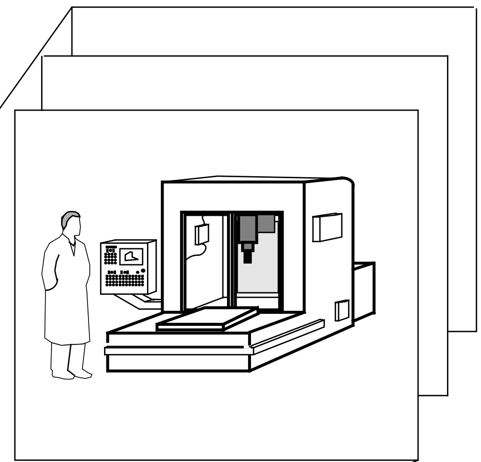
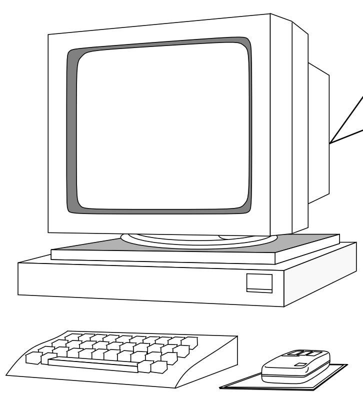
NISTIR 4666
September, 1991

Manufacturing Systems Integration

A Clock for the Manufacturing Systems Integration Testbed





Don Libes

United States Department of Commerce
National Institute of Standards and Technology
Manufacturing Engineering Laboratory
Gaithersburg, MD 20899

NISTIR 4666

September, 1991

Manufacturing Systems Integration

A Clock for the Manufacturing Systems Integration Testbed

Don Libes

United States Department of Commerce

Robert A. Mosbacher,
Secretary of Commerce

National Institute of Standards and Technology

John W. Lyons, Director

A Clock for the Manufacturing Systems Integration Testbed

Don Libes

National Institute of Standards and Technology
Gaithersburg, MD 20899
libes@cme.nist.gov

ABSTRACT

This paper describes a software module that provides timing services to the MSI, a Manufacturing Systems Integration Testbed in the automated factory.

The software "alarm clock" provides services to other MSI software including:

- synchrony,
- real-time, or non-real-time adjusted in a variety of ways,
- alarms at relative or absolute internals.

By providing a central time service, these services are provided more reliably, efficiently, and flexibly than could any client on its own.

This paper describes the implementation, interfaces, and how to design and write programs that use it.

Keywords: manufacturing, MSI, CIM, time, timing, clock, UNIX.

This work is partially supported by the Navy Manufacturing Technology Program and was prepared by U.S. Government employees as part of their official duties and is therefore a work of the U.S. Government and not subject to copyright.

September 5, 1991

Background

The Manufacturing Systems Integration (MSI) project [Senehi1, Senehi2] is developing an integrated production planning and control environment. Its aim is to provide a testbed for production management architectures integrating process planning, production planning and shop floor control.

The current MSI implementation consists of a large number of software processes running simultaneously and communicating with each other. The MSI alarm clock provides time-related services for the MSI. All communication with the clock occurs using the NIST Network Common Memory [Libes].

Introduction

An important function of the MSI is scheduling. For example, resources (e.g. machines) are scheduled in advance (when possible) to optimize throughput. In order for schedules to be useful, their times must be interpreted consistently by other MSI processes. This synchronization is provided by the clock which is the only source of time in the testbed. (We will use “time” to imply “time and date” unless otherwise specified.) The implementation provides this very efficiently, essentially by automatically providing a time stamp with every network input.

The clock keeps “MSI time”. MSI time can be made to track “real time”. More typically, the clock can be reset to a particular time. This is very useful since MSI schedulers currently produce absolute times. Rather than reproducing or editing schedues, the clock can be adjusted backwards to rerun an old schedule. The clock can also scale time. For example, the clock can tick once per second for every real hour (for watching fast operations in detail), or vice versa (for watching macro behavior). The clock can run at virtually any rate, including backwards (although we have not found a use for this last one, yet).

The clock can deliver “alarms”. Analogous to a bedside alarm clock, MSI clients may request alarms which “wake” themselves up at times of interest. For instance, according to a schedule it may be possible to determine that a process has nothing to do for the next 15 minutes. In this case, it is desirable to “sleep” until then, allowing a CPU more efficiency in serving other processes. The implementation provides optimum efficiency by waking a process when either an alarm occurs or input has arrived (the only other reason to wake up). The clock can keep track of an unlimited number of alarms.

The clock provides a variety of other features such as stopping time, single-stepping, and advancing directly to the next alarm. In addition, the clock understands time in several high-level formats.

Status

Prior implementations were inaccurate and inefficient. Processes were forced to busy-wait or estimate future clock movement. There was no alarm mechanism. Processes were also required to do date conversion and comparisons themselves. There was no means of stepping or changing the movement or rate of time.

The current clock (described in this document) keeps time accurately and efficiently. The "advance time to the next alarm" is an especially appreciated feature. The clock was successfully used in the October '90 AMRF test run.

Several enhancements would be useful. They are:

- Publishing extant alarms in common memory. This would be helpful in debugging and gaining a feel for the status of the clock clients.
- An earlier implementation of the clock used non-portable interfaces. The current clock continues using these for compatibility. These should be deprecated.

Starting the clock

The clock is implemented as a UNIX program. Assuming your path is correct, the clock is invoked by the command:

```
alarm_clock
```

The clock has several optional arguments. These can appear in any order:

`-cm <host>`
: The host from which to get common memory service. The default host is the current host.

`mm/dd/yy`
: Numeric values of the month, day, and year to initialize the clock. The default values are the current date. The year is assumed to be biased by 1900, so that only the last two digits should be specified. Numbers do not require leading zeros.

`hh:mm:ss`
: Values of the hours, minutes, and seconds to initialize the clock. The default values are the current time. Numbers do not require leading zeros.

`+debug`
: Enables debugging output. This includes each time a new time/date or alarm occurs, or when a new command is received. By default, debugging is disabled (`-debug`).

`-warn`
: Enables warning output. This includes such things as invalid commands or alarm requests. By default, warning is enabled (`+warn`).

Clock user interface

The clock accepts commands through a common memory interface. Normally, this is done by the **gti** program [Sauder]. This section provides additional information to supplement Sauder's documentation, but does not attempt to duplicate it. Please refer to it if there are further questions.

Clock commands are as follows:

Set Time
: Hours, minutes, and seconds of MSI time are set to those given in the command.

Set Date
Month, day, and year of MSI time are set to those given in the command.

Set Realtime
MSI time is set to EST. Note that the rate is not changed.

Zero Time
Hours, minutes and seconds of MSI time are set to 0.

Set Rate
The rate is set. MSI time moves *rate* times faster than real time. The default rate is 1. (See Step Time.)

Stop Time
Time is stopped until the next Start Time command.

Start Time
Time is started if not already running.

Step Time
The time is advanced to that of the next alarm. If no alarms exist, time is incremented by *rate* seconds.

Kill Clock
The clock process exits. It will accept no commands after this.

Status Report
No action (other than default).

Note that the **gti** program does not give entirely clean access to the clock. For example, while changing the time, **gti** also stops the clock. Inexplicibly, this does not occur while changing the date.

The commands are available to the C programmer as the enumerated type, **clock_command**, which is defined in the file, **msi_clock_command.h**. This file also defines two common memory variables via the preprocessor macros: `MSI_CLOCK_CMD` and `MSI_CLOCK_STS`.

Commands are written to the variable defined by `MSI_CLOCK_CMD` in a non-portable format described by the C structure:

```
struct clock_command {
     int number;
     enum type type;          /* actual command to execute */
     union {
          struct tm time;
          double rate;
     } data;
};
```

In response to each command, the clock writes a status reported to the variable defined by `MSI_CLOCK_STS` in a non-portable format described by the C structure:

```
struct clock_status {
```

```
        int number_echo;
        enum type type_echo;
        double rate;
        enum status status;
};
```

The command element **number** is transferred to status element **number_echo**. Similarly, so is **type**. **type** is also interpreted as a clock command as described above.

Depending on the type, appropriate elements are extracted from the union **data**. The current rate is always echoed in the clock status.

Finally, the **status** element is set to one of *running*, *stopped*, or *killed* as appropriate. The actual values of these and other enums may be found in the source code.

The clock status interface allows no way of returning command errors. Errors are printed on stdout (if `+warn` is in effect).

Clock program interface

The clock provides two services to MSI client programs. Namely, time service and alarm service. In order to reduce application programming as much as possible, a client-side C library exists which encapsulates the implementation of MSI clock services, leaving actual usage as the simplest of programming tasks. Programmers that cannot use the client-side C library (i.e., Symbolics users) should refer to "Implementation Notes" (below).

An example of most of the services described below exists in **~msi/src/alarm_clock/test_service**.

In order to use the clock service library, the following steps are required:

1) You must include a header file defining the library interface. In particular, you should add the following line to the beginning of any C source file that has calls to the clock services.

```
#include "msi_clock_service.h"
```

2) You must call `msi_clock_init` before calling any other clock services. `msi_clock_init` returns a 0 normally, or a -1 upon failure. For example:

```
if (-1 == msi_clock_init()) exit(-1);
```

Since the clock uses common memory, you should also call `cm_init`, if you do not do so elsewhere. `cm_init` should be called before `msi_clock_init`.

3) Any files that make clock service (or common memory, for that matter) calls should be compiled with additional `-I` flags to tell the C compiler where it can find the include files. For example:

```
cc -I/home/lurch/msi/src/alarm_clock -I/usr/local/include/cm
-c prog.c
```

4) Object files must be linked with the appropriate libraries. For example:

```
cc prog.o /home/lurch/msi/src/alarm_clock/msi_clock_servi-
ce.o -lcm -lstream
```

5) The common memory manager and the alarm clock must be running before you run your program(s). The common memory manager must be run first. For example:

```
cmm &
alarm_clock &
```

Time

When the clock is running (or if it is being stepped or set by hand), it writes the latest MSI time to the common memory. (The actual rate at which time is written is once per MSI second up to `MAX_RATE` times per real second. `MAX_RATE` is a compile-time constant currently equal to 3.)

Each time your local common memory is updated by other common memory variables (or by explicit request), you receive the latest copy of the MSI time. (Time is not sent on its own, because it is written too frequently for most processes to keep up.)

Time is transmitted as a 4-byte integer (most significant byte first) representing seconds since January 1, 1970. It is stored locally as a **time_t**. **time_t** can be directly manipulated by the usual C arithmetic operators, and printed out with **printf** using the **%d** format.

UNIX provides a number of library routines for high-level manipulation of a **time_t**, including **ctime**(3) which converts it to an ASCII representation, and **localtime**(3) which breaks it into a form in which individual elements (hours, minutes, etc.) are directly manipulable.

This representation of time may be retrieved with the following function.

```
time_t msi_time_seconds(char *time)
```

The function **msi_time_seconds** returns the number of seconds equivalent to the time described by its argument. Its argument may be 0, in which case, the current MSI time is returned. Alternatively, the "standard MSI ASCII time representation" may be used.

The *standard MSI ASCII time representation* is of the form "*yyyymmddhhmmss*". All fields must be zero-filled. For example, "`19901231190123`" represents December 31, 19:01:23 1990. This form may be directly produced by the following function:

```
char *msi_time_printable(time_t time)
```

Given a number of seconds since January 1, 1970, the function **msi_time_printable** returns the standard MSI ASCII time representation. If **time** is 0, the current MSI time is used. This function returns a pointer to a static buffer which will be reused when this function is next called.

The following C program prints the current MSI time in both formats. Error checking has been omitted for readability.

```
main()
{
     cm_init("time_reader",(char *)0,0);
     msi_clock_init();

     cm_sync(CM_WAIT_READ);
     printf("MSI time is %s or %d seconds since 1/1/70\n",
          msi_time_printable((time_t)0),
          msi_time_seconds((char *)0));
}
```

Note that `CM_WAIT_READ` (meaning "*force the common memory server to synchronously update all my variables*") is an expensive **cm_sync** option which is invariably not required by real programs, but is necessary in this example.

As mentioned earlier, C arithmetic operators can work directly on **time_t** values. This is not the case for MSI printable values. The following macros are provided as a convenience for comparing two MSI printable values. They are actually more efficient than converting both values via **msi_time_seconds**.

`int msi_time_eq(x,y)` returns 1 if x == y, 0 otherwise.

`int msi_time_lt(x,y)` returns 1 if x < y, 0 otherwise.

`int msi_time_gt(x,y)` returns 1 if x > y, 0 otherwise.

`int msi_time_ge(x,y)` returns 1 if x >=y, 0 otherwise.

`int msi_time_le(x,y)` returns 1 if x <= y, 0 otherwise.

`int msi_time_ne(x,y)` returns 1 if x != y, 0 otherwise.

Alarm

The clock will write to a named common memory variable at a requested time, in effect, providing an alarm service. Alarms are manipulated by calling the following functions:

```
void msi_alarm_create_absolute(name, time)
void msi_alarm_create_relative(name, time, count)
void msi_alarm_delete_absolute(name, time)
void msi_alarm_delete_relative(name, time)
void msi_alarm_delete(name)
char *name, *time;
int count;
```

In all of these calls, name indicates the common memory variable to be written at the given time. This variable must be declared by the user as follows:

```
cm_declare("var",CM_ROLE_READER|CM_ROLE_WAKEUP);
```

The time may be specified either using the standard MSI ASCII form (described) above, or using one or both of the strings "mm/dd/yy" and "hh:mm:ss" (separated by whitespace) in either order. The latter style does not require leading zeros.

The formats will be repeated and discussed individually here.

```
void msi_alarm_create_absolute(name, time)
```

msi_alarm_create_absolute creates an alarm for an absolute time/date. After occurring, the alarm is deleted. (This is important, since time can be set backwards.)

```
void msi_alarm_create_relative(name, time, count)
```

msi_alarm_create_relative creates an alarm for a relative time. The alarm will occur *count* times in the future, each spaced the indicated date/time from one another. A zero count indicates infinity. If the clock is advanced, alarms scheduled for a prior time will immediately occur, respacing from that point. If the clock is set back, relative alarms will not begin triggering until the previous (now future) time is reached.

Note that months and days are essentially meaningless in relative date specifications and are therefore ignored. However, for readability, they must be specified (as zeros).

For example, the following program sets an alarm for 10 seconds in future, and then wakes at that time.

```
#include "cm.h"
#include "msi_clock_service.h"

cm_value val = {0,0,0,1};

main()
{
     cm_variable *foo;

     cm_init("sleeper",(char *)0,0);
     msi_clock_init();
     msi_alarm_create_relative("foo","00:00:10");
     foo = cm_declare("foo",CM_ROLE_READER|CM_ROLE_WAKEUP);
     cm_sync(CM_WAIT_AT_MOST_ONCE);
     if (cm_get_new_value(foo,&val))
          printf("I feel very refreshed!\n");
}
```

```
void msi_alarm_delete_absolute(name, time)
```

msi_alarm_delete_absolute removes an absolute alarm set for the given time. If no time is specified, all absolute alarms which match the name are deleted.

```
void msi_alarm_delete_relative(name, time)
```

msi_alarm_delete_relative removes a relative alarm set for the given time. If no time is specified, all absolute alarms which match the name are deleted.

```
void msi_alarm_delete(name)
```

msi_alarm_delete removes all alarms associated with the given name.

The clock will remember an unlimited number of alarms. Multiple alarms may be associated with the same variable.

Alarm variables should be named in a way that distinguishes them from one another and from other variables. For example, append the process name with the string "`_alarm`", such as in "`hws_alarm`". Of course, a process may have multiple alarm variables if desired.

The actual implementation of these functions has ramifications on their use. In particular, **cm_sync** must be called after each of these functions in order to get them to work. More importantly, since they use the same common memory variable internally to communicate with the alarm clock, if you do not call **cm_sync** between alarm clock calls, the former of the two alarm requests will be overwritten by the latter request. The implementation avoids calling **cm_sync** (for you) because doing so can prevent use of common memory's reliable queuing mechanism.

Since common memory clients have no way of knowing if anyone is reading a common memory variable, the clock has no way of telling if a client has died. Thus, the clock will deliver an alarm that a client has requested, even if the client has long since exited. If there is a chance that client software may be restarted with alarms pending, I recommend you initialize your system by deleting any alarms for which you are responsible. This will prevent alarms from a previous incarnation triggering mysteriously (to you).

Implementation notes

The following implementation notes are probably not of interest to the casual user of the MSI clock. They describe precise details of the common memory interface.

Common memory time interface

The time interface is implemented by reading from the variable specified by the macro `MSI_TIME`. This should be declared `CM_ROLE_READER`. It is almost certainly wrong to declare it as `CM_ROLE_WAKEUP`, as the clock can easily overrun your input buffers this way.

As described above, time is transmitted as a **time_t** value most significant byte first. After your local common memory has been updated, you can read it by calling **cm_get_value** and then converting it to native form (e.g., **ntohl** on a Sun). The following program prints the current date.

```
cm_value time_val = {0,0,0,1};
cm_variable *time_var;

main()
{
     cm_init("reader",(char *)0,0);

     time_var = cm_declare(MSI_TIME,CM_ROLE_READER);
     cm_sync(CM_WAIT_READ);
     cm_get_value(time_var,&time_val);
     printf("seconds of MSI time since 1/1/70 is %d\n",
          ntohl(time_val.data));
}
```

Common memory alarm interface

The alarm interface is implemented by writing requests to the variable specified by the macro `MSI_ALARM_REQUEST`. This is a shared variable and should be declared `CM_ROLE_NONXWRITE`.

Requests are ASCII strings in the following format:

```
+ name a         {time/date}
+ name r count   {time/date}
- name [a|r     [{time/date}]]
```

The initial + or − indicates whether an alarm is created or deleted. The name indicates the common memory variable to be written at the given time/date. The a or r describe whether the time/date are to be used absolutely or relatively as described earlier. The following example, sets an alarm for 10 seconds in the future, and then wakes up at that time.

```
char *string = "+ a r 1 00:00:10";
cm_value val = {string,sizeof(string),sizeof(string),0};

main()
{
     cm_variable *req, *foo;

     cm_init("sleeper",(char *)0,0);
     req = cm_declare(ALARM_REQ,CM_ROLE_NONXWRITER);
     foo = cm_declare("wake",CM_ROLE_READER|CM_ROLE_WAKEUP);
     cm_set_value(req,&val);
     cm_sync(CM_WAIT_AT_MOST_ONCE);
     if (cm_get_new_value(foo,&val))
          printf("I feel refreshed!\n");
```

```
}
```

Status reporting

The clock interfaces provide no way to deliver error messages from the clock to a client or user-interface. This should be remedied. In the meantime, warnings are printed to the clock's **stdout**, and errors to the clock's **stderr**.

Warnings and errors are self-explanatory.

Testing

The program sleeper (in **~msi/src/alarm_clock**) is a test program that will report any alarm activity on the common memory variables **a**, **b**, **c**, **d**, **e**, and **f** along with the MSI time at each alarm. This is particularly useful for testing the alarm.

A separate program, such as **realuser**, must be used to write alarm requests.

Portability

Due to the original design of the command/status interfaces, the clock itself will not be able to communicate with user interface programs if they or the clock itself runs on a different hardware/software base than is currently being used, a MC680x0-based computer running SunOS.

The command/status formats used by the user interfaces currently use native integer and floating point storage formats, which differ from computer to computer. This should be corrected.

The interfaces used for time and alarm request and delivery are portable to any POSIX-conforming system.

Source Code

The source code for the clock can be found in **~msi/src/alarm_clock**. This code is extremely subtle and should not be modified without careful thought. The code is very carefully written to generate accurate time and alarms, despite servicing alarms and other requests.

The basic loop is shown below.

```
while (1) {
     /* loop here, if necessary, for example, to flush */
     /* status from stop or kill */
     while (need_to_check_cm) {
          need_to_check_cm = FALSE;
          check_cm();
     }

     if (next_timeout(&timeout) != in_the_future) {
          wakeups();
          continue;
     }
```

```
        FD_SET(cm_server_socket,&readfds);

        rc = select(maxfds,&readfds,(fd_set *)0,(fd_set*)0,
                            timeout);

        /* only significant passage of time occurs here */
        update_time();

        switch (rc) {
        case -1:
             /* hope it was a recoverable interrupt */
             /* and try again */
             perror("select");
             break;
        case 0:
             /* time expired - do wakeups */
             wakeups();
             break;
        case 1:
             need_to_check_cm = TRUE;
             break;
        default:
             wprintf("select() = %d?\n",rc);
             break;
        }
   }
```

The first important function is **next_timeout** which decides what passage of time to wait for. This function examines pending alarms, as well as the current clock rate to decide if the clock, itself, can sleep and if so for how long. Even if the clock isn't running, an alarm might still have to be generated, if, for example, the time was advanced by an explicit command. If any alarms are ready to be delivered, **wakeups** delivers them on the following statement.

Note that this is immediately followed by a C continue statement which drives control to the top of the loop. At the top is a check for common memory. If any alarms need to be delivered, common memory will need to be flushed. (**need_to_check_cm** is set when any common memory variable has been written.) On the other hand, we do not want to sync with common memory if unnecessary since this is a (relatively) slow operation.

Other fast operations such as requests to stop or set the clock will be picked up and executed when common memory is checked at this point. Rather than putting logic inside of **check_cm** to loop internally, it is much simpler to recall it immediately after it has completed. This may seem strange, but it is much cleaner and simpler than writing it any other way.

Once any alarms are flushed, the alarm clock sleeps in such a way that it will be woken up only if the (real) time it wanted to awaken has arrived, or a service requested has arrived via common memory. Because of our avoidance of unnecessary common memory synchronization, we can consider that the only lengthy passages of time occur while the clock is asleep – i.e., during **select**. Thus, this is immediately following by a call to **update_time** which updates the clock's idea of the time.

update_time also publishes the new time in common memory. (Actually, it only writes a common memory variable and preps **need_to_check_cm**, so that it will be written out to common memory when the top of the loop is reached, which will be immediately after this.)

update_time (below) looks trivial but is very subtle. **msi_time** is (or shortly will be) the current msi_time. **old_msi_time** is *not* the previous value of **msi_time**, but *is* the time of the last MSI time that was last specficially requested by command. **old_tv_real_time** is the corresponding real time when **old_msi_time** was set. Similarly, **tv_real_time** is the current real time. By subtracting these, scaling by the rate, and adding the result to the **old_msi_time**, the new MSI time is determined.

The key is not to use the MSI time computed during the previous iteration. The problem is that miniscule fractions of time creep in between when the time is sampled and the new value is computed. By recomputing the new time from a known-to-be-correct baseline each time, we avoid any time slop that might build up.

```
update_time()
{
     if (status.status != running) return;

     gettimeofday(&tv_real_time,(struct timezone *)0);

     msi_time = old_msi_time +
               rate *
                   (tv_to_dtime(&tv_real_time)
                 - tv_to_dtime(&old_tv_real_time));

     msi_time_to_cm();
}
```

Following **update_time**, we determine why the clock stopped sleeping. This can be one of four cases.

-1 or **default** means an error occurred. We have never actually seen this happen.

0 means no service request occurred, and thus the clock slept for the time it intended to. Since this could very well been for an alarm, **wakeups** is called to find out. **wakeups** will actually find all alarms (several may be set for the same time) that have the current or a previous time.

1 means that a service request occurred via common memory. Control is passed back to the top of the loop so that common memory can be synchronized and the request received.

UNIX Time

The following notes were made by the author while working on the code. They should only be read should anyone have reason to want to understand the source code. Unfortunately, it was impossible to render these notes in the source code itself (due to the accompany diagram), which is where they really belong.

Type	Contents/Representation	Defined by	Bytes
`time_t`	seconds	`<sys/stdtypes.h>`	4
`struct timeval`	seconds, microseconds	`<sys/time.h>`	8
`struct tm`	seconds, minutes, hours, days, etc.	`<time.h>`	44
`struct timeb`	seconds, milliseconds, timezone	`<sys/timeb.h>`	10
`char *`	ASCII	various funcs	?
`double`	seconds in floating point	locally	8

As one look at the above table shows, UNIX allows a large number of time formats.

The formats are used as follows:

time_t is used to deliver time to the user via common memory.

struct timeval is the basic time generated by the UNIX kernel. All other times are generated from it. It is also required for sleeping via the **select** system call.

struct tm is a convenience primarily for translating out of ASCII representations of time/date.

timeb is a form that would be ignored except that it is generated by **getdate**, and ASCII date/time scanner which is more flexible than any other mechanism (and is therefore quite tempting). Currently, the clock does not use **getdate** and therefore, **timeb**. However, it is described here for completeness should we decide to use it in the future.

Using *double* is a common technique for storing subsecond times with limited precision. Here, it is an intermediary useful for accurately scaling MSI time by the passage of real time. Specifically, while MSI time is only kept to second accuracy, rates greater than 1 require an MSI second to correlate to a sub-real-second. To perform arithmetic on real time in fractional units, it is most convenient to manipulate it as an atomic entity with floating point precision provided by **double**.

The diagram at the rear of this paper shows the function calls or casts necessary to move from one representation to another.

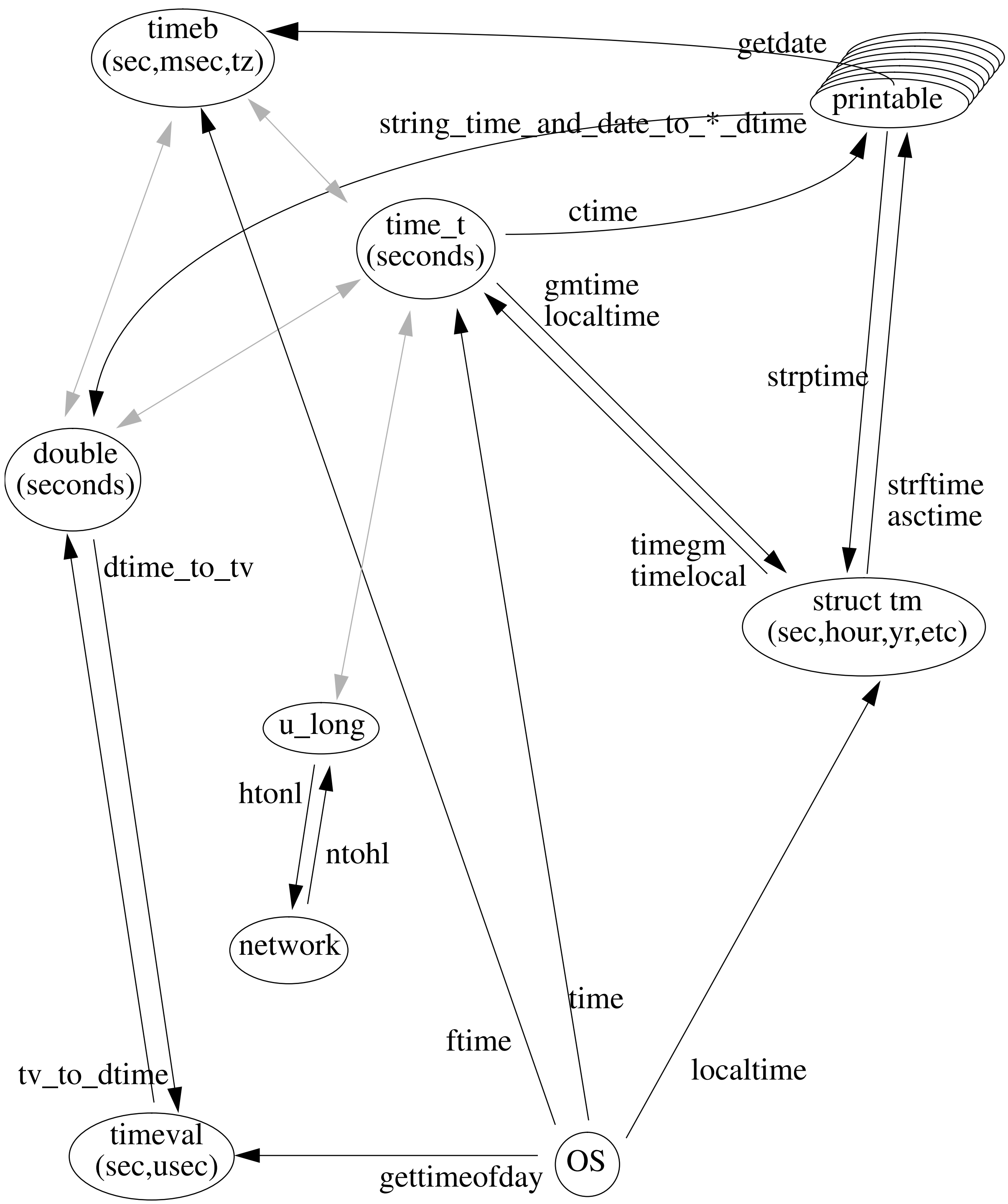
References

[Libes] Don Libes, "*NIST Network Common Memory User Manual*", NISTIR 90-4233, PB90-183260/AS, January 1990.

[Sauder] Dave Sauder, unpublished documentation describing **gti** program.

[Senehi1] M. Kathleen Senehi et. al., "*Manufacturing Systems Integration Initial Architecture Document*", National Institute of Standards and Technology, Interagency Report, forthcoming.

[Senehi2] M. Kathleen Senehi et. al., "*Manufacturing Systems Integration Control Entity Interface Document*" National Institute of Standards and Technology, Interagency Report 91-4626, June, 1991.



Bubbles indicate date/time formats. Lines indicate transformations by labelled function calls. Shaded lines indicate transformations by casts. Some functions perform hidden intermediate translations. For example, **ctime** converts **time_t** to **struct tm** before converting to ASCII. There are a myriad of printable formats. Unfortunately, none of the functions support all of the formats. For simplicity, I have omitted this quagmire from the diagram.