, the last annotation packet related to their parent.

Figure 3.1 gives a stylized MP Tree corresponding to the parse tree in Figure 1.1.

Figure 3.1: MP Tree

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Basic	CopBasicDiv	1:2	Common Operator node Pkt
AP	Basic	AnnotMpComment		Annotation Pkt
String		gradient		annotation argument
Cop	Basic	CopBasicMult	0:2	
CommonIdentifier	Receiver	a	0:	Common Id node Pkt
Cop	Basic	CopBasicMinus	0:2	
Identifier	Receiver	x2	0:	Identifier node Pkt
Identifier	Receiver	x1	0:	
Cop	Basic	CopBasicMinus	0:2	
Identifier	Receiver	x2	0:	
Identifier	Receiver	x1	0:	

3.5 Sending and Receiving Messages

As indicated earlier, data is exchanged in *messages* with one or more MP Trees (expressions) contained within a single message. There is a very important synchronizing action between the sender and receiver(s) that must be carefully observed.

```
MP_Status_t MP_EndMsg (MP_Link_pt link)
MP_Status_t MP_ResetLink (MP_Link_pt link)
MP_Status_t MP_EndMsgReset(MP_Link_pt link)
```

Table 3.5: Completing the message

On the *sending side*, the end of a message is marked with a call to `MP_EndMsg()`. This call updates information in the link indicating that this is the last fragment in a complete message and calls a transport-specific routine to hand the data over to the underlying data delivery device.

`MP_ResetLink()` prepares a link for the next message *after* a call to `MP_EndMsg()` has flushed the previous message to the transport device. Typically this reset would be done immediately after the call to `MP_EndMsg()`, but it can be delayed. However it must be called *prior* to assembling another message in the link's send buffer. For convenience, the function `MP_EndMsgReset()` performs both actions.

```
MP_Status_t  MP_SkipMsg(MP_Link_pt link)
MP_Status_t  MP_InitMsg(MP_Link_pt link)
MP_Boolean_t MP_TestEofMsg(MP_Link_pt link)
MP_Boolean_t MP_GetLinkStatus(MP_Link_pt link, int status_to_check)
```

Table 3.6: Routines for preparing to receive a message

`MP_SkipMsg()` is the *receiving side*'s complementary operation to the sending side's `MP_EndMsg()`. It ensures proper message alignment before reading a message by consuming (skipping) all remaining data in the current message and is to be called *before each* new message is read. It does *not* load any data from the *next* message into the link and therefore is a *non-blocking* procedure. Actual loading will be done by an explicit request to read in some MP data (through one of the get routines).

Alternatively, `MP_InitMsg()` may be called in place of `MP_SkipMsg()`. The difference between the two is that `MP_InitMsg()` it will begin loading the next message into the link's buffers. This is a *blocking* read, i.e., a call to `MP_InitMsg()` will block as long as there no data to be read from the link. `MP_TestEofMsg()` can be called to determine if the current message has been completely consumed. It returns `MP_TRUE` if the end of the current message on the link has been reached, else it returns `MP_FALSE`. A useful convention is to transmit a single expression per message. This provides a uniform level of error recovery and allows the receiving side to efficiently skip to the end of an expression (message) it cannot process.

`MP_GetLinkStatus()` can be used to determine if the device is ready for reading or writing and whether the link is a parent link, i.e., whether it is connected to a forked child link. The argument `status_to_check` must be either `MP_LinkReadyReading`, `MP_LinkReadyWriting`, `MP_LinkIsParent`, or `MP_LinkIsChild` The routine returns `MP_TRUE` if the status condition to check is true according to the link's underlying data delivery device, else it returns `MP_FALSE`. This call will not block.

The synchronization of sends and receives is illustrated in figure 3.2.

3.6 Sending Data with MP

Packet-level routines are provided for sending each of the MP types. These routines put a complete node packet (node type, dictionary tag, number of annotations, number of arguments, and data value) or annotation packet (annotation type, dictionary tag and flags) to the send buffer associated with a link and are prefaced

Sender	**Receiver**
while (...) {	while (...) {
	`MP_SkipMsg(link);`
calls to `put` an MP Tree	// or `MP_InitMsg(link);`
`MP_EndMsgReset(link);`	calls to `get` an MP Tree
}	}

Figure 3.2: Synchronizing sending and receiving messages

with `MP_`. No error checking is done on the arguments. In particular, there is no attempt to verify that pointers are not null. Routines which handle the string-based types (see table 3.1) expect a valid C string for the data value. On success, these routines return `MP_Success`; on failure, they return either `MP_Failure` or a specific error value accessible through the link's error variable. The function `MP_PrintError()` can be used to determine the cause of the failure. If present, the log file may also contain useful information.

Packet level routines take at least three arguments: The first one is the (presumed to be open) link, and the last one is the number-of-annotation field (`na`) which specifies the number of annotations that are attached to the packet. The annotations must immediately follow the node packet (see section 3.8). There is a symmetry between the put and get routines. For each put-packet routine, there is a corresponding get-packet routine.

Most types also have equivalents for sending a data limb. These lower level routines only send a data value and are prefaced with `IMP_`. This is useful on the receiving side when the type of the incoming packet may not be known in advance and it is necessary to retrieve the node packet header separately from the packet body. It also provides a mechanism for more compact and efficient communications of homogenous data based on the specifications of the prototype dictionary (`MP_ProtoDict`). The basic idea is to avoid the overhead of communicating the packet header for each datum in a homogenous data object. This is discussed more fully in the section on prototypes (§4.3).

To send a packet in piecemeal fashion, one must first call `IMP_PutNodeHeader()`, then the appropriate `IMP_Put`<*type*>`()` routine. For example, the packet-level call `MP_PutSint32Packet(...)` is equivalent to: `IMP_PutNodeHeader(...); IMP_PutSint32(...)`. *Note* that there are no data level routines for the "common" types as they fit within the node packet header. Also, there is no data-level routine for annotations since they are not composed of multiple parts as node packets are (e.g., header and body) .

3.6.1 sending numeric data

The `MP_Put` routines in table 3.7 put a boolean, integer, or real *node* packet to the link. The `IMP_Put` routines put an integer or real *data limb* to the link. Recall that no type information is passed with a data limb, so the receiver must have some other way to know what type of data it has been sent.

The default format for arbitrary precision numbers (integers and reals) is the GNU Multiple Precision (GMP) package. It is possible, however, to use other big number formats and, when the sender and receiver agree on an alternative format, avoid the cost of converting between a local, non-GMP format and the GMP format. See section 4.1.3 for details on how to support alternative formats. In the absence of support for alternative formats, the put routines for arbitrary precision numbers expect the number to be in the GMP format.

3.6.2 sending strings and raw data

The string handling routines take a pointer to a char. It is expected that the string is terminated with the NULL character. Raw, or "opaque" data, can be sent using one of the PutRaw routines. The data is sent as a

```
MP_Status_t MP_PutBooleanPacket (MP_Link_pt link, MP_Boolean_t bool,
                                    MP_NumAnnot_t na)
MP_Status_t MP_PutSint8Packet (MP_Link_pt link, MP_Uint8_t n, MP_NumAnnot_t na)
MP_Status_t MP_PutUint8Packet (MP_Link_pt link, MP_Uint8_t n, MP_NumAnnot_t na)
MP_Status_t MP_PutSint32Packet (MP_Link_pt link, MP_Uint32_t n,
                                   MP_NumAnnot_t na)
MP_Status_t MP_PutUint32Packet (MP_Link_pt link, MP_Uint32_t n,
                                   MP_NumAnnot_t na)
MP_Status_t MP_PutApIntPacket (MP_Link_pt link, MP_ApInt_t apint,
                                   MP_NumAnnot_t na)
MP_Status_t MP_PutReal32Packet (MP_Link_pt link, MP_Real32_t fp,
                                   MP_NumAnnot_t na)
MP_Status_t MP_PutReal64Packet (MP_Link_pt link, MP_Real64_t dfp,
                                   MP_NumAnnot_t na)
MP_Status_t MP_PutApRealPacket (MP_Link_pt link, MP_ApReal_t apreal,
                                   MP_NumAnnot_t na)
MP_Status_t IMP_PutSint32 (MP_Link_pt link, MP_Sint32_t n)
MP_Status_t IMP_PutUint32 (MP_Link_pt link, MP_Uint32_t n)
MP_Status_t IMP_PutApInt  (MP_Link_pt link, MP_ApInt_t apint)
MP_Status_t IMP_PutReal32 (MP_Link_pt link, MP_Real32_t fp)
MP_Status_t IMP_PutReal64 (MP_Link_pt link, MP_Real64_t dfp)
MP_Status_t IMP_PutApReal (MP_Link_pt link, MP_ApReal_t apreal)
```

Table 3.7: Numeric put routines

```
MP_Status_t MP_PutStringPacket(MP_Link_pt link, char *str, MP_NumAnnot_t na)
MP_Status_t MP_PutRawPacket(MP_Link_pt link, char *rawdata, MP_Uint32_t len,
                               MP_NumAnnot_t na)
MP_Status_t IMP_PutString(MP_Link_pt link, char *str)
MP_Status_t IMP_PutRaw(MP_Link_pt link, char *rawdata, MP_Uint32_t len)
```

Table 3.8: Byte-based put routines

stream of bytes, with no conversions done on it. The `len` argument specifies the number of *bytes* of raw data to be sent.

3.6.3 sending meta types and meta operators

The PutMeta routines put a meta type or a meta operator to the link. A meta type is a value (represented as a string for the "regular" meta type and as an integer for the "common" meta types) *identifying* an MP-defined basic type and is used in assembling a prototype (see table 4.8 in section 4.3 for a list of the MP-defined meta types). A meta operator specifies a particular "operator", usually a structuring operator such as Matrix, Vector, Array, etc. Most of the MP types have common meta type representations. The `dtag` argument specifies the dictionary in which the (common) meta type/operator is defined.

```
MP_Status_t MP_PutMetaTypePacket(MP_Link_pt link, MP_DictTag_t dtag,
                                  char *meta, MP_NumAnnot_t na)
MP_Status_t MP_PutCommonMetaTypePacket(MP_Link_pt link, MP_DictTag_t dtag,
                                  MP_Common_t meta, MP_NumAnnot_t na)
MP_Status_t MP_PutMetaOperatorPacket(MP_Link_pt link, MP_DictTag_t dtag,
                                  char *metaop, MP_NumAnnot_t na,
                                  MP_NumChild_t nc)
MP_Status_t MP_PutCommonMetaOperatorPacket(MP_Link_pt link, MP_DictTag_t dtag,
                                  MP_Common_t metaop, MP_NumAnnot_t na,
                                  MP_NumChild_t nc)
MP_Status_t IMP_PutMetaOperator(MP_Link_pt link, char *metaop)
MP_Status_t IMP_PutMetaType(MP_Link_pt link, char *meta)
```

Table 3.9: Meta type/operator put routines

3.6.4 sending operators, constants, and identifiers

```
MP_Status_t MP_PutOperatorPacket(MP_Link_pt link, MP_DictTag_t dtag, char *op,
                             MP_NumAnnot_t na, MP_NumChild_t nc)
MP_Status_t MP_PutCommonOperatorPacket(MP_Link_pt link,MP_DictTag_t dtag,
                                 MP_Common_t op, MP_NumAnnot_t na,
                                MP_NumChild_t nc)
MP_Status_t MP_PutIdentifierPacket(MP_Link_pt link, MP_DictTag_t dtag,
                                 char *id, MP_NumAnnot_t na)
MP_Status_t MP_PutCommonLatinIdentifierPacket(MP_Link_pt link,
                                 MP_DictTag_t dtag, MP_Common_t cm_id,
                                 MP_NumAnnot_t na)
MP_Status_t MP_PutCommonGreekIdentifierPacket(MP_Link_pt link,
                                 MP_DictTag_t dtag, MP_Common_t cm_id,
                                 MP_NumAnnot_t na)
MP_Status_t MP_PutConstantPacket(MP_Link_pt link, MP_DictTag_t dtag,
                                char *cnst, MP_NumAnnot_t na)
MP_Status_t MP_PutCommonConstantPacket(MP_Link_pt link, MP_DictTag_t dtag,
                                   MP_Common_t cm_cnst, MP_NumAnnot_t na)
MP_Status_t IMP_PutIdentifier(MP_Link_pt link, char *id)
```

Table 3.10: Operator, constant, and identifier put routines

The PutOperator routines put an operator to the link. The operator may represent an arithmetic operator, a function, such as “sine” or “integrate”, a data object, such as “vector” or “dense distributed polynomial”, etc. The `dtag` argument identifies the dictionary in which the operator is defined. The `nc` argument identifies the number of arguments (children) the operator has. In a syntactically correct MP Tree, these arguments must follow the annotations, if any, attached to the operator packet. For `MP_PutOperator()`, the operator argument is a string. For `MP_PutCommonOperator()`, the argument is an integer encoding of the operator found in the dictionary identified by `dtag`.

The two routines to put an Identifier take a *string* for the identifier. The common variants of the routine

(MP_PutCommonLatinIdentifierPacket(), MP_PutCommonGreekIdentifierPacket()) take a *single* character for the identifier which will be transmitted more efficiently than their string cousins. Again, dtag identifies the dictionary in which the identifier is defined. Typically this will be the tag MP_ReceiverDict, indicating that the sender relies on the receiver's "native" understanding of the identifier's meaning, probably established during the current session. Or, it might be MP_NoSemanticsDict, indicating that the sender attaches no meaning to the variable and is explicitly telling the receiver that it must not attach any special meaning to it either.

The PutConstant routine sends a symbolic constant, given as a string, to the link. You must identify the dictionary in which the constant is defined. The "common" variant does the same for an integer encoding of a symbolic constant, again as defined in a dictionary.

3.6.5 sending expressions

As previously indicated, an MP Tree is a prefix parse of an expression, so $3 + 511 * x$ would be transmitted as + 3 * 511 x. The sequence of calls to transmit this expression could be:

```
MP_PutCommonOperatorPacket(link, MP_BasicDict, MP_CopBasicAdd, 0, 2);
MP_PutSint8Packet(link, 3, 0);
MP_PutCommonOperatorPacket(link, MP_BasicDict, MP_CopBasicMult, 0, 2);
MP_PutSint32Packet(link, 511, 0);
MP_PutCommonLatinIdentifierPacket(link, MP_ReceiverDict, 'x', 0);
MP_EndMsgReset(link);
```

The same expression can be sent by assembling the node packets in piecemeal fashion. That is, sending the packet header followed by a data packet containing the data. But this requires more intimate knowledge of how MP sends data.

```
MP_Status_t IMP_PutNodeHeader (MP_Link_pt link,  MP_NodeType_t ntype,
                               MP_DictTag_t dtag, MP_Common_t cval,
                               MP_NumAnnot_t na, MP_NumChild_t nc)
```

Table 3.11: Node header put routine

A node packet header can be sent using IMP_PutNodeHeader(). Note that there is one argument for each field of a node packet header: the node type, the tag of the dictionary in which the node's value is defined (when relevant), a "common" data value, the number of annotations to be attached to the node, and the number of arguments (for operators). When sending a "common" type (e.g., MP_CommonLatinIdentifier, MP_BooleanType), you can supply the data value in the cval argument and forego sending an additional data limb. The example below makes this more concrete.

```
IMP_PutNodeHeader(link, MP_CommonOperatorType, MP_BasicDict,
                   MP_CopBasicAdd, 0, 2);
IMP_PutNodeHeader(link, MP_Sint8Type, 3, 0, 0, 0);
IMP_PutNodeHeader(link, MP_CommonOperatorType, MP_BasicDict,
                   MP_CopBasicMult, 0, 2);
IMP_PutNodeHeader(link, MP_Sint32Type, 0, 0, 0, 0);
IMP_PutSint32(link, 511);
IMP_PutNodeHeader(link,MP_CommonLatinIdentifierType, MP_ReceiverDict,
                   'x', 0, 0);
MP_EndMsgReset(link);
```

Since the operators are all defined in the MP basic dictionary and can be sent as "common" operators, the node packet header provides the complete packet (the body of the packet has been moved into the "common value" field of the node packet header). The integer value 511, however, must be sent as a "regular" value since it cannot be stored as a `MP_Common_t` (an unsigned char with a range of 0 to 255 for unsigned integers). So we send the node packet header first, followed by a data limb containing the signed integer 511. Finally, it is important to note when using `IMP_PutNodeHeader()` that there will be times when some of the arguments are unused. This depends on the type of data to be sent in the node packet. For example, when sending the signed integer 511, we don't use the dictionary tag, common value, or number of arguments fields. It is best to set them to 0 in cases like this. The receiver should always examine the number of annotations fields, so it is essential that if no annotations are attached to a node packet the number of annotations field is set to 0.

A set of macros is available for setting the fields of a node packet header. These are shown in table 3.12.

```
MP_NodeHeader_t  hdr;
MP_NodeType_t    ntype;
MP_NumChild_t    nargs;
MP_NumAnnots_t   nannots;
MP_DictTag_t     dtag;
MP_Common_t      cval;

MP_SetNodeTypeField(hdr, ntype)
MP_SetNodeNumChildField(hdr, nargs)
MP_SetNodeNumAnnotsField(hdr, nannots)
MP_SetNodeDictTagField(hdr, dtag)
MP_SetNodeExtraChildField(hdr)
MP_SetNodeExtraAnnotsField(hdr)
MP_SetNodeCommonValueField(hdr, cval)
```

Table 3.12: Macros for setting node packet fields

3.7 Receiving Data with MP

As mentioned above, for most Put routines there is a corresponding Get routine. At the packet-level, these routines get a complete node packet (node type, dictionary tag, the data value, number of annotations, and, for operators, the number of arguments) or annotation packet (annotation type, dictionary tag, and flags) from the `link`.

Important: Routines that take a single pointer as an argument (e.g., `MP_Sint32_t *n`) assume that the pointer is not NULL and that it points to sufficient storage to hold the data. Routines which take double pointers (e.g., `char **s`) check to see if `*s` is NULL. If so, they will allocate a new block of memory sufficiently large to handle the incoming data. *If `*s` is not NULL, the routines assume that the caller has provided sufficient memory to store the object to be read.* If the caller does not know the size of the incoming object, the pointer should be NULL. It is important that the pointer either be valid (that is, point to previously allocated memory) or be NULL. Uninitialized pointers will cause problems. See section 4.5 for more details on libMP's memory management.

It is the programmer's responsibility to free memory allocated by MP. This can be done with a call to `IMP_RawMemFreeFnc()`, passing as the argument a pointer to the block to be freed (see section 4.5 for more information on memory management).

A get-packet call will fail if the type of the next *node* packet is not of the type requested. In this case, the receive buffer is restored to the state it was in previous to the failed call.

The argument, `na`, specifies the number of annotations attached to the packet. The annotations must immediately follow the node packet (see sections 3.4 and 3.8).

3.7.1 receiving numeric data

These `MP_Get` routines get a boolean, integer, or real *node packet* from the link. The `IMP_Get` routines in table 3.13 get a boolean, integer, or real *data limb* from the link. Recall that no type information is passed with a data limb, so the receiver must have some other way to know what type of data it has been sent.

The default format for arbitrary precision numbers is the GNU Multiple Precision (GMP) package. It is possible, however, to use other arbitrary precision number formats and, when the sender and receiver agree on an alternative format, avoid the cost of converting between a local, non-GMP format and the GMP format. See section 4.1.3 for details on how to support alternative formats.

3.7.2 receiving strings and raw data

The string handling routines in table 3.14 take a double pointer to a char. The string that is returned is terminated with the NULL character. Raw, or "opaque" data, can be received using one of the GetRaw routines. The data is sent as a stream of bytes, with no conversions done on it and none is done by the receiving routine. The `len` field gives the number of *bytes* in the block of raw data.

```
MP_Status_t MP_GetBooleanPacket(MP_Link_pt link, MP_Boolean_t *bool,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetSint8Packet(MP_Link_pt link, MP_Sint8_t *n,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetUint8Packet(MP_Link_pt link, MP_Uint8_t *n,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetSint32Packet(MP_Link_pt link, MP_Sint32_t *n,
MP_Status_t MP_GetUint32Packet(MP_Link_pt link, MP_Uint32_t *n,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetApIntPacket(MP_Link_pt link, MP_ApInt_pt apint,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetReal32Packet(MP_Link_pt link, MP_Real32_t *fp,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetReal64Packet(MP_Link_pt link, MP_Real64_t *dfp,
                                MP_NumAnnot_t *na)
MP_Status_t MP_GetApRealPacket(MP_Link_pt link, MP_ApReal_pt apreal,
                                MP_NumAnnot_t *na)
MP_Status_t IMP_GetBoolean(MP_Link_pt link, MP_Boolean_t *bool)
MP_Status_t IMP_GetSint8(MP_Link_pt link,  MP_Sint8_t *n)
MP_Status_t IMP_GetUint8(MP_Link_pt link,  MP_Uint8_t *n)
MP_Status_t IMP_GetSint32(MP_Link_pt link, MP_Sint32_t *n)
MP_Status_t IMP_GetUint32(MP_Link_pt link, MP_Uint32_t *n)
MP_Status_t IMP_GetApInt(MP_Link_pt link,  MP_ApInt_pt apint)
MP_Status_t IMP_GetReal32(MP_Link_pt link, MP_Real32_t *fp)
MP_Status_t IMP_GetReal64(MP_Link_pt link, MP_Real64_t *dfp)
MP_Status_t IMP_GetApReal(MP_Link_pt link, MP_ApReal_pt apreal)
```

Table 3.13: Numeric get routines

```
MP_Status_t MP_GetStringPacket(MP_Link_pt link, char **str, MP_NumAnnot_t *na)
MP_Status_t MP_GetRawPacket(MP_Link_pt link, char **rawdata, MP_Uint32_t *len,
                             MP_NumAnnot_t *na)
MP_Status_t IMP_GetString(MP_Link_pt link, char **str)
MP_Status_t IMP_GetRaw(MP_Link_pt link, char **rawdata, MP_Uint32_t *len)
```

Table 3.14: Byte-based get routines

3.7.3 receiving meta types and meta operators

The GetMeta routines get a meta type from the link. A meta type is a value (represented as a string for the “regular” meta type and as an integer for the “common” meta types) *identifying* a basic type (see table 4.8 in section 4.3 for a list of the MP defined meta type values defined in the `MP_ProtoDict`). A meta operator specifies a particular “operator”, usually a structuring operator such as Matrix, Vector, Array, etc. Users may define their own meta types in a dictionary. The GetMeta routines are typically used in disassembling a Prototype Tree (see section 4.3 for more details).

```
MP_Status_t MP_GetMetaTypePacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                 char **meta, MP_NumAnnot_t *na)
MP_Status_t MP_GetCommonMetaTypePacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                 MP_Common_t *meta, MP_NumAnnot_t *na)

MP_Status_t MP_GetMetaOperatorPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                 char **mop, MP_NumAnnot_t *na,
                                 MP_NumChild_t *nc)
MP_Status_t MP_GetCommonMetaOperatorPacket(MP_Link_pt link,
                                 MP_DictTag_t *dtag, MP_Common_t *cm_mop,
                                 MP_NumAnnot_t *na, MP_NumChild_t *nc)
MP_Status_t IMP_GetMetaType(MP_Link_pt link, char **meta)
MP_Status_t IMP_GetMetaOperator(MP_Link_pt link, char **mop)
```

Table 3.15: Meta type get routines

3.7.4 receiving operators, constants, and identifiers

```
MP_Status_t MP_GetOperatorPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                 char **op, MP_NumAnnot_t *na,
                                 MP_NumChild_t *nc)
MP_Status_t MP_GetCommonOperatorPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                       MP_Common_t *cm_op, MP_NumAnnot_t *na,
                                       MP_NumChild_t *nc)
MP_Status_t MP_GetIdentifierPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                   char **id, MP_NumAnnot_t *na)
MP_Status_t MP_GetCommonLatinIdentifierPacket(MP_Link_pt link,
                                   MP_DictTag_t *dtag, MP_Common_t *cm_id,
                                   MP_NumAnnot_t *na)
MP_Status_t MP_GetCommonGreekIdentifierPacket(MP_Link_pt link,
                                       MP_DictTag_t *dtag, MP_Common_t *cm_id,
                                       MP_NumAnnot_t *na)
MP_Status_t MP_GetConstantPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                 char **cnst, MP_NumAnnot_t *na)
MP_Status_t MP_GetCommonConstantPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                       MP_Common_t *cm_cnst, MP_NumAnnot_t *na)
MP_Status_t IMP_GetOperator(MP_Link_pt link, char **op)
MP_Status_t IMP_GetIdentifier(MP_Link_pt link, char **id)
```

Table 3.16: Operator, constant, and identifier get routines

The GetOperator routines get an operator from the link. The operator may represent an arithmetic operator, a function, such as “sine” or “integrate”, a data object, such as “vector” or “dense distributed polynomial”, etc. The `dtag` argument identifies the dictionary in which the operator is defined. The `nc` argument identifies the number of arguments (children) the operator has. These arguments follow the annotations, if any, attached to the operator packet. For GetOperator, the operator argument is a double pointer to char. For

GetCommonOperator, the argument is a pointer to an `MP_Common_t`. This small integer will be an encoding of the operator found in the dictionary identified by `dtag`.

`IMP_Get` routines are provided for operators and identifiers to allow incoming node packets to be read in a piecemeal fashion. That is, getting the node packet header and data value separately. This is explained in more detail in the next section.

The two routines to Get an Identifier take a double pointer to char. If the pointer is NULL, sufficient memory will be allocated to store the Identifier. The common variants of the GetIdentifier routine take a pointer to a *single* character in which to store the identifier.

Again, the `dtag` field identifies the dictionary in which the identifier is "defined". Typically this will be `MP_ReceiverDict`, indicating that the sender relies on the receiver's "native" understanding of the identifier, probably established during the current session. Alternatively, `dtag` might be `MP_NoSemanticsDict`, indicating that the identifier has no special meaning to the sender and that the receiver must not attach any special meaning to it either.

There are two routines to get a symbolic constant. One receives the constant as a string and the other as an integer encoding of the symbolic constant (as defined in a dictionary).

3.7.5 receiving expressions

A receiving application parses the incoming message with a series of *gets*. As an example, let's receive the expression that was sent in section 3.6.5: `+ 3 * 511 x`. Of course, in this example we know exactly what was sent and so know the corresponding calls to make. A real application requires a general parser capable of reading MP trees.

The calls to receive the expression could be:

```
if (!MP_TestEofMsg(link))  /* see if we are at the end of a message */
  fprintf(stderr, "Current message not completely consumed!\n");
MP_SkipMsg(link);          /* do message alignment */
MP_GetCommonOperatorPacket(link, &dtag, &cop, &nannots, &nargs);
MP_GetSint8Packet(link, &n, &nannots);
MP_GetCommonOperatorPacket(link, &dtag, &cop, &nannots, &nargs);
MP_GetSint32Packet(link, &n2, &nannots);
MP_GetCommonLatinIdentifierPacket(link, &dtag, &cid, &nannots);
```

```
MP_Status_t  IMP_GetNodeHeader(MP_Link_pt link, MP_NodeType_t *ntype,
                               MP_DictTag_t *dtag, MP_Common_t *cv,
                               MP_NumAnnot_t *na, MP_NumChild_t *nc)
MP_Status_t  MP_PeekHeader(MP_Link_pt link, MP_NodeType_t *ntype,
                           MP_DictTag_t *dtag, MP_Common_t *cv)
```

Table 3.17: Node header get routines

The receiver will find it convenient to receive packets in a piecemeal fashion. That is, first get the node packet header, determine the type of node packet, then get the body of the packet containing the data. On the receiving side, to receive a node packet piecemeal, one must first call `IMP_GetNodeHeader()`, then the appropriate `IMP_Get`<*type*>`()` routine. Of course, if the type of the node packet is a "common" type, the data was retrieved when the packet header was read in, so *there is no separate data field to retrieve*. For example,

```
IMP_GetNodeHeader(link, &ntype, &dtag, &cv, &na, &nc) /* line 1 */
switch (ntype) {
 case MP_Sint32Type:                   /* line 2 */
   IMP_GetSint32(link, &n);
   break;
 case MP_Sint8Type:                    /* line 3 */
 case MP_Uint8Type:                    /* line 4 */
 case MP_CommonOperatorType:           /* line 5 */
    /* nothing to do - we read it all with IMP_GetNodeHeader()
   break;
 . . .
 case MP_OperatorType:                 /* line 6 *
   IMP_GetOperator(link, &op_str);
 . . .
 }
```

Note that for the common types shown here (lines 3 - 5), no further Get calls are needed. For example, the body of the case label for `MP_Sint8Type` on line 2 is empty since we read in the Sint8 with the call to `IMP_GetNodeHeader()` and the integer is stored in `cv`. But for the "regular" MP types (lines 2 and 6), an IMP_Get<type>() call is needed to read the data field associated with the packet header read in line 1.

`MP_PeekHeader()` returns the type of the next node packet in the argument `ntype`, as well as the common value and dictionary tag fields. This routine is distinguished from the `IMP_GetNodeHeader()` routine in that it leaves the state of the link's read buffer *unchanged*. So, for example, one could use `MP_PeekHeader()` to determine the type of the next node packet and use that type as the discriminator in a switch statement. Similarly, there may be times when you expect a common operator, but need to know *which* operator you received (e.g., List, SparseDistributedPolynomial, Matrix) so that you can call the appropriate routine to handle this data. For this situation, the common value and dictionary tag fields are also provided by `MP_PeekHeader()`. Lastly, the `cval` field will only be meaningful if `ntype` indicates an MP common type. The code below is similar to the sample code given above.

```
MP_PeekHeader(link, &ntype, &cv, &dtag);
switch (ntype) {
 case MP_Sint32Type:
   MP_GetSint32Packet(link, &n, &na);
   break;
 case MP_Sint8Type:
   MP_GetSint8Packet(link, &sin8, &na);
   break;
 case MP_Uint8Type:
   MP_GetUint8Packet(link, &uin8, &na);
   break;
 case MP_CommonOperatorType:
   MP_GetCommonOperatorPacket(link, &cop, &dtag, &na, &nc);
   break;
 . . .
 case MP_OperatorType:
   MP_GetOperatorPacket(link, &op_str, &dtag, &na, &nc);
 . . .}
```

In this code, we simply peek at the next node packet header in the link without actually reading it. This allows us to identify the type of the node packet and call the correct `MP_Get<type>Packet()` routine.

Finally, there is also a set of macros for getting the fields of a node packet header. These are shown in table 3.18. The macros return the value of the type of the specified field. For example, a call to `MP_GetNodeNumAnnotsField` would return the valued stored in the number of annotations field of the node packet header.

```
MP_NodeHeader_t hdr;

MP_GetNodeFieldType(hdr)
MP_GetNodeNumAnnotsField(hdr)
MP_GetNodeNumChildField(hdr)
MP_GetNodeDictTagField(hdr)
MP_GetNodeCommonValueField(hdr)
```

Table 3.18: Macros for getting node packet fields

3.8 Sending and Receiving Annotations

```
MP_Status_t MP_PutAnnotationPacket(MP_Link_pt link, MP_DictTag_t dtag,
                                   MP_AnnotType_t atype, MP_AnnotFlags_t flags)
MP_Status_t MP_GetAnnotationPacket(MP_Link_pt link, MP_DictTag_t *dtag,
                                   MP_AnnotType_t *atype, MP_AnnotFlags_t *flags)
```

Table 3.19: Annotation put and get routines

Annotations are transmitted in annotation packets. The decision to send an annotation is made by the sender. To “attach” a single annotation to a node:

1. set the number of annotations field in the node's packet to 1. This is done by passing 1 for `num_annots` to either a Put<type>Packet routine or `IMP_PutNodeHeader()`.
2. call `MP_PutAnnotationPacket()` with the desired annotation type, dictionary tag, and flags.
3. if the annotation is valuated, send the MP Tree giving the argument.

This generalizes easily to the case of sending multiple annotations attached to a single node packet. To send n annotations, simply replace 1 with n in step 1 and repeat steps 3 and 4 n times.

The set of macros shown in table 3.20 are defined for setting and clearing the *flags* field of an annotation.

For example, a sequence of calls to send the comment annotation from figure 1.1 could be:

```
MP_AnnotFlags_t aflag;

MP_SetAnnotRequired(aflag)
MP_SetAnnotValuated(aflag)
MP_SetAnnotTreeScope(aflag)
MP_ClearAnnotRequired(aflag)
MP_ClearAnnotValuated(aflag)
MP_ClearAnnotTreeScope(aflag)
MP_ClearAnnotFlags(aflag)
```

Table 3.20: Macros for setting/clearing annotation flags

```
MP_AnnotFlags_t aflags = 0;

/* 1 in annot field indicates a single annotation is attached */
MP_PutCommonOperatorPacket(link, MP_BasicDict, MP_CopBasicDiv, 1, 2);

/* send the annotation immediately after the node pkt */
MP_SetAnnotTreeScope(aflags);
MP_SetAnnotValuated(aflags);
MP_PutAnnotationPacket(link, MP_MpDict, MP_AnnotMpComment, aflags);

/* send the annotation's argument */
MP_PutStringPacket(link, "gradient", 0);
/* Now we would send the CopBasicDiv operator's arguments */
```

On the receiving side, the parser must check to see if there are any annotations attached to a node. In the case above, once the receiver had read the common operator packet and checked the number of annotations field, it would know to get one annotation with
`MP_GetAnnotationPacket(link, &dtag, &annot, &aflags);`
In checking the annotation type and the annotation flags, the receiver would learn that the annotation is valuated. This means that an MP Tree supplying the argument immediately follows. It is important to remember that the call to `MP_GetAnnotationPacket()` *does not* read in the annotation's argument. You would get an annotation's argument as you would any MP Tree. The predicate macros in table 3.21 are useful for determining if a particular annotation flag has been set. Each macro returns true if the specified flag is set. For example, if an annotation is required, `MP_IsAnnotRequired(aflags)` would return `MP_TRUE`.

```
MP_AnnotFlags_t  aflags;

MP_IsAnnotRequired(aflags)
MP_IsAnnotValuated(aflags)
MP_IsAnnotTreeScope(aflags)
```

Table 3.21: Macros for determining annotation flag settings

Finally, table 3.22 gives a collection of macros for setting and getting the fields of an annotation packet header.

```
MP_AnnotHeader_t hdr;
MP_AnnotType_t   atype;
MP_DictTag_t     dtag;
MP_AnnotFlags_t  aflags;

MP_SetAnnotTypeField(hdr, atype)
MP_SetAnnotDictTagField(hdr, dtag)
MP_SetAnnotFlagsField(hdr, aflags)
MP_GetAnnotFieldType(hdr)
MP_GetAnnotDictTagField(hdr)
MP_GetAnnotFlagsField(hdr)
```

Table 3.22: Macros for setting/getting annotation packet fields

3.9 Sending and Receiving Vectors of Numeric Data

Recall that each node in a normal MP Tree is communicated as a *node packet*, i.e. each datum is communicated with a header. For homogeneous data such as arrays (or vectors) of numeric data, this involves an unnecessary overhead for the node packet headers. As an optimization, MP defines mechanisms to avoid sending full node packets for each datum which, among other objects, allows the transmission of arrays of numeric data in such a way that the elements of the array can be communicated as data limbs (instead of full node packets). These optimization mechanisms are formally defined in the prototype dictionary (see `MP_ProtoDict.h`) and are explained in more detail in section 4.3. In this section, we present the primitive routines for communicating vectors of numeric data.

Briefly, vectors of numeric data are communicated using the common operator `MP_CopProtoArray` which is defined in the `MP_ProtoDict` dictionary. The number-of-arguments field of this operator gives the number of elements of the array. A prototype annotation is attached to this operator whose value (called the Prototype Tree) specifies that each of the arguments of the operator is a data limb of a specified numeric type. Subsequently, the elements of the array can be communicated as a block of data limbs, i.e. as data without node headers.

Table 3.23 gives a set of routines for sending and receiving vectors of machine-precision numbers (see table 3.24 for the supported element types). `MP_PutBasicVector()` puts a vector of elements of one of the types corresponding to the values listed in table 3.24. The routine first puts the array node "header" which provides type information using the prototype annotation (see section 4.3) and then puts `num_elems` elements starting at the address given in `elems`. Note that you cannot attach any additional annotations to the vector using this routine.

Alternatively, you can send the header and vector separately, using `IMP_PutBasicVectorHeader()` followed by `IMP_PutBasicVector()` (or the appropriate `IMP_Put<type>Vector()` routine). The routine `IMP_PutBasicVectorHeader()` builds and puts the necessary prototype annotation. The argument `na` is the number of *additional* annotations you wish to attach to the vector. These annotations must be put *after* the vector header and *before* the vector elements.

For the receiving side, only primitive routines which get the vector elements are provided. These routines take as an argument the type of the vector elements, a double pointer (e.g. `MP_Sint32_t** vec`) for storing the element values and the number of elements of the vector. Note that the element type and number of vector elements must be given explicitly as arguments to these routines, which means that their values must

```
MP_Status_t MP_PutBasicVector(MP_Link_pt link, void *elems,
             MP_Common_t elem_type, MP_NumChild_t num_elems)
MP_Status_t IMP_PutBasicVectorHeader(MP_Link_pt link, MP_Common_t elem_type,
              MP_NumAnnot_t na, MP_NumChild_t num_elems)
MP_Status_t IMP_PutBasicVector(MP_Link_pt link, void *elems,
             MP_Common_t elem_type, MP_NumChild_t num_elems)
MP_Status_t IMP_PutSint32Vector(MP_Link_pt link, MP_Sint32_t *si32vec,
                MP_NumChild_t num_elems)
MP_Status_t  IMP_PutUint32Vector(MP_Link_pt link, MP_Uint32_t *si32vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_PutSint8Vector(MP_Link_pt link, MP_Sint8_t *si8vec,
                MP_NumChild_t num_elems)
MP_Status_t  IMP_PutUint8Vector (MP_Link_pt link, MP_Uint8_t *si8vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_PutReal64Vector(MP_Link_pt link, MP_Real64_t *dvec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_PutReal32Vector(MP_Link_pt link, MP_Real32_t *r32vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_GetBasicVector(MP_Link_pt link, void **elems,
             MP_Common_t elem_type, MP_NumChild_t num_elems)
MP_Status_t IMP_GetSint32Vector(MP_Link_pt link, MP_Sint32_t **si32vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_GetUint32Vector(MP_Link_pt link, MP_Uint32_t **si32vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_GetSint8Vector(MP_Link_pt link, MP_Sint8_t **si8vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_GetUint8Vector(MP_Link_pt link, MP_Uint8_t **si8vec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_GetReal64Vector(MP_Link_pt link, MP_Real64_t **dvec,
                MP_NumChild_t num_elems)
MP_Status_t IMP_GetReal32Vector(MP_Link_pt link, MP_Real32_t **r32vec,
                MP_NumChild_t num_elems)
```

Table 3.23: Vector routines

MP_Sint8Type	MP_Uint8Type
MP_Sint32Type	MP_Uint32Type
MP_Real32Type	MP_Real64Type

Table 3.24: Vector element types

be determined by correctly parsing the vector header. Furthermore, recall that if `vec` is the double pointer passed in, then sufficient memory to hold the vector elements is allocated if `*vec == NULL`. Otherwise, it is assumed that `*vec` already points to sufficient memory to store the vector elements.

To receive a complete vector (header and elements), the routine `MPT_GetTree()` contained in `libMPT`, a library for parsing MP trees which is distributed along with MP, should be used.

One more important thing to note about the vector routines is that they not only provide a convenient and compact way for communicating array elements, but that they also very efficiently move the data between the user's memory space and the transport device since they avoid iterating over each array element and instead use memory copy (`memcpy`) functions. In other words, using the supplied vector routines is the most efficient way to Put/Get arrays of machine-precision numeric data.

Chapter 4

Advanced Topics

4.1 Customizing an MP Environment

The resources managed by an environment include memory buffers, parameters affecting the allocation of buffers, the list of available transport devices, and the file descriptor for logging events. Links are created within the context of an environment. Occasionally, the programmer may want to customize an environment. This may be done by setting environment options and by adding new transport interfaces to the environment.

4.1.1 Getting and setting environment options

```
int MP_GetEnvOption(MP_Env_pt env, int option)
int MP_SetEnvOption(MP_Env_pt env, int option, int value)
```

Table 4.1: MP environment option routines

The environment options settable by the programmer are shown in table 4.2 and explained below.

- `MP_BUFFER_SIZE_OPT` - A link's send and receive buffers are implemented as a linked list of buffer segments. This option allows the programmer to reset the size of the buffer fragments maintained by the environment.
- `MP_MAX_FREE_BUFFERS_OPT` - The environment maintains a pool of free buffers from which links get the buffer segments for their send and receive buffers. This option allows the programmer to determine the maximum number of buffer segments maintained in pool. More buffer segments may be allocated as needs require, but once the expression has been "consumed" (sent or read), only `MAX_FREE_BUFFERS` will be kept in the environment's buffer segment pool. The rest will be freed.
- `MP_INIT_FREE_BUFFERS_OPT` determines the initial number of buffer segments maintained in the environment's pool of buffer segments. The pool may grow as large as `MAX_FREE_BUFFERS` or as determined by setting `MP_MAX_FREE_BUFFERS_OPT`.

The current value for an environment option may be discovered with a call to `MP_GetEnvOption()` which returns the current value of the specified option.

Setting an option is done with `MP_SetEnvOption()` which sets the specified environment `option` to `value` and returns the old value of the option as the result of the function.

Important: An environment's options cannot be changed once the environment has been initialized with a call to `MP_InitializeEnv()`.

Option	Meaning	Default	Legal Range for `value`
MP_BUFFER_SIZE_OPT	buffer size	1024	positive long integer
MP_MAX_FREE_BUFFERS_OPT	max buffer pool size	32	positive long integer
MP_INIT_FREE_BUFFERS_OPT	initial buffer pool size	16	integer $\leq$ MP_DEFAULT_MAX_FREE_BUFF

Table 4.2: MP environment options

Instead, you must first allocate the environment (with `MP_AllocateEnv()`), then reset the desired environment options, and, finally, complete the initialization of the environment. For example, to reset the size of the buffers managed by the environment `env`, you would do:

```
MP_Env_pt env = NULL;

if ((env = MP_AllocateEnv()) == NULL){
  fprintf(stderr, "failed to allocate an environment, exiting\n");
  exit(1);
  }
MP_SetEnvOption(env, MP_BUFFER_SIZE_OPT, 2028);
env = MP_InitializeEnv(env);
```

4.1.2 Adding user-supplied transport devices to an environment

```
MP_TranspList_pt IMP_GetTranspByName (MP_Env_pt env, int transp_dev)
MP_Status_t MP_AddEnvTranspDevice (MP_Env_pt env, int transp_dev,
                                   MP_TranspOps_t *transp_ops)
```

Table 4.3: MP environment transport routines

The MP library includes support for files and TCP sockets. The source for transport interfaces for PVM and ToolBus are provided with the MP source distribution, but are not compiled into the MP library since that would require supplying the PVM and ToolBus libraries with the MP distribution. You are responsible for providing any additional libraries needed.

The programmer may add to the set of transport devices supported by an environment with a call to `MP_AddEnvTranspDevice()`, where `env` is a pointer to the environment to which you want to add the transport interface, `transp_dev` is an integer identifying device type to add (these are given as defines in `MP_Env.h`, e.g., `MP_TcpTransportDev`) and corresponds to the string name for the device by which it will be identified (i.e., through the `-MPtransp` option to `MP_OpenLink()`), and `transp_ops` is the address of a structure of type `MP_TranspOps_t` (defined in `MP_Env.h`).

The call to add a device to the environment must be done *after* the environment has been created and *before* the call to open the link. The sample code below shows how this would be done for PVM. Note that you must explicitly enable the compilation of the device-specific routines for Toolbus and Pvm by giving the appropriate `--enable-PACKAGE` option to the `configure` script prior to building MP. See the INSTALL file that accompanies the distribution for how to enable it as the package to use.

```
#include "MP.h"

 MP_Env_pt  env  = NULL;
 MP_Link_pt link = NULL;

 /* do all the PVM initialization stuff here */
 env = MP_InitializeEnv(env);
 MP_AddEnvTranspDevice(env, MP_PvmTransportDev, &pvm_ops);
 if ((link = MP_OpenLink(env, argc, argv)) == NULL) {
   MP_ReleaseEnv(env);
   pvm_exit();
   exit(1);
   }
```

4.1.3 Adding user-supplied arbitrary precision integer and real packages to an environment

MP's default format for large numbers (integers and reals) is the Gnu Multiple Precision Package. The ability to send and receive large numbers in an application's native format is especially important for parallel distributed applications which need to avoid the cost of having to convert between "standard" and native encodings. In general, the approach MP takes is to allow an application the illusion of sending and receiving large numbers in its "native" format, with any necessary conversions are quietly performed by the put and get routines. Currently, MP provides support of GMP, PARI, and SAC big integers and GMP big reals (see the INSTALL file with the distribution for how to enable and use alternative packages). For point-to-point communication (TCP device), the philosophy is:

- if two sides can agree on a non-GMP format, they may send and receive in that format and thereby gain maximum efficiency in transmission because there are no conversions to be done(assuming you are using one of the supported packages) . The negotiation of which format to use is done automatically when the link is opened.
- if the two sides cannot agree, then sending and receiving is always done in the GMP format, but the non-GMP side has the illusion of sending and receiving in its native format since the conversions are done internally by the PutApNum and GetApNum routines (again, assuming you are using one of the supported packages).

```
MP_Status_t MP_SetEnvBigIntFormat  (MP_Env_t *env, MP_BigIntOps_t *ops,
                                    MP_BigIntFormat_t format)
MP_Status_t MP_SetEnvBigRealFormat  (MP_Env_t *env, MP_BigRealOps_t *ops,
                                    MP_BigRealFormat_t format)
```

Table 4.4: MP environment routine to add an arbitrary precision integer and real package

Support for an alternative big integer/real package includes routines to put and get an integer/real in the package's format, to convert between the package's format and the GMP format, and to convert the package's format to a string for logging. These operations are contained in a structure (`MP_BigIntOps_t` and

MP_BigRealOps_t defined in MP_BigNum.h) which is passed to the environment through the argument ops. See MP_BigNum.c and MP_GmpBigReal.c for examples. format be must one of the formats defined in MP_BigNum.h.

Note that the use of alternative (arbitrary precision) number packages must explicitly enabled by giving the appropriate --enable-PACKAGE option to the configure script prior to building MP. See the INSTALL file that accompanies the distribution for how to enable it as the package to use.

4.2 Setting Link Options

```
int MP_GetLinkOption(MP_Link_pt link, int option)
int MP_SetLinkOption(MP_Link_pt link, int option, int value)
```

Table 4.5: MP link option routines

The link options settable by the programmer are shown in table 4.6. The current value for an environment option may be discovered with a call to MP_GetLinkOption() which returns the current value of the specified option. MP_SetLinkOption() sets the specified link option to value. The send/receive mode and logging options are explained below. The old value of the option is returned on success and MP_Failure is returned if the option is unknown or the value is illegal. Clearly a link's options cannot be changed before the link has been created with a call to MP_OpenLink().

4.2.1 Setting a link's send/receive modes

A few options determining the buffering layer's behavior are available to the application programmer. The way in which a link sends data is referred to as its *send mode*, and the way it reads in MP data is its *receive mode*. The user can specify the send mode as:

- MP_SEND_FRAG_MSG In this mode, the link maintains only one output buffer fragment at any time which is written to the transport device when it becomes full (the default behavior). For some transports, this mode provides some simple concurrency in message processing, allowing the receiver to process the fragment of the message received while the sender assembles and sends the rest of the message.

- MP_SEND_WHOLE_MSG Wait until the whole message is assembled on the sending side before sending it (i.e., wait for MP_EndMsg() to be called). The number of output buffer fragments a link may use in this mode is limited only by the amount of memory available to the process.

The receiver has three options in specifying how data is to be read from a link (receive mode):

- MP_RECV_FRAG_MSG_RECVER The link will read until one *receiver*'s buffer fragment is full, postponing subsequent reads until all data in the buffer has been consumed (the default behavior). The link keeps only one input buffer fragment at any time. This option may be useful when memory is low.

- MP_RECV_FRAG_MSG_SENDER The link will read one *sender*'s fragment at a time, postponing subsequent reads until all data in the fragment has been consumed. The distinction is that the sender's buffer fragment size may be greater than that of the receiver, possibly requiring the receiver to allocate extra memory buffer fragments to accommodate the incoming data.

- MP_RECV_WHOLE_MSG The link will read a complete message before returning.

The number of input buffers a link may use is limited only by the amount of memory available to the process. Also, while all buffers within a single application will be the same size, there is no requirement that the communicating applications have same-size buffers.

Behavior	Option	Legal Values
send mode	MP_LINK_SEND_MODE_OPT	MP_SEND_FRAG_MSG (default) MP_SEND_WHOLE_MSG
receive mode	MP_LINK_RECV_MODE_OPT	MP_RECV_FRAG_MSG_RECVER (default) MP_RECV_FRAG_MSG_SENDER MP_RECV_WHOLE_MSG
log mask	MP_LINK_LOG_MASK_OPT	MP_LOG_INIT_EVENTS (default) MP_LOG_READ_EVENTS MP_LOG_WRITE_EVENTS MP_LOG_ALL_EVENTS MP_LOG_NO_READ_EVENTS MP_LOG_NO_WRITE_EVENTS MP_LOG_NO_EVENTS

Table 4.6: MP link options

4.2.2 Logging a session

MP recognizes a set of events that can occur during the course of a session and information about these events can be logged in the log file.

Logging of events must explicitely be enabled by giving the `--with-debug` option to the `configure` script prior to building MP (alternatively, logging can be enabled by *undefining* the `NO_LOGGING` preprocessor macro).

If logging is enabled, the following events are currently recognized:

- *error events* `MP_ERROR_EVENT` Any error that occurs during initialization, sending, or receiving data, or during conversions. Error events are always logged and cannot be disabled (unless logging as a whole is disabled).

- *read events* `MP_READ_EVENT` Reads from the link's receive buffer and reads from the transport device to the receive buffer.

- *write events* `MP_WRITE_EVENT` Writes to the link's write buffer and writes from the write buffer to the transport device.

- *init events* `MP_INIT_EVENT` Initialization events such as the creation and release of an environment or link, and resetting options on an environment or link.

Logging is done on a per link basis. The log file is opened as part of the initialization of the MP environment. The name is `/tmp/mplog.xxx`, where `xxx` is an integer starting at 100. Each link maintains a log mask identifying which events to log. The mask may be changed through a call to `MP_SetLinkOption()`. Note that you may enable and disable event logging at will. This is especially useful when you know your program is about to output a really large amount of data (say, a huge polynomial) and you don't want to have it written to the log file. You cannot disable logging of error events (except by disabling logging entirely). The default logmask includes initialization and error events.

Figure 4.1 gives part of a sample log file. The preamble gives essential information about the creation of an environment. Each line of output associated with a link is prefaced by the link id; `L1` in the example below. This is followed by the event type and then relevant information about the event. When reading or writing data, the data and packet types are easily discerned from the field following the event field. First there is the data type (e.g., `Operator` or `Sint32`) followed by either `_NP` indicating a Node Packet generated by an MP_Put/Get<type>Packet() routine, or _DP indicating a Data Limb, generated by an IMP_Put/Get<type>() routine. Annotations are identified by the _AP suffix, indicating an Annotation Packet, with the prefix giving the annotation type. The notation used for the entries in the annotation flags field are shown in table 4.7.

```
MP environment: Multi Protocol Version 1.1.2 Fri May  2 14:45:47 1997
        host:    Eeyore
        log file: /tmp/mplog.101
        buffer size: 1024 bytes
        max free buffers: 32
        init free buffers: 16
        number output buffers: 1
        native word ordering: big endian
        native floating point format: IEEE
        native multiple precision integer format: GMP

        Added transport device TCP to environment
        Added transport device FILE to environment
L1: InitE   file_open_connection: opened file ex1.mp in write mode
L1: Application: client
L1: InitE   MP_OpenLink: opened link for transport FILE
L1: InitE   MP_SetLinkOption: set logmask to 0x4
L1: InitE   MP_SetLinkOption: set logmask to 0x6
L1: InitE   MP_SetLinkOption: set logmask to 0x7
InitE  : environment BigInt format set to GMP
InitE  : environment BigReal format set to GMP
L1: WriteE  Operator_NP   annots: 0   dict: 129   op: testSint32s   args: 3
L1: WriteE  Sint32_NP     value: 0
L1: WriteE  Sint32_NP     value: -2147483648
L1: WriteE  AnnotPkt      dict: 2   atype: 7   flags: Sup Arg Tree
L1: WriteE  String_NP     annots: 0   value: lowerbound-for-Sint32s
L1: WriteE  Sint32_NP     value: 2147483647
L1: WriteE  AnnotPkt      dict: 2   atype: 7   flags: Sup Arg Tree
L1: WriteE  String_NP     annots: 0   value: upperbound-for-Sint32s
L1: WriteE  Operator_NP   annots: 0   dict: 129   op: testUint32s   args: 2
L1: WriteE  Uint32_NP     value: 0
. . .
```

Figure 4.1: Sample logging output

MP_LogEvent() can be used to place a log message in the log file associated with link. The log_event can be one of the log events described above or a string supplied by the user. log_msg is a string containing the message you wish to log.

Logging is very useful when debugging, but can hamper performance. The default is to compile the MP library with logging enabled.

4.3 Prototypes

By default, the arguments of any operator are MP Trees (see figure 3.1 given earlier and figure 4.2 described below). However, when all of the arguments of an operator have a homogeneous syntactic format it is possible

Notation	Meaning
Req	Required annotation
Sup	Supplemental annotation
Arg	annotation has an argument
NoArg	annotation has no argument
Node	annotation has node scope
Tree	annotation has tree scope

Table 4.7: Annotation flags in log entries

to minimize the overhead incurred by the node packet headers of the arguments. Ideally, we would like to send *just* the data and avoid the overhead of all the typing information. An array of floating point numbers is a good example, but the data could be more structured - an array of complex or rational numbers, or a polynomial or generators of an ideal.

A *prototype*, then, is a syntax tree providing type and structuring information about a composite data object. The sender uses the prototype to define the object. Similarly, the receiver uses this information to properly parse the "raw" data. This section gives a brief description of how to construct a prototype; more detailed information can be found in [3].

4.3.1 prototypes and meta types

A very simple example will both motivate the need for prototypes and their use. Consider an array of 1000 `floats` (`MP_Real32Type`) as shown in the stylized MP Tree in figure 4.2.

Figure 4.2: MP Tree for an array of 1000 MP_Real32 numbers

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	CopProtoArray	0:1000	1) Common Operator Packet
MP_Real32		1.0	0:	2) first element
MP_Real32		2.0	0:	3) second element
		...		
MP_Real32		1000.0	0:	1001) thousandth element

Line (1) indicates that we have an array of 1000 arguments (elements). These elements follow from line (2) on. The use of MP_Real32 in the Type column indicates that a node packet was used to send the datum. Recall that the cost of each node packet *header* is four bytes. So the size of this MP Tree is 8,008 bytes. Of the 8,000 bytes devoted to sending the floating point numbers (line (2) on), 4,000 bytes are required for the packet headers and another 4,000 for the data. Clearly it is a nuisance (and inefficient) to have the duplication of the type information for each data element of the array. This is not merely a matter of the size of the data. Since the type information is attached to each data element, the receiver is forced to make a linear pass through the array, extracting each datum one at a time. A much more efficient approach would be to transmit the data as a block of bytes (which is likely how it is stored within the application) which not only reduces the amount of data that must be sent, but also allows the sender and receiver to use very efficient memory move operations to move the data between application- and MP-memory spaces. But this requires a mechanism which tells the receiver that each element of the array is of a single type. This is precisely what the prototype annotation does. So prototypes may be thought of as *type specifiers*.

Consider figure 4.3, which is a prototyped version of figure 4.2 (the code to send this tree is shown in figure 4.4). As before, line (1) contains the common operator `CopProtoArray` defined in the Proto (prototype) dictionary. But now the packet header indicates that an annotation is attached to the operator. The annotation,

Figure 4.3: Prototyped MP Tree for an array of 1000 IMP_Real32 numbers

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	CopProtoArray	1:1000	1) Common Operator Packet
AP	Proto	Prototype		2) Prototype Annotation Packet
Cmt	Proto	IMP_Real32	0:	3) MP type specification
IMP_Real32		1.0		4) beginning of Prototyped Data Tree:
IMP_Real32		2.0		data only, no extra
IMP_Real32		3.0		packet information
		...		

given on line (2), is a prototype annotation. A prototype annotation always takes a single argument, which is again an MP Tree providing the type information for each of the arguments of the operator to which the annotation is attached. For clarity, we refer to such an MP Tree as a *Prototype Tree*. We refer to the block of data described by a Prototype Tree as a Prototyped Data Tree. In this case, the Prototype Tree is very simple, consisting of a single type specifier indicating that *each* element of the array is of the type IMP_Real32 (see line (3)). MP uses *meta* types within Prototype Trees to specify that a data limb of the specified diata type will appear at this place in the Prototyped Data Tree. The "common" meta type (Cmt) is reserved for those types that have a unique integer assigned to them within a dictionary (see table 4.8 for a list of the MP-defined meta types). (The "regular" meta types are simply a string equivalent of this – that is, the type information is given as a sTring instead of an integer.) So the common meta type packet on line (3) specifies the type of data that will appear in this position of the MP Tree that follows. The type specified is IMP_Real32. Recall that the "IMP" indicates a *data limb* is being sent and *not* a complete node packet. Figure 4.5 relates these terms and shows how a sparse matrix can be represented using a prototype.

To return to our example here, the Prototyped Data Tree begins on line (4). Note that there are no node packet headers to *individually* specify the type of each element of the Prototyped Data Tree (this is the job of the prototype). Using the prototype mechanism to specify the array's element type reduces the total size of the array's encoding from 8,008 bytes (4,008 bytes total overhead) to 4,016 bytes (16 bytes of overhead).

There are three further important points to make about prototypes before moving on to more complex examples. The first is that the size of the prototype is largely independent of the size of the data. In the example above, the overhead for the prototype stays at 16 bytes even if, for example, the size of the data type doubles (4 to 8 bytes for an `IMP_Real64`) or the length of the array increases. Secondly, one can attach additional annotations to the nodes of the Prototype Tree, which would then apply to *all* the data limbs described by that node. Finally, the use of prototypes does not preclude the possibility of writing efficient routines for sending and receiving mathematical objects. Instead of decoding each single datum by interpreting the prototype, a receiver can simply check the entire prototype to ensure that the incoming data meets the expectations of a pre-compiled read routine. The sending side is simpler: An object-specific routine may efficiently marshall the user's data structure, *including* sending the prototype describing the object.

4.3.2 prototypes and common operators

In the example considered in figure 4.3, the Prototype Tree consisted of a single *basic* type specifier. To send a 1000 signed 32-bit integers you would simply replace the IMP_Real32 on line (3) with IMP_Sint32. Similarly for any of the other MP basic types.

Defining a prototype for more structured data (a matrix or polynomial, for example) requires the use of structuring operators within the Prototype Tree. This structuring is provided by four common operators and the meta operator type. The four common operators are: Struct, RecStruct, Union, and RecUnion. These are the only common operators that may appear within a syntactically correct Prototype Tree. A (common) meta operator is the operator equivalent of the meta type. Think of it as an "operator-specifier" instead of a "type-specifier". As we will see, its semantics are slightly different from those of the "normal" operators.

```
MP_Real32_t r32_array[1000];

/* ... Fill the array ... */

/* (1) Put the operator node packet */
MP_PutCommonOperatorPacket(link,             /* MP_Link_pt */
                           MP_ProtoDict,     /* defining dictionary */
                           MP_CopProtoArray, /* array operator */
                           1,                /* 1 annotation */
                           1000);            /* 1000 args */

/* (2) Put the prototype annotation packet */
MP_PutAnnotationPacket(link,
                       MP_ProtoDict,          /* defining dictionary */
                       MP_AnnotProtoPrototype, /* prototype annot */
                       MP_AnnotValuatedFlag & MP_AnnotRequiredFlag);

/* (3) Put typespecifaction for the data limbs */
MP_PutCommonMetaTypePacket(link,
                           MP_ProtoDict,
                           MP_CmtProtoIMP_Real32,
                           0);         /* no annots to Cmt */

/* (4) Put the data limbs */
for (i=0; i<1000; i++) IMP_PutReal32(link, r32_array[i]);

/* A more efficient way would actually be to use:
 * IMP_PutReal32Vector(link, r32_array, 1000);
 * instead of putting each data limb separately */
```

Figure 4.4: Code fragment for putting a prototyped array of IMP_Real32's

meta type	description
MP_CmtProtoIMP_Sint8	Sint8 data limb
MP_CmtProtoIMP_Uint8	Uint8 data limb
MP_CmtProtoIMP_Sint32	Sint32 data limb
MP_CmtProtoIMP_Uint32	Uint32 data limb
MP_CmtProtoIMP_ApInt	ApInt data limb
MP_CmtProtoIMP_ApReal	ApReal data limb
MP_CmtProtoIMP_Real32	Real32 data limb
MP_CmtProtoIMP_Real64	Real64 data limb
MP_CmtProtoIMP_Boolean	Boolean data limb
MP_CmtProtoIMP_Identifier	Identifier data limb
MP_CmtProtoIMP_LatinIdentifier	LatinIdentifier data limb
MP_CmtProtoIMP_GreekIdentifier	GreekIdentifier data limb
MP_CmtProtoIMP_CommonConstant	CommonConstant data limb
MP_CmtProtoIMP_Constant	Constant data limb
MP_CmtProtoIMP_String	String data limb
MP_CmtProtoRecStruct	recursive structure indicator
MP_CmtProtoRecUnion	recursive union indicator
MP_CmtNumberRational	rational number

Table 4.8: MP-defined meta types

Again, we use examples to describe and illustrate the essential concepts. First, consider an array of structures. Figure 4.5 shows how a sparse 100×100 integer matrix A which has zero entries everywhere except for $A[2,2] = A[2,3] = A[2,4] = 1.0$ can be represented using the structure definition below to represent the matrix elements. In this example we use the CopProtoStruct operator to provide the necessary structuring within the Prototype Tree.

```
struct SparseMatElem {
   MP_Uint32    row_num;
   MP_Uint32    colnum;
   MP_Real64    val;
}
```

The top-level operator on line (1) indicates that this is a three element sparse matrix. The prototype (lines 2 - 6) specifies the type of each element. Line (3) indicates that each element is a 3-field structure and the arguments to this structure operator (lines 4 - 6) specify the individual types of the arguments using common meta type packets. The annotation on lines (7 - 8) provides the dimensions of the matrix. The Prototyped Data Tree begin on line (9) with the first field of the first structure. The data for the second structure begins on line (10) and the third on line (11).

The CopProtoUnion operator looks quite similar, but has an important difference in its interpretation. As with a C `union`, the "arguments" to the union operator are really type specifiers, and, as with C, a discriminator is required to identify the particular specifier to use to extract data from the Prototyped Data Tree. So, at *data communication time* (the time at which the data described by the Prototype Tree is sent), a union discriminator is first transmitted as an IMP_Uint32. The range for the union discrimination index is $0 - n$. An index of 0 specifies that *no* data follows the index (that is, none of the type specifier's arguments is selected – it is convenient to think of this as a NULL pointer). An index i between 1 and n specifies that exactly one Prototyped Data Tree follows the index whose type is that of the i^{th} argument of the CopProtoUnion operator node packet.

MP Tree

Type	Dict	Value	#Annot:Arg	Remarks	
Cop	Matrix	SparseMat	2:3	1)	**Matrix operator**
AP	Proto	Prototype		2)	**Prototype annotation**
Cop	Proto	CopProtoStruct	0:3	3)	
Cmt	Proto	IMP_Uint32	0:	4) rowspec	**Prototype Tree**
Cmt	Proto	IMP_Uint32	0:	5) columnspec	
Cmt	Proto	IMP_Sint32	0:	6) elem type	
AP	Matrix	AnnotMatrixDim		7) dimensions	**dimension annotation and argument**
Cop	Proto	CopProtoArray	0:2		
MP_Uint32		100	0:	dim1	
MP_Uint32		100	0:	8) dim2	
IMP_Uint32		1		9)	
IMP_Uint32		1			
IMP_Sint32		1		[1, 1] = 1	
IMP_Uint32		2		10)	
IMP_Uint32		1			**Prototyped Data Tree** (whose structure and type type content are described by the Prototype Tree)
IMP_Sint32		1		[2, 1] = 1	
IMP_Uint32		2		11)	
IMP_Uint32		4			
IMP_Sint32		1		[2, 4] = 1	

Figure 4.5: Prototyped MP Tree for a sparse matrix

The example in figure 4.6 illustrates the use of the CopProtoUnion operator. This example is intended to be illustrative only - do not expect it to be efficient!

Figure 4.6: Prototyped MP Tree for an array of integers using the union operator

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	CopProtoArray	1:10	
AP	Proto	Prototype		
Cop	Proto	CopProtoUnion	0:3	1) 3 prototypes follow
Cmt	Proto	IMP_Uint32	0:	2) 1st prototype gets index 1
Cmt	Proto	IMP_Sint32	0:	3) 2nd prototype get index 2
Cmt	Proto	IMP_ApInt	0:	4) 3rd prototype gets index 3
IMP_Uint32		2		5) 1st item has index 2, use prototype 2
IMP_Sint32		-245		1st data value IMP_Sint32
IMP_Uint32		1		6) 2nd item has index 1, use prototype 1
IMP_Uint32		1057		2nd data value (IMP_Uint32)
IMP_Uint32		3		7) 3rd item has index 1, use prototype 1
IMP_ApInt		1234567890		3rd data value (IMP_ApInt)
		. . .		and so on

The CopProtoUnion operator on line (1) indicates that there are 3 type specifiers in the union. These are given on lines (2), (3), and (4). The Prototyped Data Tree begin on line (5) with an IMP_Uint32 serving as union discriminator. The value is 2, indicating that prototype 2 (defined on line (3)) specifies the type of the following data limb. Once a receiver has consumed the data according to the prototype, it is positioned to read the next discriminator on line (6). This would proceed until all 10 arguments had been read.

4.3.3 prototypes and meta operators

For objects such as structures and unions, the sender knows at *prototype specification time* (the time at which the prototype is sent) how many arguments the structuring operator has. Also, the type of each field of a struct or union is given quite easily within the Prototype Tree by specifying them through *the operator's arguments*. But this approach is not practical for objects which are a repetition of a single type specification (an array, for example), or for which the actual number of arguments is not known at prototype specification time, or for which the number of arguments may change for separate instances of the object within the tree (a pointer or ragged array, for example).

These cases are handled through meta operators (MP_MetaOperator and MP_CommonMetaOperator node packets) which are operator variants of the meta types and may only appear in a Prototype Tree. There are two important points to make about the meta operator. First, like its basic cousin the meta type, the meta operator is providing information about something, in this case a specific operator (given as the meta operator's *value*). Second, an important distinction between operators and meta operators is that for the former, the number-of-arguments field specifies the number of arguments that will follow the operator packet, while for the meta operators, the number-of-arguments field only gives *information about* the number of arguments that will appear in the Prototyped Data Tree that follows. There are only two cases to consider:

1. A meta operator node packet with a non-0 number-of-arguments field. This is the number of arguments transmitted at data communication time.

2. A meta operator node packet with 0 in the number-of-arguments field. Transmitting the actual number of arguments the operator has is delayed from prototype communication time to data communication time. That is, the actual number of arguments is transmitted at data communication time by an IMP_Uint32 data limb before the arguments are transmitted *at the place where the operator would have appeared at data communication time*.

Note that the operator is not resent at data communication time.

As a first example, consider a matrix of reals sent as an array of arrays[1]. In this case, the prototype consists of a common meta operator whose value is CopProtoArray and whose number-of-arguments field is non-zero. The common meta operator itself has a prototype annotation specifying the type of each argument of the CopProtoArray operator. Notice that the number-of-arguments field of the meta operator specifies the number of data values that will *appear at data communication time* and *not* the number of arguments that follow the meta operator within the Prototype Tree. Figure 4.7 gives prototyped and non-prototyped versions side-by-side for comparison.

The top-level operator on line (1) indicates that we have an array with 2 arguments. The prototype annotation starting on line (2) specifies that the type of each element of the array from line (1) is a 3 element array (line 3). The meta operator on line (3) also has a prototype (lines 4 - 5) specifying that the elements of each subarray is an IMP_Real32. Note that the meta operator on line (3) does not have any actual arguments within the Prototype Tree. The number-of-arguments field specifies the number of arguments to expect in the Prototyped Data Tree that follows.

As a second example, suppose that the subarrays in the example above have different lengths, say one has length 2 and the other length 3. In this case we need to use a meta operator node packet with 0 in the number-of-arguments field as shown in figure 4.8.

As in the first example, the prototype (lines 2 – 5) specifies that each element of the top-level array is again an array of IMP_Real32's. However, the 0 in the number-of-arguments field of the meta operator in line (3) specifies that the actual number of elements of the subarrays is not given at prototype specification time but is instead given at data communication time by an `IMP_Uint32` data limb *in the place where the operator would have appeared* (lines (6) and (7)).

[1] This example is illustrative only. Two-dimensional matrices should be sent as defined in the matrix dictionary. For example, using the `MP_CopMatrixDenseMatrix` with an attached prototype annotation specifying the type of the basic matrix elements and a matrix-dimension annotation giving the dimensions of the matrix.

Figure 4.7: Prototyped and unprototyped MP trees for a matrix as an array of even-length arrays

```
{
  { -1.0, -2.0, -3.0 },
  { -4.0, -5.0, -6.0 }
}
```

Prototyped version					Non-Prototyped version			
Type	Dict	Value	#Annot:Arg	Remarks	Type	Dict	Value	#Annot:Arg
Cop	Proto	CopProtoArray	1:2	1) 2 elem array	Cop	Proto	Array	0:2
AP	Proto	Prototype		2) element type				
Cmop	Proto	CopProtoArray	1:3	3) is Array				
AP	Proto	Prototype		4) subarray type				
Cmt	Proto	IMP_Real32	0:	5) IMP_Real32				
					Cop	Proto	CopProtoArray	0:3
IMP_Real32		-1.0		1st subarray	MP_Real32		-1.0	0:
IMP_Real32		-2.0			MP_Real32		-2.0	0:
IMP_Real32		-3.0			MP_Real32		-3.0	0:
					Cop	Proto	CopProtoArray	0:3
IMP_Real32		-4.0		2nd subarray	MP_Real32		-4.0	0:
IMP_Real32		-5.0			MP_Real32		-5.0	0:
IMP_Real32		-6.0			MP_Real32		-6.0	

Figure 4.8: Prototyped and unprototyped MP Trees for an array of uneven-length arrays

```
{
  { -1.0, -2.0, -3.0 },
  { -4.0, -5.0 }
}
```

Prototyped version					Non-Prototyped version			
Type	Dict	Value	#Annot:Arg	Remarks	Type	Dict	Value	#Annot:Arg
Cop	Proto	CopProtoArray	1:2	1) 2 elem array	Cop	Proto	Array	0:2
AP	Proto	Prototype		2) element type				
Cmop	Proto	CopProtoArray	1:0	3) is Array				
AP	Proto	Prototype		4) subarray type				
Cmt	Proto	IMP_Real32	0:	5) IMP_Real32				
IMP_Uint32		3		6) 1st length	Cop	Proto	Array	0:3
IMP_Real32		-1.0		1st subarray	MP_Real32		-1.0	0:
IMP_Real32		-2.0			MP_Real32		-2.0	0:
IMP_Real32		-3.0			MP_Real32		-3.0	0:
IMP_Uint32		2		7) 2nd length	Cop	Proto	Array	0:2
IMP_Real32		-4.0		2nd subarray	MP_Real32		-4.0	0:
IMP_Real32		-5.0			MP_Real32		-5.0	0:

There are two ways to get the effect of pointers within prototypes. First, we can use a meta operator CopProto**Pointer** which at prototype specification time *always* has 0 in its number-of-argument field and a prototype annotation attached to the meta operator specifying the type of the data "pointed to". Alternatively, we could use the CopProto**Union** operator with *one* argument (specifying the type of the data "pointed to"). In both cases, an IMP_Uint32 is transmitted at data communication time which can have the value 0 (corresponding to a `NULL` pointer) or 1 (the pointer is not `NULL` – or, use the first type specifier in the Union).

Consider, for example, an array of two `struct_A` structures as defined below:

```
struct struct_B {
  MP_String_t;
  MP_Uint32_t;
}

struct struct_A {
  MP_Sint32_t;
  MP_Real32_t;
  struct struct_B *;  // a ptr to struct_B
}
```

and the following data:
[(456, 90.12, ()), - First structure
(71, 2.1, ("Ubnd", 2)] - Second structure

where the structure data is surrounded by parentheses and () indicates a NULL pointer.

Figure 4.9: Prototyped MP tree for an array of structures containing pointers

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	CopProtoArray	1:2	A 2 element array
AP	Proto	Prototype		1) prototype #1
Cop	Proto	CopProtoStruct	0:3	2) each element is a structure (`struct1`)
Cmt	Proto	IMP_Sint32	0:	
Cmt	Proto	IMP_Real32	0:	
Cmop	Proto	CopProtoPointer	1:0	3) 3rd element is a pointer
AP	Proto	Prototype		4) prototype #2 describing structure of thing pointed to
Cop	Proto	CopProtoStruct	2:0	5) it is a structure of 2 things (`struct2`)
Cmt	Proto	IMP_String	0:	6)
Cmt	Proto	IMP_Uint32	0:	7)
IMP_Sint32		456		8) 1st instance of `struct_A` - retrieve using prototype #1
IMP_Real32		90.12		
IMP_Uint32		0		9) Value for Pointer is NULL, so end of 1st `struct_A`
IMP_Sint32		71		2nd instance of `struct_A`
IMP_Real32		2.1		
IMP_Uint32		1		10) Value for Pointer is non-NULL, get the struct pointed to
IMP_String		"Ubnd"		11) retrieve using prototype #2
IMP_Uint32		2		12)

Figure 4.9 shows what this would look like in MP. The first prototype on line (1) indicates that each element of the array is a 3-field structure. Line (3) indicates that the third field is a pointer to an object (in this case, CopProtoPointer was used). Lines (4 - 7) give the prototype describing the object pointed to on line (3). The structuring prototype on line (5) says that the object pointed to on line (3) is a 2-field structure. The prototyped data tree follows the Prototype tree beginning on line (8). Notice that lines (3) and (4) could have been replaced by the line

Cop Proto CopProtoUnion 0:1 (use union operator instead of a meta operator)

without any changes in the other lines. In general, for pointers involving Union and RecUnion, the discriminator field can perform double duty: a discriminator value of 0 indicates the NULL pointer and a non-0 discriminator indicates both non-NULL and which field of the union describes the data to follow.

The third field of the first element of the array appears on line (9). This is the field corresponding to the structure pointer from line (3). The value here is 0, indicating a NULL pointer, so the receiver skips the nested prototype (lines 4 - 7) describing the structure pointed to. However, the value for the pointer field for the second array element on line (10) is 1 (non-NULL), so the receiver uses the prototype from lines (4 - 7) to read the object pointed to: Line (6) tells the receiver to read a String (found on line (11)), and line (7) says to read a Uint32 (found on line (12)). There were only two elements to the array, so the end of the MP Tree has been reached.

Finally, recursive data structures are defined by matching the common meta types **Cmt**ProtoRecUnion and **Cmt**ProtoRecStruct with their common operator counterparts **Cop**ProtoRecUnion and **Cop**ProtoRecStruct.

The simple example below illustrates the concepts (also see figure 4.10).

```
struct recurse_ex {
  MP_Sint32_t;
  MP_Real32_t;
  struct recurse_ex *;  // self-referencing pointer
}
```

Figure 4.10: Prototyped MP Tree for a recursive structure

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	CopProtoArray	1:1	
AP	Proto	Prototype		1)
Cop	Proto	CopProtoRecStruct	0:3	2) defining a recursive structure
Cmt	Proto	IMP_Sint32	0:	3)
Cmt	Proto	IMP_Real32	0:	4)
Cmop	Proto	CopProtoPointer	1:0	5) pointer
AP	Proto	Prototype		6)
Cmt	Proto	CopProtoRecStruct	0:	7) points to struct from line (2)
IMP_Sint32		10		8) use prototype from lines (1 - 7)
IMP_Real32		2.3		
IMP_Uint32		1		9) recursive pointer, 1 indicates another struct follows
IMP_Sint32		20		10) use prototype from lines (1 - 7)
IMP_Real32		6.5		11)
IMP_Uint32		0		12) 2nd struct ends here, 0 indicates no more follow

The prototype on lines (1 - 7) describes a recursive data structure. The CopProtoRecStruct operator on line (2) indicates each argument of the top-level CopProtoArray operator is a structure of three elements and indicates that this is a recursive structure. The third field of this structure on line (5) is a meta operator node packet (used as a pointer), whose prototype on lines (6 - 7) specifies that this is a recursive pointer "pointing to" the *most closely nested* CopProtoRecStruct occurring previously within the Prototype Tree, which is on line (2). Note that the pointer's number-of-arguments field is 0. The Prototyped Data Tree immediately follows the Prototype Tree (line (8) on). The first occurrence of the recursive pointer field appears on line (9). Here the actual number-of-arguments of the CopProtoPointer is found. Since it is 1 (non-NULL), another instance of this structure (lines 10 - 12) follows. The recursive pointer field of this second structure is shown on line (12). Here the value is 0 indicating that the pointer is NULL. Since this is a recursive data structure, we have reached the end of the "linked list" and no more data follows.

The next example combines CopProtoRecUnion, CopProtoStruct, and recursive type specifications. The data structure is a recursive representation of a sparse polynomial. One possible declaration looks like:

```
union SparseRecPoly  {
  MP_Sint32_t;                // coefficient
```

```
  struct inner_poly{
    MP_String_t;              // varname
    MP_Uint32_t;              // exponent
    union SparseRecPoly *;    // multiplic subpoly
    union SparseRecPoly *;    // additive subpoly
  };
};
```

The sample data is:

```
3*x^4(y+2) + x^2  =  x^4    +    x^2
                     *
                     (y     +    2)
                     *
                     3
```

Its encoding in MP is shown in the usual table form in figure 4.11 and as a source-code fragment using the MP API in figure 4.12. Notice how the union discriminators are used as indirect recursive pointers.

Figure 4.11: Protoyped MP Tree for a sparse recursive polynomial

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Poly	SparseRecPoly	1:1	top-level operator
AP	Proto	Prototype		
Cop	Proto	RecUnion	0:2	recursive union
Cmt	Proto	IMP_Sint32	0:	coeficient typespec
Cop	Proto	Struct	0:4	inner_poly struct typespec
Cmt	Proto	IMP_String	0:	var name typespec
Cmt	Proto	IMP_Uint32	0:	exponent typespec
Cmt	Proto	RecUnion	0:	mult. subpoly recursive typespec
Cmt	Proto	RecUnion	0:	add. subpoly recursive typespec
IMP_Uint32		2		use proto 2
IMP_String		x		the outermost var is x
IMP_Uint32		4		its exponent is 4
IMP_Uint32		2		mult subpoly to x^4 – use proto 2
IMP_String		y		the current var is y
IMP_Uint32		1		its exponent is 1
IMP_Uint32		1		mult subpoly to y – use proto 1
IMP_Sint32		3		coeff
IMP_Uint32		1		add subpoly y – use proto 1
IMP_Sint32		2		coeff
IMP_Uint32		2		add subpoly to x^4 – use proto 2
IMP_String		x		var is again x
IMP_Uint32		2		its exponent is now 2
IMP_Uint32		1		mult subpoly to x^2 – use proto 1
IMP_Sint32		1		coeff
IMP_Uint32		0		no add. subpoly to x^2 – use proto 0

Based on the recursive polynomial example and the rules governing meta operator type specifications, it is easy to develop the specifications for more complex polynomial data, such as matrices of polynomials, or the generators of a polynomial ideal. In these cases, the top-level CopPolySparseRecPoly operator of the example above appears as a common meta operator at the root of the type specifier to the top-level CopPolyIdeal operator (or, of the CopMatrixDenseDistMatrix operator) and the rest of the type specification is exactly as above. As a last example, we show how this is accomplished in figure 4.13.

```
MP_PutCommonOperatorPacket(link,          /* top-level operator */
                           MP_PolyDict,
                           MP_CopPolySparseRecPoly,
                           1,1);
MP_PutAnnotationPacket(link,              /* prototype annot */
                       MP_ProtoDict,
                       MP_AnnotProtoPrototype,
                       MP_AnnotValuatedFlag & MP_AnnotRequiredFlag);
MP_PutCommonOperatorPacket(link,          /* recursive union */
                           MP_ProtoDict,
                           MP_CopProtoRecUnion,
                           0, 2);
MP_PutCommonMetaTypePacket(link,          /* coefficient typespec */
                           MP_ProtoDict,
                           MP_CmtProtoIMP_Sint32, 0);
MP_PutCommonOperatorPacket(link,        /* inner_poly struct typespec */
                           MP_ProtoDict,
                           MP_CopProtoStruct,
                           0, 4);
MP_PutCommonMetaTypePacket(link,          /* var name typespec */
                           MP_ProtoDict,
                           MP_CmtProtoIMP_String, 0);
MP_PutCommonMetaTypePacket(link,          /* exponent typespec */
                           MP_ProtoDict,
                           MP_CmtProtoIMP_Uint32, 0);
MP_PutCommonMetaTypePacket(link,  /* mult. subpoly recursive typespec */
                           MP_ProtoDict,
                           MP_CmtProtoRecUnion, 0);
MP_PutCommonMetaTypePacket(link, /* add subpoly recursive typespec */
                           MP_ProtoDict,
                           MP_CmtProtoRecUnion 0);
/* Poly data begins */
IMP_PutUint32(link, 2);   /* use proto 2 */
IMP_PutString(link, "x"); /* the outermost var is x */
IMP_PutUint32(link, 4);   /* its exponent is 4 */
IMP_PutUint32(link, 2);   /* mult subpoly to x^4 -- use proto 2  */
IMP_PutString(link, "y"); /* the current var is y */
IMP_PutUint32(link, 1);   /* its exponent is 1 */
IMP_PutUint32(link, 1);   /* mult subpoly to y -- use proto 1 */
IMP_PutSint32(link, 3);   /* coeff */
IMP_PutUint32(link, 1);   /* add. subpoly to y -- use proto 1 */
IMP_PutSint32(link, 2);   /* coeff */
IMP_PutUint32(link, 2);   /* add subpoly to x^4 -- use proto 2 */
IMP_PutString(link, "x")  /* var is again x */
IMP_PutUint32(link, 2);   /* its exponent is now 2 */
IMP_PutUint32(link, 1);   /* mult subpoly to x^2 -- use proto 1  */
IMP_PutSint32(link, 1);   /* coeff is 1 */
IMP_PutUint32(link, 0);   /* no add. subpoly to x^2-- use proto 0 */
```

Figure 4.12: Sending the polynomial $3x^4y + x^2$ using the MP API

Figure 4.13: Prototyped MP Tree for an ideal

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Poly	CopPolyIdeal	1:10	an ideal consisting of 10 polys
AP	Proto	Prototype		each poly is a
Cmop	Poly	CopPolySparseRecPoly	1:1	sparse-rec poly – notice use of Cmop here
AP	Proto	Prototype		
Cop	Proto	CopProtoRecUnion	0:2	recursive union
Cmt	Proto	IMP_Sint32	0:	coefficient typespec
Cop	Proto	CopProtoStruct	0:4	inner_poly struct typespec
Cmt	Proto	IMP_String	0:	var name typespec
Cmt	Proto	IMP_Uint32	0:	exponent typespec
Cmt	Proto	CmtProtoRecUnion	0:	mult. subpoly recursive typespec
Cmt	Proto	CmtProtoRecUnion	0:	add. subpoly recursive typespec
data of the 10 polynomials goes here, starting with the prototype to be used for the first poly, etc				

4.4 Buffer Level Routines

It is not expected that the programmer will access a link's buffers directly using these routines. They are described for completeness and to allow developers to build their own higher level routines.

```
MP_Status_t IMP_GetLong(MP_Link_pt link, long *lp)
MP_Status_t IMP_PutLong(MP_Link_pt link, long *lp)
MP_Status_t IMP_GetBytes(MP_Link_pt link, caddr_t addr, unsigned long len)
MP_Status_t IMP_PutBytes(MP_Link_pt link, caddr_t addr, unsigned long len)
```

Table 4.9: Buffer level routines

`IMP_PutLong()` puts a four byte quantity, treated as a long integer, to the link pointed to by `link`. `MP_Failure` is returned only if the buffer is full and the write operation fails. Conversely, `IMP_GetLong()` gets a four byte quantity, treated as a long integer, from the link. `MP_Failure` is returned if all data in the current message has been exhausted or if there is a problem with the link. These routines handle byte ordering according to link-specific attributes.

`IMP_GetBytes()` and `IMP_PutBytes()` get/put `len` bytes starting at address `addr` from/to the link pointed to by `link`. `MP_Failure` is returned from the put routine only if the buffer is full and the write operation fails. `MP_Failure` is returned from the get routine if all data in the current logical message has been exhausted or if there is a problem with the link. `len` may be greater than the buffer size. The get routine assumes that there is sufficient room at `addr` to store the given number of bytes.

4.5 Memory Management

The memory functions shown in table 4.10 are reassignable. By default they use the standard memory functions `malloc()`, `realloc()`, and `free()`.

Important: Changing the memory management routines should be done at the beginning of the program, before any memory allocation has been done and certainly before a call to *any* other MP routines.

MP allocates/frees memory using the `IMP_Mem*` functions for all data types except for strings (arrays of characters) and raw data (data received by the `GetRaw` routines). Memory for these data types is allocated/freed using the `IMP_RawMem*` functions.

```
void* (*IMP_MemAllocFnc)(size_t)
void* (*IMP_MemReallocFnc)(void *, size_t, size_t)
void  (*IMP_MemFreeFnc)(void *, size_t)

void* (*IMP_RawMemAllocFnc) (size_t)
void* (*IMP_RawMemReallocFnc) (void *, size_t)
void  (*IMP_RawMemFreeFnc) (void *)
```

Table 4.10: Memory routines

4.6 Error Handling

```
void MP_PrintError(MP_Link_pt link)
char* MP_ErrorStr(MP_Link_pt link)
MP_Status_t MP_ClearError(MP_Link_pt link)
MP_Status_t MP_SetError(MP_Link_pt link, MP_Status_t the_err)
```

Table 4.11: Error handling routines

Where it makes sense to do so, MP routines return a status value: `MP_Success`, `MP_Failure`, or a specific error value. Table 4.12 lists the MP error codes. Each link maintains a local `errno` which records the error status of the last event involving the link. An error message can be printed to `stderr` with `MP_PrintError()`. This does not reset the link's error value. If logging has not been disabled, a message describing the error will automatically be entered in the log file. `MP_ErrorStr()` returns a pointer to a string describing the error associated with the link's `errno` field.

The values for `MP_Success` and `MP_Failure` are consistent with C's encoding of true and false, allowing the programmer to test for success/failure in the usual manner. For example,

```
if (!MP_PutSint32Packet(link, n, 0))
  MP_PrintError(link);
```

`MP_ClearError()` sets the link's error variable `errno` to `MP_Success` and returns `MP_Success`.
`MP_SetError()` sets the link's error variable to `the_err` and returns `MP_Failure`.

Acknowledgements

The author wishes to thank Ch'ng Shi Baw for his work implementing the new buffering layer and in preparing the initial interface to PVM3. Olaf Bachmann contributed the MPT (MP Tree) library which is included with and builds upon the MP library, numerous fixes and enhancements to the code, and an improved configure script. Thanks also go to Paul Wang, Norbert Kajler, and Olaf Bachmann for their comments on earlier drafts of this Guide.

Error Code	Meaning
`MP_Failure`	Failure
`MP_Success`	Success
`MP_UnknownType`	Unknown type
`MP_UnknownAnnotation`	Unknown annotation
`MP_UnknownRequiredAnnotation`	Unknown required annotation
`MP_NullLink`	Null link
`MP_NullTransport`	Null transport device
`MP_CantOpenFile`	Can't open file link
`MP_CantConnect`	Can't make connection
`MP_MemAlloc`	Can't allocate memory
`MP_CantConvert`	Can't convert
`MP_ValueOutOfRange`	Value is out of the legal range for this type
`MP_CantPutNodeHeader`	Can't put node header
`MP_CantPutNodePacket`	Can't put node packet
`MP_CantPutDataLimb`	Can't put data limb
`MP_CantPutAnnotationPacket`	Can't put annotation packet
`MP_CantGetNodeHeader`	Can't get node header
`MP_CantGetNodePacket`	Can't get node packet
`MP_CantGetDataLimb`	Can't get data limb
`MP_CantGetAnnotationPacket`	Can't get annotation packet
`MP_NotRequestedPacketType`	Incoming packet not type requested
`MP_CantConvertBadString`	Can't convert, bad string
`MP_CantWriteLink`	Can't write link
`MP_CantReadLink`	Can't read link
`MP_NotCommonIdentifier`	Identifier longer than one character
`MP_WrongBasicVectorType`	Illegal element type for a Basic Vector
`MP_DummyBigNum`	Can't put/get dummy bignum

Table 4.12: MP error codes

Bibliography

[1] O. Bachmann, S. Gray, and H. Schönemann. A Framework for Distributed Polynomial Systems Based on MP. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'96)*, Zurich, Switzerland, July 1996. ACM Press.

[2] O. Bachmann, S. Gray, and H. Schönemann. A proposal for syntactic data integration for math protocols. In *to appear in the Proceedings of the International Symposium on Parallel Symbolic Computation (PASCO'97)*, Hawaii, USA, July 1997. .

[3] Bachmann, O. and Gray, S. and Schönemann, H. MP Prototype Specification. In *Reports On Computer Algebra*, number 12. Centre for Computer Algebra, University of Kaiserslautern, January 1997.

[4] J.A. Bergstra and P. Klint. The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam, 1995.

[5] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. RFC 1521, September 1993.

[6] S. Gray, N. Kajler, and P. S. Wang. MP: A Protocol for Efficient Exchange of Mathematical Expressions. In M. Giesbrecht, editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'94)*, pages 330–335, Oxford, GB, July 1994. ACM Press.

[7] S. Gray, N. Kajler, and P. S. Wang. Specification of Expression Encoding in the Multi Protocol, version 0.5. Technical Report ICM-9404-64, Institute for Computational Mathematics, Kent State University, 1994.

[8] S. Gray, N. Kajler, and P. S. Wang. Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions. *Journal of Symbolic Computing*, 1997. Forthcoming.

[9] Institute of Electrical and Electronics Engineers. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard 754-1985, August 1985.

[10] N. Kajler. Building a Computer Algebra Environment by Composition of Collaborative Tools. In J. P. Fitch, editor, *Proc. of DISCO'92, Bath, GB*, volume 721 of *LNCS*, pages 85–94. Springer-Verlag, April 1992.

[11] D. Schefström and G. van den Broek, editors. *Tool Integration*. Wiley Press, 1993.

[12] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency – Practice & Experience*, 2:315–339, 1990.

Index