Implementation of the MPL Compiler*†

Jan M. Rizzuto and James da Silva

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

February 14, 1995

Abstract

The Maruti Real-Time Operating System was developed for applications that must meet hard real-time constraints. In order to schedule real-time applications, the timing and resource requirements for the application must be determined. The development environment provided for Maruti applications consists of several stages that use various tools to assist the programmer in creating an application. By analyzing the source code provided by the programmer, these tools can extract and analyze the needed timing and resource requirements. The initial stage in development is the compilation of the source code for an application written in the Maruti Programming Language (MPL). MPL is based on the C programming language. The MPL Compiler was developed to provide support for requirement specification. This report introduces MPL and describes the implementation of the MPL Compiler.

*This work is supported in part by ONR and DARPA under contract N00014-91-C-0195 to Honeywell and Computer Science Department at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, ONR, the U.S. Government or Honeywell.
Computer facilities were provided in part by NSF grant CCR-8811954.

†This work is supported in part by ARPA and Philips Labs under contract DASG60-92-0055 to Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, PL, or the U.S. Government.

1 Introduction

A *real-time* system requires that an application meet the timing constraints specified for it. For *hard real-time*, a failure to meet the specified timing constraints may result in a fatal error [2]. Timing constraints are not as critical for *soft real-time.* The Maruti Operating System was developed to meet the real-time constraints required by many applications. In order to schedule and run an application under Maruti, the timing and resource requirements for that application must be determined. The development environment for Maruti consists of several tools that can be used to extract and analyze these requirements [2].

The *Maruti Programming Language* (MPL) is a language developed to assist users in creating applications that can be run under Maruti. MPL is based on the C programming language, and assumes the programmer is familiar with C. MPL provides some additional constructs that are not part of standard C to allow for resource and timing specification [1]. In addition, when an MPL file is compiled, some of the resource requirements can be recognized and recorded to an output file. This output file is used as input to the integration stage, which is the next stage in the development cycle. During integration, additional timing requirements may be specified.

Previously, an MPL file was compiled by first running the source code through the Maruti pre-compiler, which created a C file that was then compiled using a C compiler [1]. The pre-compiler extracted the necessary information, and converted the MPL constructs that were not valid C statements into C code. This required the additional pass of the pre-compiler over the source code. We have created a compiler for MPL that integrates both the actions of the pre-compiler and the compiler into one stage. In this report, we present MPL, and a description of the compiler we implemented. Section 2 defines the abstractions used in Maruti. In Section 3, the syntax of the constructs unique to MPL is defined. The details of the implementation of the compiler are given in Section 4. Section 5 describes the resource information that is recorded during compilation. Conclusions appear in Section 6, followed by an Appendix containing a sample MPL file, and the resource information recorded for that file.

2 Maruti Abstractions

An MPL application is broken up into units of computation called *elemental units* (EUs). Execution within an EU is sequential, and resource and timing requirements are specified for each EU. A *thread* is a sequential unit of execution that may consist of multiple EUs. MPL allows threads of execution to be specified by the programmer through several of the constructs provided. A *task* consists of a single address space, and threads that execute in that address space. *Modules* contain the source code of the application as defined by the programmer. An application may consist of several modules. During execution, modules are mapped to one or more tasks.

3 MPL Constructs

There are several constructs defined in MPL that are not a part of standard C. These constructs have been implemented in the MPL compiler.

3.1 Module Name Specification

A *module* may consist of one or more source files written in MPL. At the start of each MPL file, the name of the module that the source file corresponds to must be indicated. This is given by the following syntax:

```
module-name-spec ::= 'module' <module-name> ';'.
```

The `module-name` may be any valid identifier that is accepted by standard C. The module name specification must appear at the beginning of the source file, before any other MPL code. The specification is not compiled into any executable code. It is simply used to indicate the module that the functions within the file belong to.

3.2 Shared Buffers

A *shared buffer* can be used to declare memory that may be shared by several tasks, to permit communication between the tasks. A declaration of a shared buffer requires the type be defined as with a variable declaration. The syntax of a shared declaration is:

```
shared-buffer-decl ::= 'shared' <type-specifier> <shared-buffer-name>.
```

The `shared-buffer-name` can be any valid identifier, and the `type-specifier` can be any valid type for a variable. A shared declaration is compiled as a pointer to the type given in the declaration of the shared buffer, rather than the type given.

3.3 Region Constructs

The are two constructs used to allow for mutual exclusion within an application.

3.3.1 Region Statement

The *region* statement is used to enforce mutual exclusion globally throughout an entire application, and is given by the syntax:

```
region-statement ::= 'region' <region-name>
                     { mpl-statements }.
```

The `mpl-statements` may be any number of valid MPL statements. These statements make up a critical section.

3.3.2 Local Region Statement

The *local_region* statement is used to enforce mutual exclusion within a task, and follows the same syntax of the region statement:

```
local-region-statement ::= 'local_region' <local-region-name>
                           { mpl-statements }.
```

3.4 Channel Declarations

Channels are used to allow for message passing within a Maruti application. Each channel declared has a type associated with it given by a valid C type-specifier. This type indicates the type of data that the channel will carry.

Channels may be declared in both *entry* and *service functions,* which will be defined below. The syntax for channel declarations is:

```
channel-declaration-list-opt ::= { channel-declaration-list }.
channel-declaration-list ::= channel-declaration { channel-declaration }.
channel-declaration ::= channel-type channels ';'.
channel-type ::= 'out' | 'in' | 'in-first' | 'in-last'.
channels ::= channel { ',' channel }.
channel ::= <channel-name> ':' type-specifier.
```

3.5 Entry Functions

An *entry function* is a special type of function that may be defined in an MPL source file. Each entry function corresponds to a thread within the application. The syntax for an entry function definition is:

```
entry-function ::= 'entry' <entry-name> '(' ')' entry-function-body.
entry-function-body ::= channel-declaration-list-opt mpl-function-body.
```

3.6 Service Functions

Service functions are another type of special function supported by MPL. A service function is invoked when a message is received from a client. Each service function definition requires an *in* channel and message buffer be included in the definition. The service function will be executed when there is a message on the channel given in the definition. The definition of a service function is similar to that of an entry function:

```
service-function ::= 'service' <service-name>
                '(' <in-channel-name> ':' type_specifier ',' <msg-ptr-name> ')'
                service-function-body.
service-function-body ::= channel-declaration-list-opt mpl-function-body.
```

3.7 Communication Function Calls

There are several library functions used to allow for message passing within a Maruti application.

3.7.1 Send Calls

Each call to the *send* function must specify an outgoing channel for the message:

```
void send ( channel channel_name, void *message_ptr );
```

3.7.2 Receive and Optreceive Calls

Both *receive* calls, and *optreceive* calls must be associated with an incoming channel (*in, in_first,* or *in_last*):

```
void receive ( channel channel_name, void *message_ptr );
int optreceive ( channel channel_name, void *message_ptr );
```

A call to receive requires that there be a message on the incoming channel. Optreceive should be used when a message may or may not be on the channel. Optreceive checks for the message, and returns a value indicating if a message was found.

3.8 Initialization Function

Each task has an *initialization routine* that is executed when the application is loaded. This function is specified by the user with the following name and arguments:

```
int maruti_main (int argc, char **argv)
```

4 Implementation

We started with version 2.5.8 of the Gnu C compiler. By modifying the source code for the C compiler, we have created a compiler for applications written in MPL. In addition to what the standard Gnu C compiler does, this modified compiler handles the additional constructs defined in MPL, and records information about the source code that is needed by Maruti. A source code file written in MPL is specified with an *mpl* extension.

4.1 Modifications to GCC File Structure

In the process of modifying the compiler, some existing files were modified. In addition, some new files were also created. The source code for version 2.5.8 of GCC allows compilers to be created for several different languages: C, C++, and Objective C. The GCC compiler uses different executable files for the different languages that it compiles. There are separate files for C, C++, and Objective C *(cc1, cc1plus, cc1obj).* The GCC driver, *gcc.c,* uses the extension of the source file specified to determine the appropriate executable (and therefore language) to compile the source file. The driver then executes the compiler, passing on the appropriate switches. The driver was modified to accept input files with an *mpl* extension. *Cc1mpl* is the new executable that was created to compile MPL source files. When a file with an *mpl* extension is specified as a source file to be compiled, this new executable file is used. When an MPL file is compiled, it automatically passes on the switch *-Maruti_output,* which indicates that the needed output should be recorded to a file with an *eu* extension.

The executable files for each language are composed of many object files. Some of these files are common to all the languages, and some of the files are language-specific. The language-specific files added for compiling MPL files are those files with an *mpl-* prefix.

Gperf is a tool used to generate a perfect hash function for a set of words. Gperf is used to create a hash function for the reserved words for each language. The files containing the input to gperf are indicated by a file name with a *gperf* extension. There are several different **.gperf* files containing the reserved words for the different languages recognized by the compiler. The *mpl-parse.gperf* file contains all the reserved words for C, in addition to those added for MPL. For each language, the output from running gperf is then incorporated into the **-lex.c* file. This output includes a function *is_reserved_word()* that is used to check if a token is a reserved word. The file *mpl-lex.c* is basically the *c-lex.c* file, with the output of running gperf on *mpl-parse.gperf* instead of *c-parse.gperf.*

The file *maruti.c* contains the routines that have been written to implement MPL. This file is linked in with the executable for all of the languages, to prevent undefined symbol errors from occuring. Calls to the routines contained in this file occur in both the language-specific, and the common files. The flag *maruti_dump* is set in *main()* to indicate whether information about the source code should be recorded to the appropriate output file. This flag prevents calls to the routines in *maruti.c* which are made in the common files from occuring for the languages other than MPL. The files containing these calls are:

- *calls.c*
- *explow.c*
- *expr.c*
- *function.c*
- *toplev.c*

There are several reasons why the new language-specific files have to be created for MPL. The files *mpl-lex.h* and *mpl-lex.c* needed to be created for MPL because MPL contains several additional reserved words not present in C, as mentioned earlier. The file *c-common.c* relies on information in the header file *c-lex.h.* Since MPL uses *mpl-lex.h, mpl-common.c* includes *mpl-lex.h,* instead of *c-lex.h. Bison* is a tool that allows a programmer to define a grammar through rules, and converts them into a C program that will parse an input file. The **-parse.y* files are the bison files used to create the grammar to parse a source file. Since the grammar for MPL needed to be modified to accept the additional constructs, the *mpl-parse.y* file was created. There is one function used in compiling MPL source files that is defined in *mpl-parse.y,* instead of *maruti.c.* This function needed to access the static variables declared in *mpl-parse.y,* and in order to do so, the function definition was placed in that file. Finally, the file *mpl-decl.c* was created, because of its dependence on *mpl-lex.h,* and also to allow for an additional type specification used in MPL.

4.2 Compiling MPL Constructs

MPL extends the C language to allow for various constructs. In order to implement these extensions, the grammar used to recognize C in GCC had to be extended. The following are recognized as reserved words for MPL, in addition to the standard reserved words for C: *shared, region, local_region, module, in, out, in_first, in_last, entry, service, send, receive,* and *optreceive.* The keywords *in* and *out* were reserved words in the *c-** files, because they are used by Objective C, but in MPL they are used as channel types. In addition to the new reserved words, rules were added and modified resulting in the rules in *mpl-parse.y.*

4.2.1 Module Name Specification

A rule was added to the grammar to parse the module name specification in an MPL file. The rule for a whole program was also modified to include this module statement. This rule expects the module statement to appear before any other definitions. Since the module name specification does not result in any executable code, the only action taken is to record the module name given by the programmer.

4.2.2 Shared Buffers

There are no rules added to the grammar for a shared buffer declaration. When a variable declaration is parsed, a tree is created that keeps track of all the specification information given for that declaration. For example, *typedef* and *extern* are two of the possible type specifications. The token *shared* is recognized as a type specification, just as *typedef* and *extern* are recognized. When a declaration is made, these specifications are processed in the function *grokdeclarator()* in *mpl-decl.c.* When a shared specification is encountered, the declaration is converted to a pointer to the type specified, instead of just the type specified. Other than this conversion to a pointer, the declaration is compiled just as any other declaration would be compiled in C.

4.2.3 Region Constructs

The region constructs are considered statements in MPL. Several rules were added to parse these constructs, and the *region* and *local_region* statements were added as options for a valid statement in the grammar for MPL.

Both region and local_region statements are compiled in the same manner. Each region has a name, and a body which is the code within the critical section. In order to protect these critical sections, calls are made to the Maruti library function *maruti_eu().* When a region is parsed, the compiler generates two calls to *maruti_eu(),* in addition to the code in the body of the region. The first call is generated just before the body, and the second call just after. These calls are generated through functions in *maruti.c.* The functions are based on the actions that would have been taken, had the parser actually parsed the calls to *maruti_eu()* in the source file.

4.2.4 Channel Declarations

The rules added for a channel declaration allow any number of channels to be declared in either an entry or a service function. Each channel declaration requires several pieces of information:

- *Channel-type*
- *Channel-name*
- *Type specifier indicating the type of data that channel carries*

A linked list of declared channels is maintained. For each declared channel the following information is saved:

- *Channel-name*
- *Type information*
  1. *Size in bytes*
  2. *String encoding the type of the data*
- *Channel-id*

The *channel-id* is a unique identification number assigned to each declared channel. Channel declarations do not add to the compiled code. The channels are not allocated memory. The information describing each channel is simple stored in the linked list. During compilation, whenever a channel is referenced, the appropriate information is obtained from this list.

4.2.5 Entry Functions

Entry function definitions are compiled differently than other function definitions. An entry function would appear in an MPL file in the following form:

```
entry <entry_name> ()
<channel_declaration_list_opt>
{
  <mpl_function_body>
}
```

Where `entry_name` is an identifier that is the name of the entry function, the `channel_declaration_list_opt` contains any channels the user wants to define for that function, and `mpl_function_body` is any function body that would be accepted as a definition in a standard MPL function. Semantically the entry function is equivalent to the following MPL code:

```
_maruti_entry_name ()
{
  while(1)
  {
    maruti_eu();
    entry_name () ;
  }
}

entry_name ()
{
  mpl_function_body
}
```

An entry function is compiled into two functions, as if the two functions given above had been part of the source file. Essentially, the first function is just a stub function that calls *maruti_eu(),* then calls the second function compiled. As with generating function calls, the routines to generate the code for entry function definitions are based on the actions that would have been taken had the parser actually parsed the code for the two separate functions.

4.2.6 Service Functions

Service functions definitions are handled very much like entry function definitions. The syntax of a service function differs slightly from that of an entry function, since it requires that an incoming channel and a message buffer be defined:

```
service <service_name> (<in_channel_name> : <type_specifier>, <msg_ptr_name>)
<channel_declaration_list_opt>
{
   <mpl_function_body>
}
```

Like the entry functions, service functions are semantically equivalent to two functions, where one is simply a stub function calling the second function that is generated:

```
_maruti_service_name ()
{
type_specifier _maruti_msg_ptr_name ;

  while(1)
  {
    if ( optreceive ( _maruti_in , id  , & _maruti_msg_ptr_name, size ) )
    {
       service_name (& _maruti_msg_ptr_name );
    }
  }
}

service_name (msg_ptr_name)
type_specifier  *msg_ptr_name;
{
  mpl_function_body
}
```

The `service_name`, `channel_declaration_list`, and `mpl_function_body` are all the same as described previously for entry functions. In addition, service functions have two other items specified in their definitions. The first is a channel. Every service function requires a channel be specified. This channel is always declared as an *in* channel with the name `in_channel_name`. The type is given by `type_specifier` as if it had been declared in the `channel_declaration_list`. The channel is used to invoke the service function. This in channel is used by the optreceive in the stub function that calls the function containing the service function body. When a message is received on this channel, the service function is executed. The second additional item is a message buffer used by the service function. The name of this message buffer is given by `msg_ptr_name`, the type is given by `type_specifier`. This buffer is used to hold the message received from the client that invoked the service function, and is passed to the second function containing the body of the service function.

4.2.7 Communication Function Calls

There were three library functions provided for message passing mentioned previously: send, receive, and optreceive. Function calls to any of these three library functions are handled differently than other function calls. In the MPL grammar, *send, receive,* and *optreceive* are all reserved words. The MPL syntax for all of these calls is the following:

```
<function-name> (<channel-name>, <parameter-2>);
```

`Channel-name` should be a previously declared channel, and `parameter-2` should be a pointer. These function calls must be compiled differently, since these are not the actual parameters used when the call is generated. In the case of a call to send, the actual parameters must be as follows:

```
send (<channel-id>, <parameter-2>, <channel-size>);
```

In the case of a call to either receive, or optreceive, the parameters required are:

```
receive | optreceive (<channel-type>, <channel-id>, <parameter-2>, <channel-size>);
```

The `channel-type` for a receive or optreceive call is an integer generated by the compiler that will indicate an *in, in_first,* or *in_last* channel.

When one of these three function calls are encountered, there are special rules in the grammar to handle it. A function in *maruti.c* is called which generates the appropriate parameters, and then the function call itself. These function calls are generated as mentioned above for the calls to *maruti_eu().* The `channel-name` specified by the user is used to obtain the necessary parameters. Given the channel name, the linked list of channels is searched to find the corresponding channel, then the `channel-id` and the `channel-size` are obtained from that node in the linked list. There is also some type checking done at this stage. The compiler verifies that only an outgoing channel is specified for a send call, or an incoming channel for the receive and optreceive calls. The compiler also checks that any channel referenced has been previously defined.

The grammar for MPL was modified so that a call to any of the communication functions may occur anywhere that a primary expression occurs, since that is where other function calls are permitted to occur.

4.2.8 Initialization Function

The user-defined function *maruti_main()* is compiled as an ordinary C function.

5 PEUG File

The source code of an MPL file is broken up into *elemental units.* Each elemental unit identifies the resources that it requires. These elemental units are used later in the development process for scheduling the application. The output file created by the MPL compiler creates a *Partial Elemental Unit Graph* (PEUG) for the given source file. The name of this file is the name of the source file, with the *mpl* extension replaced by an *eu* extension.

There are several different types of information recorded in this PEUG file.

5.1 Module Name

The first line in the output file indicates the name of the module, and will appear as:

```
peug <module-name>
```

The `module-name` is taken directly from the module name specification given in the MPL source file.

5.2 File Name

The second line in the source file indicates the name of the target file that is created by the compiler, where `file-name` is the target:

```
file <file-name>
```

5.3 Shared Buffers

Each time a shared buffer is declared its name and type information is recorded to the output file:

```
shared <shared-buffer-name> : (type-description-string>, <type-size>)
```

The `type-description-string` and `type-size` of a shared buffer is obtained from the type specification, and is represented in the same manner as the type and size for a channel. Although the shared buffer is actually a pointer to the type it is declared as, the `type-description-string` represents the object being pointed to, and not the pointer itself.

5.4 Entry, Service, and User Function Definitions

In MPL, a user may define ordinary functions in addition to the entry and service functions that are permitted in MPL. For each entry, service or ordinary user-defined function, there is an entry in the output file. This entry has the following format:

```
<function-type> <function-name>
        .
        .
        .
        size <stack-size>
```

`Function-type` can be either function, entry, or service, indicating which type of function is being defined. `Function-name` is the declared name of the function in the source file. `Stack-size` is the maximum stack size needed by this function. This `stack-size` includes the arguments pushed onto the stack preceding any function calls occuring within the function body. There will also be other information concerning the body of the function that will appear between the `function-name`, and the `stack-size`. The entry for the *maruti_main()* function will be the same as those for other user defined functions. Entry and service functions will contain some additional information not applicable to ordinary functions that will be described below.

5.4.1 Channels

For each channel that is declared, a description of the channel is written to the output file. These descriptions will occur right after the statement indicating the name of the current function:

```
        <channel-type> <name> : (<description-string>, <size>)
```

The `channel-type` and `channel-name` will be the type and name specified in the source file. The `description-string` and `size` are based on the type specification in the channel declaration. Channel descriptions will occur only in entry and service functions. A service function will always contain at least one channel description, since the syntax of a service function requires a channel be named in the definition. A channel description will also be output for every send, receive, and optreceive call, since these calls require a channel as one of their parameters.

5.4.2 Function Calls

Each time a function call is parsed, there will be a line in the output file:

```
calls <function-name> {in_cond} {in_loop}
```

This line indicates where a function call occurs, and which function is being called. The `in_cond` and `in_loop` indicate if this function call appears within a conditional statement or within a loop. These labels will be seen only if their respective conditions are true.

5.4.3 Communication Function Calls

Any call to a communication function is recorded similarly to other function calls. There is a line indicating the name of the function, as shown above for a function call. In addition, there will be a line describing the channel associated with that communication function call. This line will appear just as the line for the channel definition described above appears.

5.4.4 EU Boundaries

The output file for an MPL source file indicates where each elemental unit (EU) begins by the following:

```
eu <N> {region_list}
```

The N indicates an EU number. Each EU within a source file has a unique number. There are several places where EU boundaries are created:

- *Start of a function*
- *Start of a region*
- *End of a region*
- *Explicit calls to maruti_eu()*

The initial EU occuring at the beginning of a function that is not a service or entry function is a special case. This is always labeled as "eu 0" in the output file, and does not represent an actual EU.

Each EU may also be followed by a list describing one or more regions. This list represents the regions that this EU occurs within. The description of a region appears as:

```
(region-name instance access type)
```

The `region-name` is just that given by the user, and the `type` indicates if a region is *local* (local_region construct) or *global* (region construct). The `access` indicates if the access is read or write. The `instance` indicates the instance of this region within the source file. Each instance for a region within a source file is unique.

6 Conclusions

Basing MPL on C has simplified the development of both the language and its compiler. The language is easy to learn for any programmer that has used C before, since there are a limited number of additional constructs unique to MPL. Using the GCC C source code provided an existing compiler, rather than implementing a new one. The source code for GCC only needed to be modified to handle some additional constructs, and produce some additional output. This made the implementation fairly simple. However, the GCC C compiler also provides some functionality that is not needed by MPL. Much of this functionality provided is not even permitted. These restrictions are not enforced by the compiler, but should be detected within the development cycle.

Prior to the development of the MPL compiler using GCC, compiling an MPL source file required two steps. The source files were initially passed through a pre-compiler to extract the available resource information and parse the MPL constructs. The pre-compiler was responsible for converting the MPL code into valid C code, which was then compiled using a standard C compiler. The new implementation of the compiler eliminates some of the redundant processing that is done when the pre-compiler is used. The information obtained through the pre-compiler already existed in the internal structure used by the GCC compiler. This information just needed to be recorded. Instead of parsing source code files in the two steps independently, the functionality of the pre-compiler has been incorporated into the compiler itself. The MPL compiler provides a single tool that extracts all the available information at the initial stage of develpment.

In the future, a version of MPL may be implemented that is based on the Ada programming language. GNAT is a compiler for Ada 9X that is being developed at NYU. GNAT depends on the backend of the GCC compiler. Using the source code for GNAT, an implementation of MPL based on Ada would be similar to the current implementation based on C.

Appendix

A MPL File

The following is a sample of MPL source code:

```
module timer;

typedef struct {
   int seconds;
   int minutes;
   int hours;
   } time_type;

shared time_type global_time;

maruti_main(argc, argv)
int argc;
char **argv;
{
global_time->seconds = 0;
global_time->minutes = 0;
global_time->hours = 0;

return 0;
}

entry update_second()
out disp : time_type;
{
time_type msg;

region time_region {
   global_time->seconds++;
   if (global_time->seconds == 60)
      global_time->seconds = 0;
   msg = *global_time;
   }

send (disp, &msg);
}

entry update_minute()
out display : time_type;
{
time_type msg;

region time_region {
   global_time->minutes++;
```

```
   if (global_time->minutes == 60)
      global_time->minutes = 0;
   msg = *global_time;
   }

send (display, &msg);
}

entry update_hour()
out display : time_type;
{
time_type msg;

region time_region {
   global_time->hours++;
   if (global_time->hours == 24)
      global_time->hours = 0;
   msg = *global_time;
   }

send (display, &msg);
}

service display_time(inchan : time_type, time)
{
printf("Current Time: %d : %d : %d", time->hours, time->minutes, time->seconds);
}
```

B PEUG File

The corresponding PEUG file for the source code above is:

```
peug timer
file timer.o
shared global_time : ($(iii), 12)
function maruti_main
        eu 0
        size 4
entry update_second
        out disp : ($(iii), 12)
        eu 2
        eu 3 (time_region 1 W global)
        calls maruti_eu
        eu 4
                calls maruti_eu
                calls send
                out disp : ($(iii), 12)
        size 32
entry update_minute
        out display : ($(iii), 12)
        eu 5
        eu 6 (time_region 2 W global)
                calls maruti_eu
        eu 7
                calls maruti_eu
                calls send
                out display : ($(iii), 12)
        size 32
entry update_hour
        out display : ($(iii), 12)
        eu 8
        eu 9 (time_region 3 W global)
                calls maruti_eu
        eu 10
                calls maruti_eu
                calls send
                out display : ($(iii), 12)
        size 32
service display_time
        in inchan : ($(iii), 12)
        eu 11
                calls optreceive
                in inchan : ($(iii), 12)
                calls printf
        size 52
```

References

[1] James da Silva, Eric Nassor, Seongsoo Hong, Bao Trinh, and Olafur Gudmundsson. Maruti 2.0 Programmer's Manual. Unpublished.

[2] Manas Saksena, James da Silva, and Ashok Agrawala. Design and Implementation of Maruti-II. In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 4. Prentice Hall, 1995.

[3] Richard Stallman. The GNU C compiler, version 2.5.8., Manual. Info file obtained from gcc.texi in source code distribution.