

MRT Programmer's Guide

Version 2.0.0 Alpha

(Draft 11/5/99)

Table of Contents

Introduction	4
1. Overview	6
2. MRT Base Library	11
3. Select Library	13
4. Gateways and Prefixes	14
5. Timer Library	15
6. Trace Library	17
7. User Interactive Interface (UII)	19
8. Interface Library	20
9. I/O Library	21
10. linked_list.a Library	23
11. hash.a Library	36
12. Packet Formats	46

Copyright (c) 1997, 1998

The Regents of the University of Michigan ("The Regents") and Merit Network, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of Michigan, Merit Network, Inc., and their contributors.

4. Neither the name of the University, Merit Network, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Introduction

This chapter introduces the *MRT Programmer's Guide* and explains how to obtain further information about MRT.

Document Conventions

The following document conventions are used in the *Programmer's Guide*:

- Commands and keywords are in **boldface**.
- User-supplied variables are enclosed in <angle brackets>.
- Optional elements are shown in [square brackets].
- Alternative but required keywords are grouped in {braces} and separated by a vertical bar.

Related MRT Manuals

The following documentation is also available for MRT users (see http://www.merit.edu/~mrt/mrt_doc/):

- *Installation Guide*
- *Programmer's Manual*
- Tutorial (in preparation)

The MRT web site will also have the most up-to-date documentation and code.

Getting Help

For more information about MRT, send mail to mrt-support@merit.edu.

The MRT and IPMA development teams are available to answer questions and provide configuration advice. We are also very interested in bug reports, feature requests, and general feedback.

A mailing list, mrt-discuss-request@merit.edu is also available for MRT users to share advice and experiences with the toolkit.

Notes

NOTE: The MRT libraries may undergo significant changes prior to the 2.0 release of MRT. After the 2.0 public release, we expect the library API will remain stable.

Credits

MRT was developed by Merit Network, Inc., under National Science Foundation grant NCR-

9318902, "Experimentation with Routing Technology to be Used for Inter-Domain Routing in the Internet." The current research is supported by the National Science Foundation (NCR-9710176) and a gift from Intel Corporation.

The design and ideas behind many of the MRT libraries draws heavily on the architecture pioneered in the GateD routing daemon.

The University of Michigan/Merit Network MRT development team includes: Craig Labovitz, Masaki Hirabaru, Farnam Jahanian, Susan Hares and Susan Rebecca Harris. Additional code and architecture ideas were supplied by Marc Unangst and John Scudder.

Francis Dupont developed the initial BGP4+ code.

The public domain Struct C-library of linked list, hash table and memory allocation routines was developed by Jonathan Dekock <dekock@cadence.com>.

David Ward <dward@netstar.com> provided bug fixes and helpful suggestions.

Pedro Roque developed the first port to Linux IPv6, and wrote many of the interface routines to the Linux kernel.

We would also like to thank our other colleagues in Japan, Portugal, the Netherlands, the UK, and the US for their many contributions to the MRT development effort.

1. Overview

The Multi-threaded Routing Toolkit provides researchers and application developers with a powerful collection of routing protocol libraries and services.

1.1 Libraries

The MRT libraries fall into two main categories:

- Lower-level services and support routines (timer, interface, socket routines, etc.)
- Protocol modules (BGP, RIPng, routing table support, etc.)

Service Libraries

The lower-level service libraries provide routines common to most routing protocol implementations. For example, most protocols have periodic processes, such as sending out KeepAlive packets or timing out routing table entries. The MRT timer library includes routines for multiplexing the Unix signal timer over multiple protocols. Similarly, most routing implementations have a need for receiving packets and buffering outbound packets. The MRT select library provides routines for handling network I/O. Other libraries, including the trace and UII libraries, provide routines to facilitate logging of trace information and the User Interactive Interface.

Routing Libraries

The MRT libraries also include high-level interfaces into routing protocols. These interfaces include access to BGP, RIPng and other protocol communications, as well as access to the kernel's routing table and interface management. For example, initiating a BGP peering session with a remote router can be as simple as linking in the MRT BGP library and calling `Add_BGP_Peer (hostname, AS)`.

1.2 Modules

MRT provides an object-oriented, multi-threaded programming environment. Under the MRT architecture, functional entities, such as routing protocols and other application-level services, are modeled as **modules**, or objects. Each module is associated with its own thread of control. On threads-capable multiprocessor machines, modules run on top of native kernel threads. On operating systems lacking threads support, MRT modules run on top of emulated threads.

Modules maintain control of their own private resources, which may include buffers, file descriptors, and network sockets. Modules provide public methods, or call routines, for accessing the data and services supplied by the module. For example, a BGP module may provide public routines for initiating a peering session, trapping BGP packets, and providing event notification.

Each MRT object maintains a queue of pending events, which we will refer to as tasks. Other objects and services may schedule events by calling one of the object's public methods. For example, the BGP module might call `RIP.route_change_notify()` to alert the RIP module that

some external BGP routes have changed, and RIP should begin announcing these new routes.

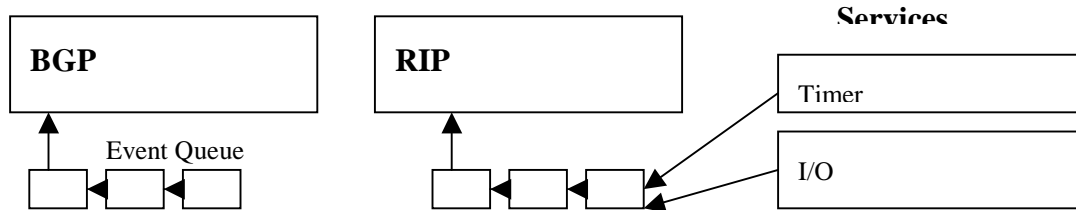


Figure 1 - MRT Pending Events

1.3 Services

The MRT architecture also includes specialized threads of control called **services**. Services generally perform specialized, narrowly focused tasks such as I/O multiplexing or timer notification. These threads only schedule events with other modules and do not maintain event queues, nor conduct any event scheduling or processing of their own. Examples of services include a timer and I/O monitor service.

The current implementation of MRT includes the following modules and services: timer, select, BGP, RIP, RIPng, user-interface, and a main controlling thread. Since most UNIX implementations only associate a single timer per process, the MRT timer thread is used to multiplex the single process timer over multiple threads. Objects like BGP and RIP schedule alarms by calling the timer scheduler method and providing both an interval and an absolute time, and a callback method. The timer thread maintains a sorted queue of time-based events, which associate pending alarms with objects and their callback methods. The timer service also includes mechanisms for one-time alarms and adding jitter to timer intervals.

The select service performs synchronous I/O multiplexing on behalf of MRT modules. Since modules already block while waiting for the scheduling of new events, the objects require another mechanism for learning of pending I/O. The select thread monitors object socket descriptors for pending read, write and exception events. Objects register an interest in socket by calling a select service method. Once the select service detects an I/O event, the service invokes the callback method of an object and stops monitoring the socket. After an object completes processing of a socket, it notifies the select thread to once again begin monitoring.

1.4 Getting Started

A basic MRT program looks like the following:

```
cc [flag ...] file... [library...] -lmrt

#include <mrt.h>
main (int argc, char *argv[]) {
    init_mrt (NULL);
}
```

This program is the bare minimum needed to make use of any MRT services. Of course, this program does not do anything even remotely interesting – yet...

Adding Timers

Let's suppose that we were writing a new routing protocol, LPF (longest path first). The RFC for LPF specifies that it must send out a KeepAlive packet every 30 seconds. Although we are not sure how to build a LPF KeepAlive packet yet, we can begin by at least constructing the framework for a routine that would send the KeepAlive at the appropriate times. We use the MRT timer routines:

```
#include <mrt.h>
void send_lpf_keepalive (void) {
    /* send keep alive when we figure that much out */
}

main (argc, arv) {
    mtimer_t *keep_alive_timer;
    init_mrt (NULL);
    keep_alive_timer = New_Timer (send_lpr_keep_alive, 30,
        "LPF Keep Alive Timer", NULL);

    Timer_Turn_ON (keep_alive_timer);
}
```

The above code calls **New_Timer** to create an MRT timer that calls the keepalive callback routine every 30 seconds. The first argument of **New_Timer** specifies the stub callback function, **send_lpf_keepalive**, which we will eventually get around to writing. The second argument to **New_Timer** is the callback interval, or the time to wait between calls to **send_lpf_keepalive**. Finally, the fourth argument is a text string used for debugging, and the final argument (**NULL**) is a pointer to data that will be passed to our **send_lpf_keepalive** routine. After creating a timer, we turn it on by a call to **Timer_Turn_ON** to begin the initial 30-second interval.

Dealing with I/O

Now that we know how to send keepalives, we turn our attention to how we can receive them. MRT provides a select library to facilitate the multiplexing of I/O. The select library is really a wrapper around the standard UNIX select system call.

```
void recv_lpf_packet (void) {
    /* process lpf packet */
    select_enable_fd (lpf_fd);
}

main (argc, argv) {
    int lpf_fd;

    fd = create_lpf_socket ();

    select_add_fd (lpf_fd, SELECT_READ, recv_lpf_packet, NULL);

    /* start our main program loop */
    main_loop ();
}
```

Assuming the socket was created and initialized in the user routine `create_lpf_socket`, we can tell MRT to call `recv_lpf_packet` each time there is new data to read on the lpf socket. We add the lpf socket by calling `select_add_fd`. We need to call `select_enable_fd` after processing the packet in `recv_lpf_packet` to notify MRT that we have read the data. By default, MRT will ignore a socket after receiving data on the socket (and invoking the callback routine) until the program re-asserts an interest in the socket by calling `select_enable_fd`.

Going Multi-threaded

Until now, our programs have used a single module with an associated thread of control. If we have multiple protocols, say OSPF running in combination with LPF, we will probably want to design LPF as a separate module with its own thread of control. We begin by creating a **schedule** for LPF. Schedules are the basic building block of modules in MRT. All communication between protocols and modules or services occurs by scheduling, or adding events, to a module's schedule queue.

A call to `New_Schedule` creates a new schedule queue structure. This schedule structure is used by other modules to queue events .

```
#include <mrt.h>
schedule_t *lpf_schedule;

void lpf_thread () {
    /* we are now a module !*/

    /* Wait around for things to happen (like packets
     * arriving and timers firing
     */
    schedule_wait_for_event (lpf_schedule);
}
```

```
void send_lpf_keepalive (void) {
    /* we are a method called by a different module. Since we
     * don't have access to any of LPF's resources, we need
     * to schedule the sending of a lpf packet with the LPF
     * module.
     */
    schedule_event (lpf_schedule, _send_lpf_schedule, NULL);
    return;
}

main () {

    lpf_schedule = New_Schedule ("LPF Schedule", NULL);
    mrt_thread_create ("LPF Thread", lpf_schedule, lpf_thread, NULL);

    /* main loop */
}
```

In main, we first create a schedule structure by calling **New_Schedule**. We provide **New_Schedule** with a string used for debugging, "LPF Schedule." We then create a new MRT thread/module by calling **mrt_thread_create**. This routine creates a new thread of control and begins running the thread with the specified function -- **lpf_thread** in the above example.

Once MRT creates a thread of control, and begins execution of **lpf_thread**, the lpf module calls **schedule_wait_for_event** to wait for other modules and services to schedule events.

2. MRT Base Library

The MRT base library provides basic routines for initializing MRT services and managing modules and threads of control.

Synopsis

```
cc [flag ...] file... [library...] -lmrt
#include <mrt.h>

int init_mrt(trace_t* trace);
mrt_thread_t* mrt_thread_create (char *name, schedule_t *schedule, void
(*call_fn) (), void *arg);
schedule_t *New_Schedule (char *description, trace_t *trace);
int schedule_event (schedule_t *schedule, void (*call_fn)(), int nargs, ...);
int schedule_wait_for_event (schedule_t *schedule);
void Delete_Event (event_t *event)
int clear_schedule (schedule_t *schedule);
int mrt_thread_exit (mrt_thread_t *thread);
```

Description

All MRT programs must call `init_mrt()`. This function initializes memory, bookkeeping functions and lower level library routines.

An MRT module is created by calling `mrt_thread_create`. This function creates a new thread of control and begins execution of this thread by calling the function pointed to by the `call_fn` argument. The procedure `mrt_thread_create` also takes a name character string used for debugging, a pointer to a schedule, and a argument which will be passed to `call_fn`. On thread-capable operating systems, `mrt_thread_create` is layered on top of `pthread_create(1)`.

`new_schedule` returns a pointer to a `schedule_t` data structure. This structure maintains the queue of pending events for a module. In most cases, programs will need to maintain the pointer to a schedule as a global variable. Other MRT libraries and routines will need this schedule pointer to access the methods and data of the module owning the schedule.

Once an MRT module, or thread, has been launched from `mrt_thread_create`, the new module may perform some initialization functions and then usually waits for pending events by calling `schedule_wait_for_event`. This procedure blocks the thread until another module or service notifies the blocked module that there are new events pending.

Other modules and services, such as the timer or select service, access another module's data and procedures by calling the modules schedule methods to schedule the processing of new events. `schedule_event` takes a point to a module's schedule structure, the name of the method to be scheduled, `narg` number of arguments to be passed, and a variable list of void pointers to the

arguments.

After a module has returned from processing an event dispatched from `schedule_wait_for_event`, the event is removed from the schedule queue, and the module needs to explicitly delete the event before returning.

3. Select Library

The select library provides a collection of utilities built on top of the Unix `select(1)` function call for multiplexing I/O data.

Synopsis

```
cc [flag ...] file... [library...] -lmrt
#include <mrt.h>

int select_disable_fd (int fd);
int select_enable_fd (int fd);
int select_delete_fd (int fd);
int select_add_fd (int fd, int type_mask, void (*call_fn)(), void *arg);
```

Description

After creating a socket, a module can request to receive notification of events (socket ready to read, socket ready to write, exceptions), by calling `select_add_fd`. The `fd` argument is the socket, the `type_mask` is `SELECT_READ`, `SELECT_WRITE`, `SELECT_EXCEPTION`, and the `call_fn` is the callback routing to call whenever one of the selected events occurs.

When one of the conditions occurs, MRT will invoke the callback function. The module can then process socket by reading, writing data or closing the socket. After the module is done with processing, it must call `select_enable_fd` to reregister and interest in the socket with the MRT select service.

4. Gateways and Prefixes

MRT provides a number of convenience routines for handling network addresses. A MRT prefix is similar to the Unix sockaddr structure with a number of important differences – a prefix maintains information about the mask, or number of important bit positions in an address, and a prefix can contain either an IPv4 or IPv6 address.

A gateway structure maintains an Autonomous System number (AS) with an associated prefix address.

Synopsis

```
cc [flag ...] file... [library...] -lmrt
#include <mrt.h>

gateway_t *New_Gateway (prefix_t *prefix, int AS);
char *gateway_toa (char *tmp, gateway_t *gateway);
gateway_t *find_gateway (prefix_t *prefix, int AS, interface_t *interface);
prefix_t *New_Prefix (int family, u_char *dest, int bitlen);
void Delete_Prefix (prefix_t *prefix);
prefix_t *ascii_toprefix(char *string, trace_t *trace);
char *prefix_toa (prefix_t *prefix);
char *prefix_toa2 (prefix_t *prefix, char *tmp);
u_char *prefix_tochar (prefix_t *prefix);
int prefix_compare (prefix_t *p1, prefix_t *p2);
```

Description

The function `New_Prefix` creates a new `prefix_t` structure. The arguments to `New_Prefix` are an address family (either `AF_INET` or `AF_INET6`), a pointer to a 4 byte (IPv4) or 128 byte (IPv6) address, and the mask or number of significant bits. Like `New_Prefix`, `ascii2prefix` creates a new prefix structure. The `ascii2prefix` structure takes a string of the form “x.x.x.x/x” or the ASCII DNS hostname of a machine.

Several routings convert prefix information to character strings for logging and debugging. `Prefix_toa` returns a prefix address in the form “x.x.x.x”. `Prefix_toax` includes “/x” mask information.

`prefix_compare` compares two prefixes returning 1 if they are the same and -1 if they are different.

5. Timer Library

SunOS associates a single timer per process. Unfortunately, a single timer is usually insufficient for most routing protocols. Protocols like RIP, BGP and IDRP require multiple timers to schedule events like sending KeepAlive packets, timing out dead peers, and retrying the opening of peering sessions. The MRT timer library provides an easy-to-use programming interface to multiplex Unix alarm signals.

The MRT Timer library provides easy programming interfaces to create, destroy, set and manipulate timers.

Synopsis

```
cc [flag...] file... -lmrt [library...]
#include <timer.h>

void init_timer_master();

Timer *New_Timer (void (*call_fn)(), int interval, char *name, void *arg);

void Timer_Turn_ON (Timer *timer);

void Timer_Turn_OFF (Timer *timer);

void Timer_Reset_Time (Timer *timer);
```

Description

All programs making use of the MRT timer library **MUST** call **init_timer_master()** before calling any timer routines. Failure to call **init_timer_master** will result in uninitialized timer structures and lead to segment faults

After initializing the master timer information, individual timers are created by calling **New_Timer ()**. The **New_Timer ()** function initializes a new timer that calls the user-defined *call_fn* function after the timer fires every *interval* number of seconds. After the *call_fn* is executed, the timer automatically resets to expire *interval* seconds in the future. The *name* argument is a descriptive string used in logging timer events. **New_Timer** returns a pointer to a Timer data structure. The *call_fn* is called with two arguments, a pointer to the Timer data structure that fired, and a pointer to the user-defined data pointer *arg*. The user-defined call function has the form:

```
void my_call_fn (Timer *timer, void *my_data);
```

New_Timer returns a NULL pointer upon failure.

All timers are initially created in the OFF state. In this state, the timer is not scheduled, and the *call_fn* will not execute. To turn a timer ON, you must call the function **Timer_Turn_ON ()**.

Similarly, if a timer is no longer needed, **Timer_Turn_OFF ()** will remove *timer* from the timer queue and prevent the timer from firing in the future.

The **Timer_Reset_Time** function resets the *timer* to fire after the timer's original interval number of seconds expires. This function is used by protocols like BGP which need to reset KeepAlive timers after receiving a KeepAlive from a peer. If a *timer* is OFF, the **Timer_Reset_Time** call will turn the *timer* on. **Timer_Set_Time** changes the interval value of an initialized *timer*. After calling **Timer_Set_Time**, *timer* will fire after every *interval* seconds. If a *timer* is OFF, the **Timer_Set_Time** call will turn the *timer* on.

Finally, a protocol that no longer has need of any timer may clear all memory associated with timers by calling **Destroy_Timer_Master**. Most protocols will not make use of this function. Protocols **MUST** not call other timer library routines after invoking **Destroy_Timer_Master**.

Code Examples:

```
#include <stdio.h>
#include <timer.h>

void my_timer_fire1() { printf("10 I have fired\n"); }
void my_timer_fire2() { printf("60 I have fired\n"); }

main() {
    mtimer_t *timer1, *timer2;

    init_timer();

    timer1 = New_Timer (my_timer_fire1, 60, "timer1");
    timer2 = New_Timer (my_timer_fire2, 10, "timer2");

    Timer_Turn_ON (timer1);
    Timer_Turn_ON (timer2);

    while (1) mrt_alarm();
}
```

Makefile Example:

```
INCDIR=-I$(HOME)/mrt/include
LIBDIR=-L$(HOME)/mrt/lib
LIBS=-lstruct -lutil -ltimer

all: my_file

my_file: my_file.o
    $(CC) -o my_file $(LIBDIR) $(LIBS)

my_file.o: my_file.c
    $(CC) $(INCDIR) -c my_file.c
```

6. Trace Library

The MRT Trace library provides simple programming interface for programs and protocols to log trace information either to disk, or to the console.

Synopsis

```
cc [flag...] file... -ltrace [library...]  
#include <trace.h>  
  
void trace (int severity, Trace_Struct *trace_struct, ...);  
  
Trace_Struct *New_Trace_Struct (int first, ...);  
  
int Set_Trace_Struct (Trace_Struct *tmp, int first, ...)
```

Description

Protocols making use of the MRT Trace library must first call **New_Trace_Struct** to allocate memory and initialize the trace information. As arguments, **New_Trace_Struct** takes a null terminated list of option flag, option value pairs. Possible options, their associated type of argument, and their default value include:

TR_LOG_FILE	char*	"/tmp/mrt.log"
TR_FLAGS	int	NORM
TR_APPEND	boolean	TRUE
TR_FLOCK	boolean	TRUE

TR_LOG_FILE specifies the name of the file to which trace calls will write logging information. If this flag is followed by an argument of "stdout", trace calls will write to standard out.

The *TR_FLAGS* argument specifies the level, or verbosity, of tracing information that will be generated. Possible arguments include:

FATAL	/* fatal error -- die after receiving */
TRACE	/* verbose tracing, like keepalives */
NORM	/* normal events to trace */
TR_PARSE	/* trace parsing of config files */
TR_PACKET	/* trace packet coming and goings */
TR_STATE	/* trace state changes and events */
TR_TIMER	/* trace timer changes and events */
TR_POLICY	/* trace policy changes and events */
TR_ALL	/* trace everything */

TR_APPEND specifies whether the trace library should append (argument of TRUE) new tracing information onto the end of an existent file, or overwrite an existent file (argument of FALSE).

TR_FLOCK controls whether the trace library uses the flock library (see flock(2)).

The **trace** function writes logging information of *severity* to disk. The *trace_struct* arguments specified as a format string followed by format.

Code Examples:

```
#include <stdio.h>
#include <trace.h>
#include <version.h>

main () {
    Trace_Struct *protocol_trace;

    protocol_trace = New_Trace_Struct (TRACE_LOGFILE, "/tmp/my_logfile",
                                     TRACE_FLAGS, NORM,
                                     NULL);

    trace (NORM, protocol_trace, "This is a trace message");
    trace (TR_POLICY, protocol_trace, "This will not show up");

    Set_Trace_Struct (protocol_trace,
                     TRACE_FLAGS, TR_ALL,
                     TRACE_LOGFILE, "stdout",
                     NULL);

    trace (TR_POLICY, protocol_trace, "This will show up on the console");

    trace (FATAL, protocol_trace,
          "Now I will die... (this shows up in syslog)");
}
```

7. User Interactive Interface (UII)

The User Interactive Interface provides support for creating telnet vty management and configuration sessions.

Synopsis

```
cc [flag...] file... -lmrt [library...]  
#include <timer.h>  
  
int uii_add_command (int state, char *string, int *call_fn);  
int parse_line (u_char *line, char *format, ...);  
set_uui (uui_t *tmp, int first, ...);  
int uii_send_data (uui_connection_t *uui, ...);  
int uii_destroy_connection (uui_connection_t *connection);
```

Description

The function `set_uui` can be used to configure uui options:

```
set_uui (UII_PASSWORD, char *password)  
set_uui (UII_ACCESS_LIST, int num)  
set_uui (UII_PORT, int port_num)  
set_uui (UII_PROMT, state, char *prompt)
```

The function `Parse_line` is a utility for scanning and parsing a character string. Like the Unix `scanf` function, `Parse_line` takes a format string and variable arguments. Supported format arguments include:

```
%p   IPv4 prefix  
%P   IPv6 prefix  
%M   either IPv4 or IPv6  
%d   int  
%s   string  
%S   to end of line  
%a   IPv4 prefix with no bitmask (no |)  
%A   IPv6 prefix with no bitmask (no |)  
%n   ASCII name  
%D   integer (| ... max list)  
%m   IPv4 prefix
```

8. Interface Library

Synopsis

```
cc [flag ...] file... [library...] -linterface -lmrt
#include <mrt.h>
#include <interface.h>

int init_interfaces (trace_t *ltrace);
interface_t *find_interface (prefix_t *prefix);
interface_t *find_interface_byname (char *name);
LINKED_LIST *find_network (prefix_t *prefix);
interface_t *find_interface_local (prefix_t *prefix);
int local_interface (int family, u_char *cp);
interface_t *interface_iter (interface_iter_t **iter);
```

Description

Description goes here.

9. I/O Library

Because MRT is structured as a number of separate processes (BGP peers, MRS, etc.), it is necessary to provide a way for these processes to communicate. Solaris provides several different methods for programs to communicate (IPC methods), but the interface to these IPC methods is not standardized, and some amount of support code is invariably necessary to provide services which MRT requires but the Solaris IPC methods do not provide.

The I/O library provides an easy-to-use, standardized abstract interface to IPC. Several different IPC methods (file, stdin/stdout, and System V message queues) are supported. A message-queue key registry is provided, to allow processes to communicate without knowledge of each other's keys.

Synopsis

```
cc [flag ...] file... [library...] -lio -lstruct -lmrt

#include <mrt.h> #include <io.h>

int io_init();

int io_set(int key, int val, ...);

int io_write(long tstamp, short type, short subtype, int length, char
*value);

MRT_MSG_STRUCT *io_read(void);
```

Description

All programs using the I/O library must call `io_init` and `io_set` before any `io_write` or `io_read` calls are performed. In addition, the key registry server (`msgserver`) must be running in order for the `IO_INMSGQ` and `IO_OUTMSGQ` I/O types to work.

`io_set` is used to set up the input and output channels for the I/O library. The arguments to `io_set` are a sequence of key-value pairs, with the key being an I/O attribute, and the value being attribute-specific. Available keys are specified in [table 1](#).

Attribute	Value
IO_INNONE	(char *) NULL
IO_OUTNONE	(char *) NULL
IO_OUTFILE	filename (string)
IO_OUTAPPEND	filename (string)
IO_INFILE	filename (string)
IO_INMSGQ	client-ID (string)
IO_OUTMSGQ	client-ID (string)

Table 1: io_set Attributes

After `io_set` has been used to configure the I/O channels, `io_read` and `io_write` can be used to read and write data, respectively.

Examples

See the `src/programs/msgserver/testio.c` program for an example of a program that does I/O with message queues.

Bugs

- Each process can only have one input and one output channel.

10. linked_list.a Library

Synopsis

The linked_list.a library provides a generic set of operations for maintaining a linked list. The library is designed to free the user from maintaining the pointers required in a linked list. This library supports both intrusive and container style doubly linked lists and the lists may be automatically sorted by the insertion procedures. Functions and macros are also provided for iteration, processing, sorting, and conversion to other types of data structures.

Data and Function Types

```
#include <linked_list.h>
```

LINKED_LIST	- Structure containing information about a linked list.
LL_CONTAINER	- Structure used as a container for a non-intrusive linked list.
LL_POINTERS	- Structure which may be used to define the intrusive pointers in a structure to be held on a linked list.
enum LL_ATTR	- Enumeration of attributes possible on a linked list.
enum LL_ERROR	- Enumeration of errors possible in a linked list operation.
LL_DestroyProc	- Pointer to a function which destroys an item of data. void ()(DATA_PTR data);
LL_CompareProc	- Pointer to a function which compares two items of data. It should return <0 if d1 < d2, 0 if d1 == d2, and >0 if d1 > d2. int ()(DATA_PTR d1, DATA_PTR d2);
LL_FindProc	- Pointer to a function which compares an item of data to a key. It should return False (0) if data != key, and True (1) if data == key. int ()(DATA_PTR data, DATA_PTR key);
LL_ProcessProc	- Pointer to a function which processes an item of data. void ()(DATA_PTR data);
LL_ProcessPlusProc	- Pointer to a function which processes an item of data relative to a second argument. void ()(DATA_PTR data, DATA_PTR arg2);
LL_SortProc	- Pointer to a function which sorts a linked list. void ()(LINKED_LIST *ll, LL_CompareProc compare);
LL_ErrorProc	- Pointer to a function which handles/reports linked list errors. void ()(LINKED_LIST *ll, enum LL_ERROR error_num, char *function);

Attribute Description

The linked lists maintained by this library are doubly-linked terminating lists, i.e. the library does not support singly-linked lists or circular lists. Within these bounds a number of attributes may be set to define the operation of the list. Such attributes include whether the list is intrusive or container style, whether the list is automatically sorted, what information is reported to stdout with the debug library and various functions used to find, compare, and destroy items contained on the list. The most important difference is whether the list is

intrusive or container style as shown in Figure 1. The full list of attributes and their default values is shown in Table 1.

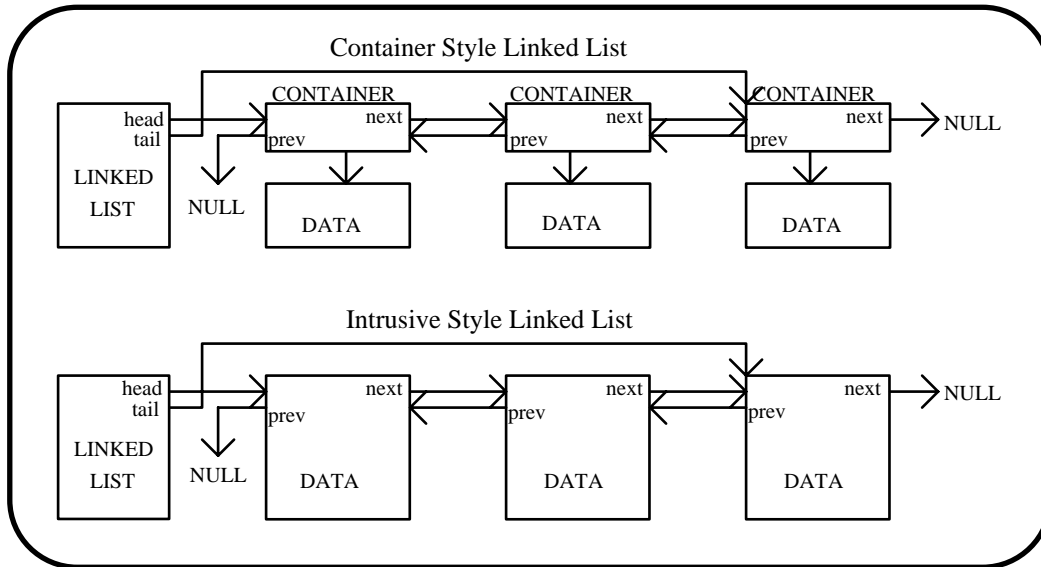


Figure 1: List Styles

Attribute	Type	Default
LL_Intrusive	int	False
LL_AutoSort	int	False
LL_ReportChange	int	False
LL_ReportAccess	int	False
LL_NextOffset	unsigned short	0
LL_PrevOffset	unsigned short	sizeof(DATA_PTR)
LL_FindFunction	LL_FindProc	NULL
LL_CompareFunction	LL_CompareProc	NULL
LL_DestroyFunction	LL_DestroyProc	NULL

Table 1: Attribute Definitions and Default Values

The `LL_Intrusive` attribute has the synonym attribute `LL_NoContainer`, and the antonyms `LL_NonIntrusive` and `LL_Container`. Any of these may be used to define the intrusive attribute.

Linked lists with the `LL_AutoSorted` attribute set will automatically insert new items in sorted order according to the attribute function `LL_CompareFunction` with each call to either `LL_Add()` or `LL_InsertSorted()`.

When the debug library is in use (see `Debugging` for more info.) the attributes `LL_ReportChange` and `LL_ReportAccess` determine (on a list by list basis) what information to report to `stdout` on each function call.

Offset values for the attributes `LL_NextOffset` and `LL_PrevOffset` can be calculated by the macro `LL_Offset(address, address)`. `LL_Offset()` takes the address of a structure to be stored on a list and the address of the next (prev) pointer in the structure and calculates the offset of the pointer. The order of the addresses is not important, i.e. `LL_Offset(&my_struct, &my_struct.next)` is equivalent to `LL_Offset(&my_struct.next, &my_struct)`. For convenience, the attribute `LL_PointersOffset` may be used to declare the offset of the struct `LL_POINTERS` within the user structure, this attribute sets both the next and prev offsets to the appropriate values. NOTE: If any offset is set, it automatically declares the list to be intrusive, thus the `LL_Intrusive` attribute does not necessarily need to be set.

Three functions may be installed as attributes on the linked list. The `LL_FindFunction` attribute is the function which will automatically be used when a call to `LL_Find()` or its derivatives is made. The `LL_CompareFunction` attribute is the function which will be used to compare two items of data when the list is being sorted or an element is being inserted into an auto-sorted list. The `LL_DestroyFunction` attribute is the function to be called to destroy (free) a single element when `LL_Remove()` or its derivatives is called.

Note: The Attributes `LL_Intrusive` and `LL_<type>Offset` can not be modified if there are already items on the linked list. The debug library checks for this case, but the optimized library does not and may result in a segment fault or loss of data if these attributes are modified.

Functions and Procedures

Creation, Modification, and Destruction

```
LINKED_LIST*  LL_Create(<argument list>†);  
void          LL_Destroy(LINKED_LIST *ll);  
void          LL_SetAttribute(LINKED_LIST *ll, enum LL_ATTR attr, type value);  
void          LL_GetAttribute(LINKED_LIST *ll, enum LL_ATTR attr, type* value);
```

Data Entry and Removal

```
DATA_PTR      LL_Append(LINKED_LIST *ll, DATA_PTR data);  
DATA_PTR      LL_Prepend(LINKED_LIST *ll, DATA_PTR data);  
DATA_PTR      LL_InsertSorted(LINKED_LIST *ll, DATA_PTR data, LL_CompareProc comp†);
```

```
DATA_PTR      LL_Add(LINKED_LIST *ll, DATA_PTR data);  
void          LL_Remove(LINKED_LIST *ll, DATA_PTR data);
```

Data Retrieval

```
DATA_PTR      LL_GetHead(LINKED_LIST *ll);  
DATA_PTR      LL_GetTail(LINKED_LIST *ll);  
DATA_PTR      LL_GetNext(LINKED_LIST *ll, DATA_PTR current);  
DATA_PTR      LL_GetPrev(LINKED_LIST *ll, DATA_PTR current);  
DATA_PTR      LL_Find(LINKED_LIST *ll, DATA_PTR key, LL_FindProc comp†);  
DATA_PTR      LL_FindNext(LINKED_LIST *ll, DATA_PTR key, DATA_PTR cur, LL_FindProc comp†);
```

Sorting

```
DATA_PTR      LL_Sort(LINKED_LIST *ll, LL_CompareProc comp†);  
DATA_PTR      LL_ReSort(LINKED_LIST *ll, DATA_PTR data);
```

Processing and Iteration

```
void          LL_Process(LINKED_LIST *ll, LL_ProcessProc process);  
void          LL_ProcessPlus(LINKED_LIST *ll, LL_ProcessPlusProc process, DATA_PTR arg2);  
MACRO°       LL_Iterate(LINKED_LIST *ll, DATA_PTR data)
```

Error Handling

```
LL_ErrorProc  Set_LL_Handler(LL_ErrorProc handler);
```

Notes

[†] *<argument list>* is a NULL terminated attribute-value pair list.

[†] *comp* is an optional argument, see descriptions for more information.

[°] *MACRO* indicates that the procedure is a convenience macro emulating a for-loop.

Description

`LL_Create()` creates a linked list, initializes its attributes to the default values, sets the attributes defined in the *<argument list>* and returns a pointer to the linked list.

`LL_Destroy()` destroys a linked list and all the memory retained by the linked list. This is done by first calling `LL_Clear()` to remove each item from the list via a call to `LL_Remove()`. If the `LL_DestroyFunction` attribute is set, the destructor function will be called for each item on the list. Once the list is empty, the memory retained for the linked list structure is freed via a call to `Delete()`.

`LL_SetAttribute()` sets a single attribute in the linked list. The argument *attr* should be one of the linked list attributes, and its *value* should be of the appropriate type for that attribute. `LL_GetAttribute()` retrieves the value of a single attribute from a linked list. The argument

value should be a pointer to an item of the appropriate type for *attr*. The value of the attribute will be placed into the item pointed to by *value*.

LL_Append() appends *data* onto the tail of the linked list. LL_Prepend() prepends *data* onto the head of the linked list. LL_InsertSorted() inserts *data* into a sorted list. The argument *comp* is an optional argument, if the LL_CompareFunction attribute is set, this function will be used regardless of value of *comp*, however if the attribute is not set, the argument *comp* must exist. LL_Add() adds data onto the linked list by either using LL_InsertSorted() or LL_Append() depending on whether the list is auto-sorted or not. Each of these functions returns the *data* added onto the list. If an error occurs during the addition, they return NULL.

LL_Remove() removes *data* from a linked list. LL_Remove() first unlinks *data* from the list and then if the LL_DestroyFunction attribute is set the destructor function is called to destroy (free) *data*.

LL_GetHead() returns the item of data on the head of the list. LL_GetTail() returns the item of data on the tail of the list. LL_GetNext() returns the next item of data, i.e. *data->next*. LL_GetPrev() returns the previous item of data, i.e. *data->prev*. For LL_GetNext() and LL_GetPrev() *data* must be either NULL or an item of data on the list. If *data* is NULL LL_GetNext() returns the head and LL_GetPrev() returns the tail. If *data* is not on the list the function is likely to segment fault (the debug library calls the error handler with the error LL_NoMember and returns NULL).

LL_Find() iterates through the list beginning at the head until an item of data is found matching *key* according to the function *comp*. LL_FindNext() iterates through the list beginning at *cur->next* until an item of data is found matching *key* according to the function *comp*. If an item is found it is returned, however if no matching items are found the function returns NULL. The argument *comp* is an optional argument, i.e. you may call either LL_Find(list, key) or LL_Find(list, key, comp). If the LL_FindFunction attribute is set, this function will be used regardless of the value of *comp*, however if the attribute is not set the argument *comp* must exist.

LL_Sort() sorts a linked list according to the function *comp*. The argument *comp* is an optional argument, i.e. you may either call LL_Sort(list) or LL_Sort(list, comp). If the LL_CompareFunction attribute is set, this comparison function is used regardless of the value of *comp*, however, if the attribute is not set, the argument *comp* must exist.

LL_ReSort() resorts a single element *data* after it has changed such that it is longer in sorted order.

LL_Process() iterates through the linked list running the function *process(data)* on each item of data in the list. LL_ProcessPlus() iterates through the linked list running the function *process(data, arg2)* on each item of data in the list. LL_Iterate() is a convenience macro which expands to a for loop of the type:

```
for(data = LL_GetHead(ll); data; data = LL_GetNext(ll, data)).
```

Note: It is inadvisable to change the argument *data*'s next or prev pointers or to delete the argument *data* from the list within any of these iteration loops as this may cause the loop to

terminate early, never terminate, or segment fault. At the very least, at the end of each loop *data* must be either NULL or an item of data on the list. i.e. the following code is permissible:

```
LL_Iterate(ll, data) {
    if (some condition) {
        DATA_PTR tmp = LL_GetPrev(ll, data);
        LL_Remove(ll, data);
        data = tmp; /* ensure that data points to a real value */
    }
}
```

Set_LL_Handler() installs the function *handler* as the error handler and returns the previously installed handler. The default error handler reports the error that occurred to stderr (stdout in the debug library).

Extended Functions and Procedures

Modification

```
void          LL_Clear(LINKED_LIST *ll);

void          LL_SetAttributes(LINKED_LIST *ll, <argument list>†);

void          LL_GetAttributes(LINKED_LIST *ll, <argument list>†);
```

Data Insertion

```
DATA_PTR      LL_InsertAfter(LINKED_LIST *ll, DATA_PTR data, DATA_PTR relative_to);

DATA_PTR      LL_InsertBefore(LINKED_LIST *ll, DATA_PTR data, DATA_PTR relative_to);
```

Data Retrieval

```
DATA_PTR      LL_FindFromTail(LINKED_LIST *ll, DATA_PTR key, LL_FindProc comp†);

DATA_PTR      LL_FindPrev(LINKED_LIST *ll, DATA_PTR key, DATA_PTR cur, LL_FindProc comp†);
```

Sorting

```
void          LL_BubbleSort(LINKED_LIST *ll, LL_CompareProc compare);

void          LL_QuickSort(LINKED_LIST *ll, LL_CompareProc compare);

void          LL_MergeSort(LINKED_LIST *ll, LL_CompareProc compare);

LL_SortProc   Set_LL_Sorter(LL_SortProc sort_fn)
```

Iteration

```
MACRO°       LL_IterateFind(LINKED_LIST *ll, DATA_PTR key, DATA_PTR data)
```

Conversion

```
DATA_PTR*     LL_ToArray(LINKED_LIST *ll, DATA_PTR *array, int *nel);

LINKED_LIST*  LL_FromArray(LINKED_LIST *ll, DATA_PTR *array, int nel);
```

Misc.

```
unsigned      LL_GetCount(LINKED_LIST *ll);
```

Notes

† *<argument list>* is a NULL terminated attribute-value pair list.

‡ *comp* is an optional argument, see descriptions for more information.

◦ *MACRO* indicates that the procedure is a convenience macro emulating a for-loop.

Extended Function Description

LL_Clear() clears all data from a linked list via calls to the function LL_Remove(). If the attribute LL_DestroyFunction is set the destructor will be called to destroy (free) each item of data on the list. The LL_Clear() function leaves the list's attributes intact, i.e. intrusive, auto-sorted etc..

LL_SetAttributes() sets a number of attributes on the linked list. The *<argument list>* is a NULL terminated list of attribute-value pairs. Each value argument should match the type designated for its attribute. LL_GetAttributes() retrieves a number of attributes from the linked list. The *<argument list>* is a NULL terminated list of attribute-value pointer pairs. Each value argument should be a pointer to an item of the appropriate type for the attribute where the function will return the attribute's value.

LL_InsertAfter() inserts *data* after the item *relative_to*. LL_InsertBefore inserts *data* before the item *relative_to*. Both functions return the item of *data* inserted onto the list. The argument *relative_to* must either be NULL or an item of data on the list. If *relative_to* is NULL, LL_InsertAfter() inserts the item on the head and LL_InsertBefore() inserts the item on the tail. If *relative_to* is not on the list, the error handler will be called with the error LL_NoMember() and the function will return NULL.

LL_FindFromTail() and LL_FindPrev() operate in the same manner as LL_Find() and LL_FindNext() except that they search backwards through the list for a matching item.

LL_BubbleSort(), LL_QuickSort(), and LL_MergeSort() provide three different methods of sorting a linked list. LL_BubbleSort() sorts the list in place, requiring no additional memory. LL_QuickSort() and LL_MergeSort() first converts the list to an array, then calls ARRAY_<type>Sort() to perform the actual sorting, and finally converts the result back into the linked list.

LL_IterateFind() is a convenience macro which searches for an item of data matching *key* using the installed attribute LL_FindFunction as the comparison function. This macro is restricted by all the restrictions placed on both the LL_Iterate functions and the LL_Find functions.

LL_ToArray() converts a linked list to an array and returns a pointer of type DATA_PTR* to the first item on the list. The argument *array* must be a pointer to an array of sufficient size to hold the data on the list or NULL in which case an array of the appropriate size is allocated. The argument *nel* is a pointer used to return the number of elements in the array (it

may be NULL in which case the size is not returned). If an array pointer is given, but the array is of insufficient size, it is likely that the function will segment fault while adding data onto the array.

LL_FromArray() converts an array to a linked list and returns a pointer to the linked list. The argument *array* must be a pointer to the first element of an array of pointers and *nel* must be the number of elements in the array. The argument *ll* is a pointer to the linked list to fill. If *ll* is NULL a default linked list will be created otherwise all the data on the list *ll* will be ignored (removed without calling the destructor function), but the attributes will remain intact.

LL_GetCount() returns the number of items currently held in the linked list. It may be used to allocate sufficient space to hold an array prior to the LL_ToArray() conversion.

Fast Functions and Procedures

Intrusive Style Lists

```
DATA_PTR LL_IntrGetHead(LINKED_LIST *ll);
DATA_PTR LL_IntrGetTail(LINKED_LIST *ll);
DATA_PTR LL_IntrGetNext(LINKED_LIST *ll, DATA_PTR current);
DATA_PTR LL_IntrGetPrev(LINKED_LIST *ll, DATA_PTR current);
MACRO° LL_IntrIterate(LINKED_LIST *ll, DATA_PTR data)
```

Container Style Lists

```
DATA_PTR LL_ContGetHead(LINKED_LIST *ll);
DATA_PTR LL_ContGetTail(LINKED_LIST *ll);
DATA_PTR LL_ContGetNext(LINKED_LIST *ll, DATA_PTR current);
DATA_PTR LL_ContGetPrev(LINKED_LIST *ll, DATA_PTR current);
MACRO° LL_ContIterate(LINKED_LIST *ll, DATA_PTR data)
```

Note

° *MACRO* indicates that the procedure is a convenience macro emulating a for-loop.

Fast Function and Macro Description

The fast functions are macro expansions which will in general operate faster than the generic functions. This is mostly due to a decrease in the size of the macro expansion by approximately half, and the removal of a comparison to determine which style of list to retrieve the data from. All the restrictions placed on these functions are the same as their generic counterparts plus the LL_Intr<>() functions can only be used on intrusive lists while LL_Cont<>() functions can only be used on container style lists.

Override Functions and Procedures

Destruction and Removal

```
void      LL_ClearFn(LINKED_LIST *ll, LL_DestroyProc destroy);
void      LL_DestroyFn(LINKED_LIST *ll, LL_DestroyProc destroy);
void      LL_RemoveFn(LINKED_LIST *ll, LL_DestroyProc destroy);
```

Data Insertion

```
DATA_PTR LL_InsertSortedFn(LINKED_LIST *ll, DATA_PTR data, LL_CompareProc compare);
```

Data Retrieval

```
DATA_PTR LL_FindFn(LINKED_LIST *ll, DATA_PTR key, LL_FindProc compare);
DATA_PTR LL_FindFromTailFn(LINKED_LIST *ll, DATA_PTR key, LL_FindProc compare);
DATA_PTR LL_FindNextFn(LINKED_LIST *ll, DATA_PTR key, DATA_PTR cur, LL_FindProc compare);
DATA_PTR LL_FindPrevFn(LINKED_LIST *ll, DATA_PTR key, DATA_PTR cur, LL_FindProc compare);
```

Sorting

```
void      LL_SortFn(LINKED_LIST *ll, LL_CompareProc compare);
```

Iteration

```
MACRO°    LL_IterateFindFn(LINKED_LIST *ll, DATA_PTR key, DATA_PTR data, LL_FindProc compare)
```

Note

° *MACRO* indicates that the procedure is a convenience macro emulating a for-loop.

Override Function Description

The override functions provide the same operation as their generic counterparts with the exception that they override the function attributes installed on the linked list. i.e. The function *destroy* or *compare* is used rather than the built in attributes `LL_DestroyFunction`, `LL_FindFunction`, and `LL_CompareFunction`.

Debugging Functions

```
int      LL_Verify(LINKED_LIST *ll);
void     LL_Print(LINKED_LIST *ll, LL_ProcessProc print);
LL_CONTAINER* LL_GetContainer(LINKED_LIST *ll, DATA_PTR data);
```

Debugging Function Description

`LL_Verify()` verifies the linked list by ensuring that (1) the attribute set is valid, (2) that the head and tail are actually the head and tail of the list, (3) for each item verify that `data->next->prev == data`, (4) on an auto-sorted list verify that the list is in sorted order, and (5) that the count is equal to the number of items.

LL_Print() first verifies the list, and then calls the function *print* on each item of data on the list to print it out. If *print* is NULL the default printer function will be used which prints the data's address to stdout.

LL_GetContainer() retrieves data's container on a container style linked list.

Debugging

Linked list operations can be debugged by defining either LL_DEBUG or STRUCT_DEBUG in any ".c" file prior to the inclusion of the <linked_list.h> header file and by using the list_debug.a library in place of the linked_list.a library at compile time. Defining LL_DEBUG causes each macro to be expanded to force a function call to the library (rather than the optimized expansion). The debug library makes additional checks to verify the call's arguments and the correctness of internal variables used. In addition, the attributes LL_ReportAccess and LL_ReportChange may be set on a list whereby each function call will report the operation performed to stdout.

Notes

This library relies on the array.a library to perform the LL_QuickSort() and LL_MergeSort() operations which also rely on the stack.a library. This library also relies on the New.a library to allocate and deallocate memory for the linked list structure, the container structures, and the arrays in the conversion operations.

The call New(LINKED_LIST) should not be used to create a linked list as the internal data may not be initialized correctly, thus always use LL_Create() to create a linked list.

It is extremely important that within iterations and processing calls, the current item of data neither has its pointers changed nor is removed. Doing so may cause unpredictable results. This also applies to any function installed on the linked list or passed as an argument to one of the functions (with the obvious exception of the destructor function).

Errors

Any of the functions or procedures in this library may fail as a result of the errors described below. At the occurrence of such a failure, the linked list error handler is called to report the error to the user. The global variable `char *LL_errlist[]` provides a short description of the error which occurred (see the examples for information on how to use the `*LL_errlist[]` variable).

- LL_UnknownErr - Undocumented or unknown error.
- LL_MemoryErr - Unable to allocate the necessary space to complete the operation.
- LL_NoMember - One of the data arguments passed to the function is not a member of the list.
- LL_ListNotEmpty - Attempt to set an attribute defining the list on a non-empty list.
- LL_ListCorrupted - The data in the list structure does not match the information held on the list.

-
- LL_BadOperation - Attempt to perform an operation on a list which is not of the appropriate type.
 - LL_BadAttributes - The set of attributes is invalid.
 - LL_BadArgument - One of the arguments passed to the function is invalid.

Examples

Example Makefile

```

INCDIR=-I$(HOME)/struct/include
LIBDIR=-L$(HOME)/struct/lib
LIBS=-llinked_list -larray -lstack -lNew

all: my_file

my_file: my_file.o
    $(CC) -o my_file myfile.o $(LIBDIR) $(LIBS)

my_file.o: my_file.c
    $(CC) -O $(INCDIR) -c my_file.c

```

Example Use #1: Container Style List

```

#include <linked_list.h>
#include <New.h>          /* used for New()          */
#include <stdio.h>        /* used for printing  */
#include <string.h>       /* used for comparison */

typedef struct blee {
    char *name;
    int  val;
} BLEE;

#define SIZE 10
const struct blee array[SIZE] = {
    {"Joe",    0}, {"Frank",  1}, {"Buddy",  2}, {"Jim",    3}, {"George", 4},
    {"Jane",   5}, {"Jay",    6}, {"Laura",  7}, {"Craig",  8}, {"Bill",   9},
};

int Compare (BLEE *b1, BLEE *b2) { return(strcmp(b1->name, b2->name)); }
int Find    (BLEE *b, char *key) { return(!strcmp(b->name, key)); }
void Destroy (BLEE *b)          { Delete(b->name); Delete(b); }
void Print   (BLEE *b)          { printf("0x%.8x = %s, %d\n", b, b->name, b->val); }

main () {
    LINKED_LIST *list1, *list2;
    BLEE        *b;
    int         i;

    list1 = LL_Create(NULL);
    list2 = LL_Create(LL_AutoSort, True, LL_CompareFunction, Compare, NULL);

    for (i = 0; i < SIZE; i++) {
        b = New(BLEE);
        b->name = strdup(array[i].name);
        b->val  = array[i].val;
        LL_Add(list1, b); /* Add it onto list 1 */
        LL_Add(list2, b); /* Add it sorted into list 2 */
    }
}

```

```

    LL_Print(list1, Print);
    LL_Print(list2, Print);

    b = LL_Find(list1, "Joe", Find);
    LL_Remove(list1, b);      /* Remove Joe from the list */

    LL_SetAttribute(list2, LL_FindFunction, Find);
    LL_SetAttribute(list2, LL_DestroyFunction, Destroy);

    b = LL_Find(list2, "Joe");
    LL_Remove(list2, b);      /* Remove and Destroy "Joe" */

    LL_Destroy(list1);
    LL_Destroy(list2);
}

```

Example Use #2: Intrusive Style List

```

#include <linked_list.h>
#include <New.h>          /* used for New()      */
#include <stdio.h>       /* used for printing */
#include <string.h>      /* used for comparison */

typedef struct blee {
    char *name; int val;
} BLEE;

typedef struct blee_on_list1 {
    char *name; int val;
    struct blee_on_list1 *next, *prev;
} BLEE_1;

typedef struct blee_on_list2 {
    char *name; int val;
    LL_POINTERS ptrs;
} BLEE_2;

#define SIZE 10
const struct blee array[SIZE] = {
    {"Joe", 0}, {"Frank", 1}, {"Buddy", 2}, {"Jim", 3}, {"George", 4},
    {"Jane", 5}, {"Jay", 6}, {"Laura", 7}, {"Craig", 8}, {"Bill", 9},
};

int Compare (BLEE_2 *b1, BLEE_2 *b2) { return(strcmp(b1->name, b2->name)); }
int Find (BLEE_2 *b, char *key) { return(!strcmp(b->name, key)); }
void Destroy1 (BLEE_1 *b) { Delete(b->name); Delete(b); }
void Destroy2 (BLEE_2 *b) { Delete(b->name); Delete(b); }
void Print1 (BLEE_1 *b) { printf("0x%.8x = %s, %d\n", b, b->name, b->val); }
void MyHandler(LINKED_LIST *ll, enum LL_ERROR error_num, char *fn) {
    printf("Error in list 0x%.8x function %s error %d = \"%s\"\n",
        ll, fn, error_num, LL_errlist[error_num]);
}

main () {
    LINKED_LIST *list1, *list2;
    BLEE_1 *b1, tmp1;
    BLEE_2 *b2, tmp2;
    int i;
    LL_ErrorProc old_handler;

    old_handler = Set_LL_Handler(MyHandler);
    list1 = LL_Create(LL_Intrusive, True, LL_NextOffset, LL_Offset(&tmp1, &tmp1.next),
        LL_PrevOffset, LL_Offset(&tmp1, &tmp1.prev),
        LL_DestroyFunction, Destroy1, NULL);

    list2 = LL_Create(LL_Intrusive, True, LL_PointersOffset, LL_Offset(&tmp1, &tmp2.ptrs),
        LL_AutoSort, True, LL_CompareFunction, Compare,
        LL_DestroyFunction, Destroy2, LL_FindFunction, Find, NULL);
}

```

```
for (i = 0; i < SIZE; i++) {
    b1 = New(BLEE_1);                b2 = New(BLEE_2);
    b1->name = strdup(array[i].name); b2->name = strdup(array[i].name);
    b1->val  = array[i].val;         b2->val  = array[i].val;
    LL_Add(list1, b1); /* Add it onto list 1 */
    LL_Add(list2, b2); /* Add it sorted into list 2 */
}

b2 = LL_Find(list2, "Joe");
LL_Remove(list2, b2); /* Remove and Destroy "Joe" */
LL_FindNext(list2, "Laura", b2); /* ERROR: No such member: b2 (it was removed) */
/* Note: The optimized library may not catch this error, it may segment fault, or
   it may find "Laura". The debug library will always catch the error. */

LL_Iterate(list2, b2) {
    printf("Hi, I am at %s, %d\n", b2->name, b2->val);
}
LL_Destroy(list1); LL_Destroy(list2);
}
```

Author

Jonathan L. DeKock

11. hash.a Library

Synopsis

The hash.a library provides a generic set of operations for maintaining a hash table. The library is designed to free the user from maintaining the hash array and pointers. Functions and macros are provided to create and destroy hash tables, to convert hash tables to/from other types, and to insert, remove, lookup, iterate through and process items of data on a hash table.

Data and Function Types

```
#include <hash.h>
```

- | | |
|----------------------|---|
| HASH_TABLE | - Structure containing information about a hash table. |
| HASH_CONTAINER | - Structure used as a container on a container style hash table. |
| enum HASH_ATTR | - Enumeration of attributes possible on a hash table. |
| enum HASH_ERROR | - Enumeration of errors possible in a hash table function. |
| HASH_HashProc | - Pointer to a function to convert a key to a hash index.
It should return a value in the range [0:size-1].
<code>unsigned ()(DATA_PTR key, unsigned size);</code> |
| HASH_LookupProc | - Pointer to a function to compare two keys.
It should return False (0) if data_key != key and True (1) if data_key == key.
<code>int ()(DATA_PTR data_key, DATA_PTR key);</code> |
| HASH_DestroyProc | - Pointer to a function which destroys an item of data.
<code>void ()(DATA_PTR data);</code> |
| HASH_ProcessProc | - Pointer to a function which processes an item of data.
<code>void ()(DATA_PTR data);</code> |
| HASH_ProcessPlusProc | - Pointer to a function which processes an item of data relative to a 2 nd argument.
<code>void ()(DATA_PTR data, DATA_PTR arg2);</code> |
| HASH_ErrorProc | - Pointer to a function which handles/reports hash table errors.
<code>void ()(HASH_TABLE *h, enum HASH_ERROR error_num, char *fn);</code> |
| HASH_PrintProc | - Pointer to a function which prints out an item of data.
<code>void ()(unsigned index, DATA_PTR data, DATA_PTR key);</code> |

Attribute Description

The hash tables maintained by this library are link-style hash tables which allow more than one item of data to be hashed to a single index. The linked lists used for this purpose are singly linked lists (not the generic lists provided by the linked_list.a library) and are optimized for fast lookup of data in the list. A number of attributes may be set to define the operation of the hash table including information about how the structures are to be hashed and functions for hashing, lookup, destruction. The attributes are defined in Table 1.

Attribute	Type	Default
HASH_EmbeddedKey	int	False
HASH_Intrusive	int	False
HASH_ReportChange	int	False
HASH_ReportAccess	int	False
HASH_DynamicResize	unsigned	False (0)
HASH_KeyOffset	unsigned short	0
HASH_NextOffset	unsigned short	sizeof(DATA_PTR)
HASH_HashFunction	HASH_HashProc	(See Descr.)
HASH_LookupFunction	HASH_LookupProc	(See Descr.)
HASH_DestroyFunction	HASH_DestroyProc	NULL

Table 1: Attribute Definitions and Default Values

Each structure on a hash table must retain a key by which it is hashed. This key may be either a pointer or embedded into the structure as defined by the `HASH_EmbeddedKey` attribute. By default, the hash table assumes that the key is a pointer to a NULL terminated character string. Either way, the hashing and lookup functions receive a pointer to the key.

<u>Non-Embedded Key (default)</u>	<u>Embedded Key</u>
<pre>struct my_struct { ... char *name; /* key */ ... };</pre>	<pre>struct my_struct { ... char name[100]; /* key */ ... };</pre>

The `HASH_Intrusive` attribute determines if the hash table's lists will be intrusively linked or linked by containers. By default the lists are container style. However, intrusive lists are more optimal in both time and space. This attribute has the synonym `HASH_NoContainer` and the antonyms `HASH_NonIntrusive` and `HASH_Container`. Any of these may be used to define the intrusive attribute.

When the debug library is in use (see Debugging for more info.) the attributes `HASH_ReportChange` and `HASH_ReportAccess` determine (on a hash table by hash table basis) what information to report to stdout after each function call.

The `HASH_DynamicResize` attribute is used to define when (if ever) to resize the hash table to make the hashing more optimal. On an insertion, if the hash table's Count/Size ratio is greater than this value, the hash table will double in size and re-hash all the items into the new table. By default resizing is turned off (0).

The offsets `HASH_KeyOffset` and `HASH_NextOffset` are used to determine the locations of the key and intrusive next pointer in the structure to be hashed. A key offset must always be provided while a next offset is only necessary on intrusive tables. The table is automatically made intrusive if the next offset attribute is set. The macro `HASH_Offset` may be used to calculate the offset of the items in the table. This macro takes the addresses of a structure and the item in it and calculates the offset. The order of the parameters is not important to the macro, i.e. `HASH_Offset(&my_struct, &my_struct.name)` is equivalent to

```
HASH_Offset(&my_struct.name, &my_struct).
```

The attribute `HASH_HashFunction` defines the hashing function to use. The value returned must be in the interval `[0:size-1]`, indices out of this range may result in a call to the error handler or a segment fault. The default hash function hashes NULL terminated character strings into a relatively random distribution.

The attribute `HASH_LookupFunction` defines the function to use to when scanning through a linked list on a given hash index. The function compares two keys and True (1) if they are a match and False (0) if they are not. The default lookup function compares NULL terminated character strings for an exact match.

The attribute `HASH_DestroyFunction` defines the function to use when an item of data is removed from a hash table. The purpose of this function is to free (release) the memory allocated for the structure on the hash table.

Functions and Procedures

Creation and Destruction and Modification

```
HASH_TABLE*   HASH_Create(unsigned size, <argument list>†);
void          HASH_Destroy(HASH_TABLE *h);
void          HASH_SetAttribute(HASH_TABLE *h, enum HASH_ATTR attr, type value);
void          HASH_GetAttribute(HASH_TABLE *h, enum HASH_ATTR attr, type *value);
```

Insertion and Removal

```
DATA_PTR      HASH_Insert(HASH_TABLE *h, DATA_PTR data);
void          HASH_Remove(HASH_TABLE *h, DATA_PTR data);
void          HASH_RemoveByKey(HASH_TABLE *h, DATA_PTR key);
DATA_PTR      HASH_ReHash(HASH_TABLE *h, DATA_PTR data, DATA_PTR old_key);
```

Data Retrieval

```
DATA_PTR      HASH_Lookup(HASH_TABLE *h, DATA_PTR key);
```

Processing and Iteration

```
void          HASH_Process(HASH_TABLE *h, HASH_ProcessProc process);  
void          HASH_ProcessPlus(HASH_TABLE *h, HASH_ProcessPlusProc process, DATA_PTR arg2);  
MACRO°       HASH_Iterate(HASH_TABLE *h, DATA_PTR data)
```

Error Handling

```
HASH_ErrorProc Set_HASH_Handler(HASH_ErrorProc handler);
```

Notes

- † *<argument list>* is a NULL terminated attribute-value pair list.
- ° *MACRO* indicates that the procedure is a convenience macro emulating a for-loop.

Description

`HASH_Create()` creates a hash table with *size* indices, initializes its attributes to the default values, sets the attributes defined in the *<argument list>*, and returns a pointer to the new hash table.

`HASH_Destroy()` destroys a hash table and all the information retained by the hash table. This is done by first calling `HASH_Clear()` to remove each item from the table via calls to `HASH_Remove()`. If the `HASH_DestroyFunction` attribute is set the destructor function is called to free the item of data. Once the table is empty, the table and the hash-index array are freed via calls to `Delete()`.

`HASH_SetAttribute()` sets a single attribute in the hash table. The argument *attr* should be one of the hash table attributes, and its *value* should be of the appropriate type for that attribute. `HASH_GetAttribute()` retrieves the value of a single attribute from a linked list. The argument *value* should be a pointer to an item of the appropriate type for *attr*. The value of the attribute will be placed into the item pointed to by *value*.

`HASH_Insert()` inserts an item into the hash table. `HASH_Remove()` removes an item from a hash table. `HASH_RemoveByKey()` finds the item matching *key* and removes it from the table.

`HASH_ReHash()` rehashes an item of data after the key has changed. The argument *old_key* must have the value of the previous key in it so that the item can be found on the hash table. This function finds the item on the table, does a quick remove (without the destructor function) and re-inserts into the hash table.

`HASH_Lookup()` finds an item of data in the table which matches the argument *key* and returns a pointer to the data. If no such item exists the function returns NULL.

`HASH_Process()` and `HASH_ProcessPlus()` are used for running a function *process* across every item in a hash table. `HASH_ProcessPlus()` takes a second argument *arg2* which is passed to each function call so that the processing function can process the data relative to

arg2. `HASH_Iterate()` is a convenience macro which expands to a for loop to iterate through all items of data in the hash table. It is inadvisable to change the argument *data*'s key or pointers while in an iteration or process loop. It is especially important that the item *data* is not removed from the hash table during such a loop. Doing so may cause the loop to terminate early, never terminate, or segment fault.

`Set_HASH_Handler()` installs the function handler as the error handler for all hash tables and returns the previously installed handler. The default handler reports the error to `stderr` (stdout with the debug library).

Extended Functions and Procedures

Destruction and Modification

```
void          HASH_Clear(HASH_TABLE *h);
void          HASH_ChangeSize(HASH_TABLE *h, unsigned size);
void          HASH_SetAttributes(HASH_TABLE *h, <argument list>†);
void          HASH_GetAttributes(HASH_TABLE *h, <argument list>†);
```

Conversion

```
DATA_PTR*    HASH_ToArray(HASH_TABLE *h, DATA_PTR *array, unsigned *nel);
HASH_TABLE*  HASH_FromArray(HASH_TABLE *h, DATA_PTR *array, unsigned nel,
                           unsigned offset‡, int embedded‡);
LINKED_LIST* HASH_ToLinkedList(HASH_TABLE *h, LINKED_LIST *ll);
HASH_TABLE*  HASH_FromLinkedList(HASH_TABLE *h, LINKED_LIST *ll,
                                 unsigned offset‡, int embedded‡);
```

Misc.

```
unsigned     HASH_GetCount(HASH_TABLE *h);
unsigned     HASH_GetSize(HASH_TABLE *h);
```

Notes

† *<argument list>* is a NULL terminated attribute-value pair list.

‡ *offset* and *embedded* are an optional argument, see descriptions for more information.

Extended Function Description

`HASH_Clear()` removes all data from a hash table via calls to the function `HASH_Remove()`. If the attribute `HASH_DestroyFunction` is set, the destructor function will be called to destroy (free) each item of data on the list. The `HASH_Clear()` function leaves the table's attributes intact, i.e. embedded key, intrusive, etc..

`HASH_SetAttributes()` and `HASH_GetAttributes()` perform the same operation as their singular forms with the exception that they take a variable number of attributes. The *<argument list>* is a NULL terminates list of attribute-value pairs (in the Get function it is

attribute-value pointer pairs).

`HASH_ToArray()` converts a hash table to an array of pointers. The argument *array* must either be a pointer to an array of sufficient size to hold the number of items in the table or `NULL` in which case an array of sufficient size is allocated. The argument *nel* (if non-`NULL`) is the location where the function will return the number of items placed in the array.

`HASH_FromArray()` converts an array to a hash table. The argument *array* must be a pointer to the first item of an array of pointers to structures and *nel* is the number of elements in the array. The argument *h* is the hash table to use for all the items in the array. If *h* is non-`NULL` the *offset* and *embedded* arguments are ignored (they do not need to be placed in the function call) and all data currently in the hash table will be removed (without calling the destructor function) and the array will be converted in. If *h* is `NULL` a default hash table with *nel* indices will be created and the arguments *offset* and *embedded* will be used to define the structure's hashing properties.

`HASH_ToLinkedList()` converts a hash table to a linked list. If the argument *ll* is `NULL` a default linked list will be created. Additions to the linked list are made by calling the function `LL_Add()` for each item of data.

`HASH_FromLinkedList()` converts a linked list to a hash table. If the argument *h* is `NULL` the arguments *offset* and *embedded* are used to create a default hash table whose size is the same as the linked list count.

If *h* is non-`NULL` the *offset* and *embedded* arguments are ignored and all the data in the hash table is removed (without calling the destructor function).

`HASH_GetCount()` and `HASH_GetSize()` return the number of elements and the number of indices (respectively) in the hash table *h*.

Override Functions and Procedures

```
void HASH_ClearFn(HASH_TABLE *h, HASH_DestroyProc destroy);
void HASH_DestroyFn(HASH_TABLE *h, HASH_DestroyProc destroy);
void HASH_RemoveFn(HASH_TABLE *h, DATA_PTR data, HASH_DestroyProc destroy);
void HASH_RemoveByKeyFn(HASH_TABLE *h, DATA_PTR key, HASH_DestroyProc destroy);
```

Override Function Description

The override functions perform the same operation as their generic counterparts except that the argument *destroy* is used to destroy the data rather than the destructor function attribute installed on the table.

Debugging Functions

```
int          HASH_Verify(HASH_TABLE *h);  
void        HASH_Print(HASH_TABLE *h, HASH_PrintProc print);  
HASH_CONTAINER* HASH_GetContainer(HASH_TABLE *h, DATA_PTR data);  
HASH_CONTAINER* HASH_GetContainerByKey(HASH_TABLE *h, DATA_PTR key);
```

Debugging Function Description

`HASH_Verify()` verifies that all data in the hash table is correct. It checks that each item of data is hashed at the correct index and that the count matches the expected count.

`HASH_Print()` first verifies the table, and then prints all the data out via calls to the function *print* for each item of data on the hash table. If *print* is NULL the default printing function will be used which prints out the index, data's address, the key's address, and the character string key. Note: Do not use NULL if data is not hashed by a character string.

`HASH_GetContainer()` returns the container of the item *data*. `HASH_GetContainerByKey()` returns the container of the item found by a lookup on *key*.

Debugging

Hash table operations can be debugged by defining either `HASH_DEBUG` or `STRUCT_DEBUG` in any ".c" file prior to the inclusion of the `<hash.h>` header file and by using the `hash_debug.a` library in place of the `hash.a` library at compile time. Defining `HASH_DEBUG` causes each operation to force a function call rather than the (occasional) faster macro expansion. The debug library makes additional checks to verify the validity of the arguments passed to each function and the correctness of the internal variables (including the index returned by the hash function). In addition, the attributes `HASH_ReportChange` and `HASH_ReportAccess` may be set resulting in each function call on a particular list reports the operation performed to stdout.

Notes

This library relies on the `New.a` library to perform the allocation and deallocation of memory for the tables, their hash-index arrays, and the containers on container style lists. It also relies on the `linked_list.a` library for the conversion functions. In total, using this library requires that the `linked_list.a`, `array.a`, `stack.a`, and `New.a` libraries be linked in at compile time.

The call `New(HASH_TABLE)` should not be used to create a hash table as the internal data may not be initialized correctly, nor will the allocation of the array take place. Therefore always use the `HASH_Create()` function to create a hash table.

It is extremely important within iterations and processing calls that the current item of data does not have its key or pointers changed in any way (including an attempt to remove the item of data). Doing so may cause unpredictable results. This also applies to the functions

installed on the hash table (with the obvious exception of the destructor function).

Errors

Any of the functions or procedures in this library may fail as a result of the errors described below. At the occurrence of such a failure, the hash table error handler is called to report the error to the user. The global variable `char *HASH_errlist[]` provides a short description of the error which occurred (see examples for information on how to use the `*HASH_errlist[]` variable).

<code>HASH_UnknownErr</code>	- Undocumented or unknown error.
<code>HASH_MemoryErr</code>	- Unable to allocate the necessary space to complete the operation.
<code>HASH_TableCorrupted</code>	- The hash table is corrupted in some way.
<code>HASH_BadIndex</code>	- Index returned by the hash function is too large.
<code>HASH_BadArgument</code>	- One or more of the arguments to the function are invalid.
<code>HASH_NoMember</code>	- No member matching key (...ByKey() functions).

Examples

Example Makefile

```
INCDIR=-I$(HOME)/struct/include
LIBDIR=-L$(HOME)/struct/lib
LIBS=-lhash -llinked_list -larray -lstack -lNew

## LIBS=-lstruct ## optional library which includes all of the libraries

all: my_file

my_file: my_file.o
    $(CC) -o my_file myfile.o $(LIBDIR) $(LIBS)

my_file.o: my_file.c
    $(CC) -O $(INCDIR) -c my_file.c
```

Example Use #1: Container Style Table

```
#include <hash.h>
#include <New.h>      /* used for New()      */
#include <stdio.h>    /* used for printing  */
#include <string.h>   /* used for comparison */

typedef struct blee {
    char *name;
    int  val;
} BLEE;

#define SIZE 10
const struct blee array[SIZE] = {
    {"Joe",    0}, {"Frank",  1}, {"Buddy",  2}, {"Jim",    3}, {"George", 4},
    {"Jane",   5}, {"Jay",    6}, {"Laura",  7}, {"Craig",  8}, {"Bill",   9},
};

unsigned my_hash_fn  (int *val, unsigned size) { return((*val)%size); }
unsigned my_lookup_fn (int *key1, int *key2)   { return(!((*key1) - (*key2))); }
void      my_destructor(BLEE *b)              { Delete(b->name); Delete(b); }
unsigned my_print_fn (unsigned index, BLEE *b, int *key) {
    printf("%u (%d): %s, %d\n", index, *key, b->name, b->val);
}
void      my_handler  (HASH_TABLE *h, enum HASH_ERROR error_num, char *fn) {
    printf("Hash Table 0x%.8x had error %d = %s in %s.\n", h, error_num,
           HASH_errlist[error_num], fn);
}

main () {
    HASH_TABLE *h1, *h2;
    BLEE *b, tmp;
    int i;

    h1 = HASH_Create(7, HASH_KeyOffset, HASH_Offset(&tmp, &tmp.name),
                     HASH_DestroyFunction, my_destructor, NULL);
    h2 = HASH_Create(7, HASH_EmbeddedKey, True, HASH_KeyOffset, HASH_Offset(&tmp, &tmp.val),
                     HASH_HashFunction, my_hash_fn, HASH_LookupFunction, my_lookup_fn,
                     NULL);

    for (i = 0; i < SIZE; i++) {
        b = New(BLEE); b->name = strdup(array[i].name); b->val = array[i].val;
        HASH_Insert(h1, b);
        HASH_Insert(h2, b);
    }

    HASH_Print(h1, NULL);
    HASH_Print(h2, my_print_fn);

    b = HASH_Lookup(h1, "Laura");
    printf("%s = %d\n", b->name, b->val);

    i = 3; b = HASH_Lookup(h2, &i); /* Key must be a pointer */
    printf("%d = %s\n", b->val, b->name);

    HASH_Remove(h2, b);
    HASH_Remove(h1, b); /* This one calls the destructor */

    b = HASH_Lookup(h1, "Frank");
    HASH_Remove(h2, b);
    HASH_RemoveByKey(h1, "Frank"); /* Destructor called */

    HASH_Print(h1, NULL);
    HASH_Print(h2, my_print_fn);
}
```

```
    HASH_SetHandler(my_handler);
    HASH_RemoveByKey(h1, "Frank"); /* Should fail */

    HASH_Destroy(h2);
    HASH_Destroy(h1);
}
```

Example Use #2: Intrusive Style Table

```
#include <hash.h>
#include <New.h>      /* used for New()      */
#include <stdio.h>    /* used for printing */
#include <string.h>  /* used for comparison */

typedef struct blee {
    char *name; int val;
} SIMPLE_BLEE;

typedef struct blee_intr {
    char *name; int val;
    struct blee_intr ptr;
} BLEE;

#define SIZE 10
const struct blee array[SIZE] = {
    {"Joe",    0}, {"Frank",  1}, {"Buddy",  2}, {"Jim",    3}, {"George", 4},
    {"Jane",  5}, {"Jay",    6}, {"Laura",  7}, {"Craig",  8}, {"Bill",   9},
};

/* See example #1, all functions are identical with the exception of the creation
   functions which also include the following:
   HASH_Intrusive, True, HASH_NextOffset, HASH_Offset(&tmp, &tmp.ptr),
*/
```

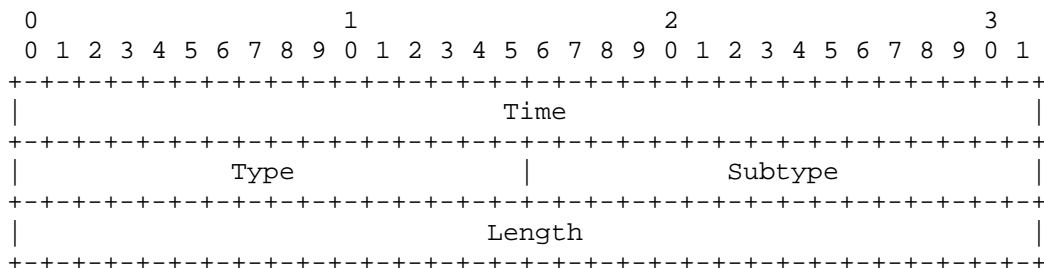
Author

Jonathan L. DeKock

12. Packet Formats

Following are the binary packet formats used by most MRT tools.

1) MRT header (12 bytes)



All MRT messages have this common header, which includes Timestamp, message type and subtype and the length of the message. The length does not include the MRT header itself.

MRT_MSG_TYPES

```

0  MSG_NULL,
1  MSG_START,           /* sender is starting up */
2  MSG_DIE,            /* receiver should shut down */
3  MSG_I_AM_DEAD,     /* sender is shutting down */
4  MSG_PEER_DOWN,     /* sender's peer is down */
5  MSG_PROTOCOL_BGP,  /* msg is a BGP packet */
6  MSG_PROTOCOL_RIP,  /* msg is a RIP packet */
7  MSG_PROTOCOL_IDRP, /* msg is an IDRP packet */
8  MSG_PROTOCOL_RIPNG, /* msg is a RIPNG packet */
9  MSG_PROTOCOL_BGP4PLUS, /* msg is a BGP4+ packet */
10 MSG_PROTOCOL_BGP4PLUS_01, /* msg is a BGP4+ (draft 01) packet */

```

2) BGP (MSG_PROTOCOL_BGP, MSG_PROTOCOL_BGP4PLUS, MSG_PROTOCOL_BGP4PLUS_01)

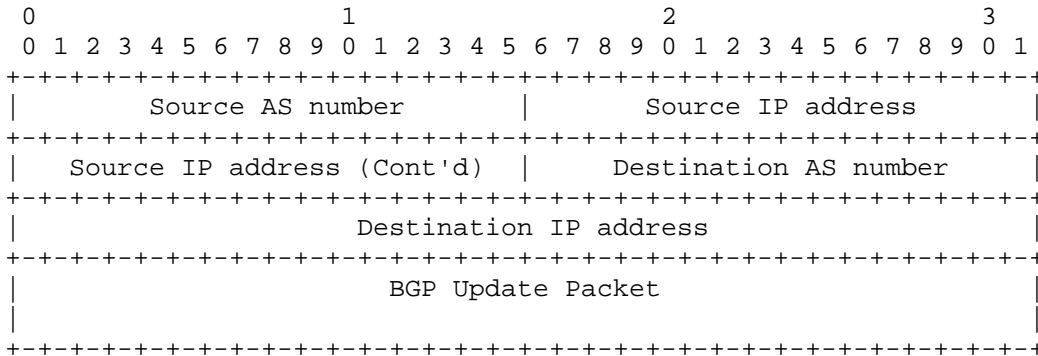
MRT_MSG_BGP_TYPES

```

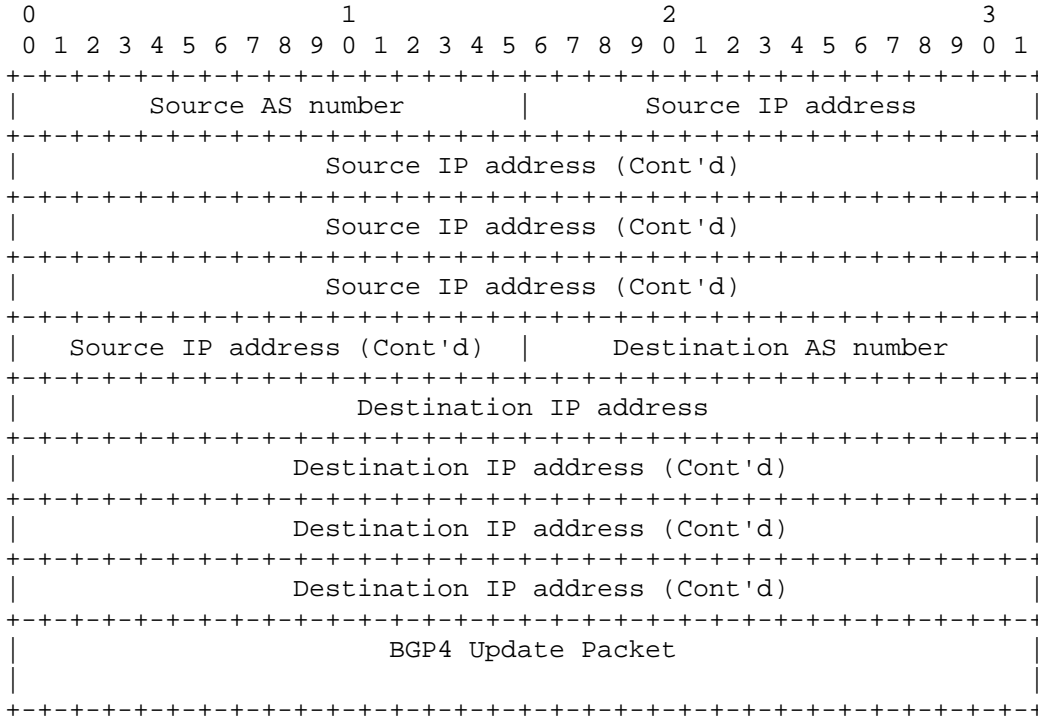
0  MSG_BGP_NULL,
1  MSG_BGP_UPDATE,    /* raw update packet (contains both with and ann) */
2  MSG_BGP_PREF_UPDATE, /* tlv preferences followed by raw update */
3  MSG_BGP_STATE_CHANGE /* state change */
4  MSG_BGP_SYNC

```

If the message type is MSG_PROTOCOL_BGP and the message subtype is MSG_BGP_UPDATE, the following applies:



If the message type is MSG_PROTOCOL_BGP4PLUS and the message subtype is MSG_BGP_UPDATE, the following applies:



If the message subtype is MSG_BGP_STATE_CHANGE, the following format should be used for state change messages:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Source AS number           |           Source IP address           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Source IP address (Cont'd)           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Source IP address (Cont'd)           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Source IP address (Cont'd)           |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Source IP address (Cont'd) |           Old State           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           New State           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

If the message subtype is MSG_BGP_SYNC, the following format applies:

```

+-----+-----+-----+-----+-----+-----+-----+-----+
|           View #           |           File Name [variable]           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

The filename ends with '\0' (null.)

3) The format for our routing table dump is:

```

+-----+-----+-----+-----+-----+-----+-----+-----+
|           View #           |           Prefix           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Prefix (continue)           |           Mask           |           Status           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Time last change           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Attr Len           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           BGP Attribute (Variable length)...           |
+-----+-----+-----+-----+-----+-----+-----+-----+

```