AGH University of Science and Technology
Computer Science Laboratory
Department of Automatics
Al. Mickiewicza 30
30-059 Kraków, POLAND

The Hexor robot control software.

Piotr Matyasik
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND
ptm@agh.edu.pl

Piotr Zięcik
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND
kosmo@agh.edu.pl

Published online: xx.06.2008

CSL Technical Report No. x/2008

CSL Technical Report No. x/2008

The Hexor robot control software.*

Piotr Matyasik
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND
ptm@agh.edu.pl

Piotr Zięcik
AGH University of Science and Technology
Department of Automatics
Kraków, POLAND
kosmo@agh.edu.pl

Abstract. This paper presents the new *Hexor* control application providing Prolog level access to the robot. The details of the software design are presented. The *Embedded Prolog Platform* is introduced as a middle layer between the C code and Prolog providing basic Real-Time support and simplifying application development. Motivations and differences from the original design are discussed.

Keywords: Mobile robots, Hexor, Prolog, EPP, Embedded Systems, Prolog, Linux.

1 Introduction

HexorII is an autonomous 6-legged intelligent robot developed by *Stenzel* (www.stenzel.com.pl) company for educational purposes. It runs the Bascom AVR [6] based code on its main board and win32 application on the host computer. Robot application allows to control basic functions of the machine, like movement, head position, broadcasting and monitor data from the sensors. The *ActiveX* control is also provided for controlling Hexor from custom made applications. Unfortunately, that limits its usage to software platforms supporting it.

An alternative to the mentioned earlier *ActiveX* component is presented below [5] [4]. It is the C language library providing a set of functions for controlling the robot. Moreover, additional layers build on top of it are introduced witch allows for writing Hexor's programs in Prolog.

The following sections present *Hexor* itself: The C language control library with detailed API specification, the EPP helper component, the Prolog layer and finally, some examples.

2 The Hexor robot

Hexor is a modular, metal-based construction which can be easily extended by additional components. It has a scorpion like shape with characteristic bended tail. On top of the tail, the camera and sonar are mounted, which can be positioned by two servos. The video and sound from the camera are transmitted to the host computer by the radio. The sonar mounted with the camera can measure distance to objects from 3 to 300 cm.

*The paper is supported by the *Hekate* project.

The robot is moved by only three servos: one for tilt, two for forward and backward leg moving. With this kind of mechanical construction Hexor executes *the fast insect movement algorithm* [8].

Hexor is fitted with four infrared proximity sensors; three mounted at the back, and one mounted at the front, where there are also two touch sensors (tentacles).

The electronic design consists of:

- power source (valve regulated lead acid battery)
- DC/DC converters
- main board

The electronic board fitted on Hexor's hull is equipped with three microcontrollers. The main microcontroller is ATmega128 [1]. The tasks of that chip are the following:

- scanning sensors (sonar, tentacles, infrared),
- generating signals for servos (PWM),
- executing the movement algorithm,
- communicating with the host computer,
- executing higher level algorithm (obstacle avoidance, obstacle search, etc.)

The second microcontroller is ATmega8 [2]. It is a helper controller used mainly for extensions (speech synthesis, speech recognition, etc.). The third microcontroller is a part of a radio communication module. It is responsible for maintaining reliable bidirectional communication with the host computer. Every microcontroller can be easily reprogrammed using special connectors mounted on the *Hexor* main board. A proper code execution is guaranteed by the hardware watchdog chip also fitted on the board.

The control algorithm implemented in Hexor robot is presented in Figure 1. After a reset the main controller sets up *IO* ports and communication and then reads configuration data from *EEPROM* memory. After that, it sets servos to neutral position. From that point the robot is ready to receive commands from the host computer. After receiving a command, the robot decodes the frame and executes appropriate action. After completing an action, the robot sends a confirmation frame, possibly with some data. The movement related command is an exception to that rule. If such a command is received, the robot confirms command with *ACK* frame and then starts to move. While moving *Hexor* does not respond to commands until a full step is made.

The software for Hexor is written with the BascomAVR [6] Basic dialect. It relies on its predefined procedures for:

- serial communication (UART),
- *I2C* bus procedures,
- servo control,
- time measurement. Such approach allows for very fast program development. Unfortunately, this approach does not allow the robot to perform multiple tasks simultaneously.

3 EPP

The EPP [7] consists of the three major parts: the Prolog interpreter with an appropriate *Prolog Abstraction Layer*, two *Call Engines* synchronous and asynchronous as well as the *real-time supervisor*. Connections and dependencies between these are illustrated in Figure 2.

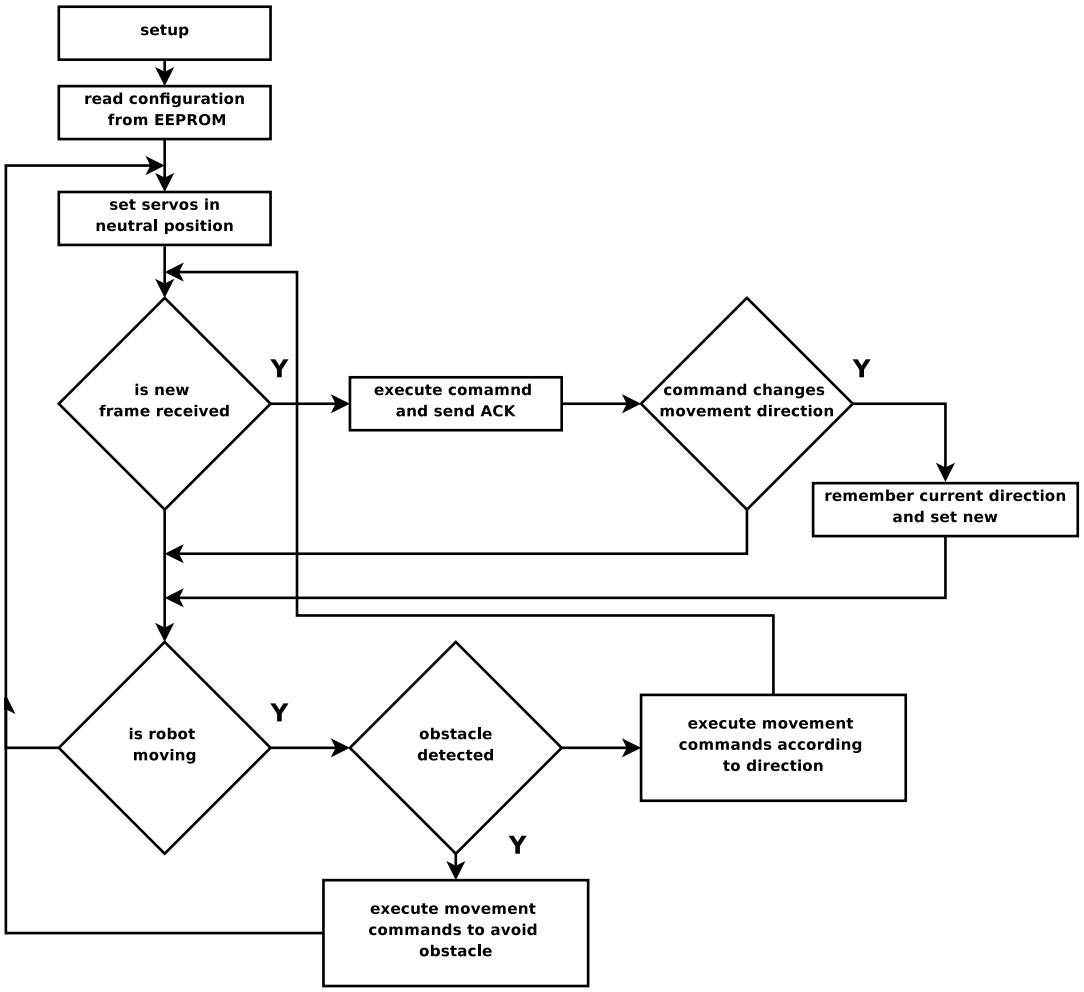

Figure 1: The algorithm of Hexor robot.

3.1 Prolog Abstraction Layer

The Prolog Abstraction Layer is required because major differences in the Prolog implementations exist. Everyone provides their own API for interfacing with other programming languages, with their own data types and calling scheme. For example, in *SWI-Prolog* data exchange between Prolog and C is provided through universal data type (`term_t`) converted to C data type by a programmer. A different approach is presented by *GNU Prolog* which makes conversion automatic by restricting types used in predicates while communicating with a low level language. There are also differences in provided features, like *Constraint Logic Programming* (not supported by all interpreters).

The PAL unifies Prolog $\leftrightarrow$ C API used by EPP itself, and software based on it, making them independent from Prolog implementations. Additionally, it provides an automatic data conversion with one exception: C functions exported to Prolog as predicates use universal data types.

The API provided by PAL is simpler than the ones exported by the Prolog interpreters. As a results of internal checks, it is also more resistant to the programmer's mistakes. Wrong usage simply generates an error and does not destabilize a platform in opposition to standard Prolog $\leftrightarrow$ C interfaces, where small error often causes full application crash.

3.2 Call Engines

The EPP contains two call engines, asynchronous and synchronous, one for each call type. Synchronous calls block calling thread or process. Asynchronous does not block them but works similarly to interrupts. All calls to the Prolog engine are prioritized. The details about the execution

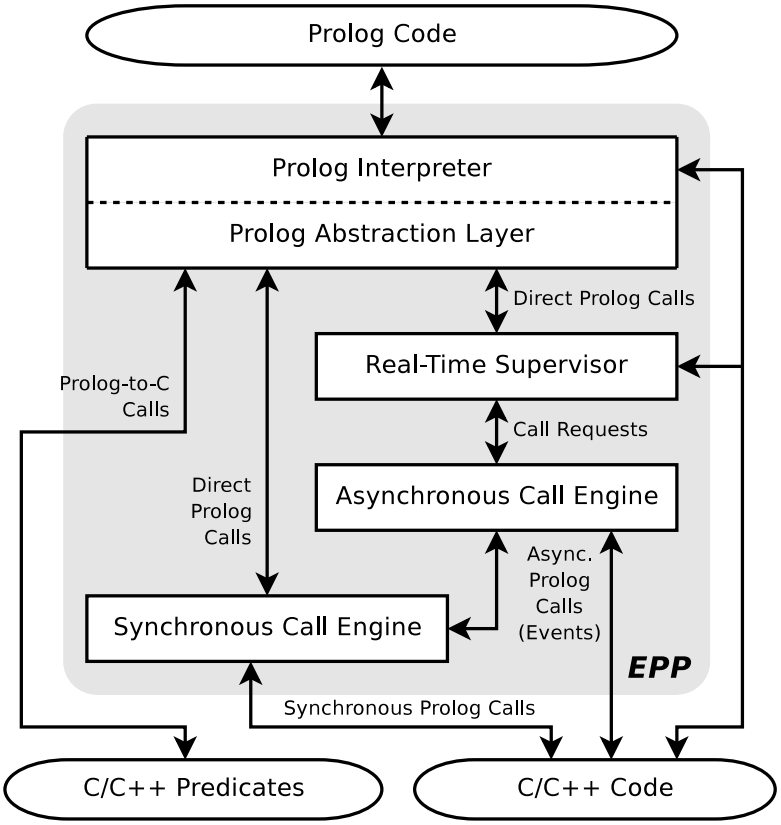

Figure 2: EPP Architecture

scheduling are presented in section 3.3.

The synchronous call engine uses the asynchronous facility to make a call with a specified priority. If an application does not require real-time capabilities, calls may be executed directly through the PAL.

The asynchronous engine uses the real-time supervisor for executing calls. After the execution, data generated by the call can be obtained through a callback function or via predicate written in C and executed from Prolog. The EPP's system asynchronous calls can carry data in opposition to interrupts.

3.3 Real-Time Supervisor

The *Real-Time Supervisor*, presented in detail in Figure 3, is the most important part of EPP. It provides support for the parallel and asynchronous call execution and adds real-time capabilities to Prolog. The *Real-Time Supervisor* relies on multithreading support in Prolog. In most implementations, every thread is a separate Prolog interpreter working on a common prolog code. The internal states of the interpreters are not shared, however, facts database is automatically synchronized at the end of each query (see Figure 4).

Unlike other real-time Prologs which are written in real-time environment and limited in functionality, the *Real-Time Supervisor* is constructed in a different way. It uses a normal and fully featured Prolog implementation with multithreading support. Secondly, it routes calls with a given priority to the adequate thread (engine). As a result, the real-time operating system can transparently manage the Prolog execution, without any modification of the Prolog interpreters.

For example, in a system with two thread priorities, *high* and *low*, EPP is executing the *low* priority call in the *low* priority thread. When the user requests a *high* priority call, the *Real-Time Supervisor* will route it to the *high* priority thread with the attached Prolog engine. Operating system gives this call more system resources suspending the *low* priority thread execution, and resuming it

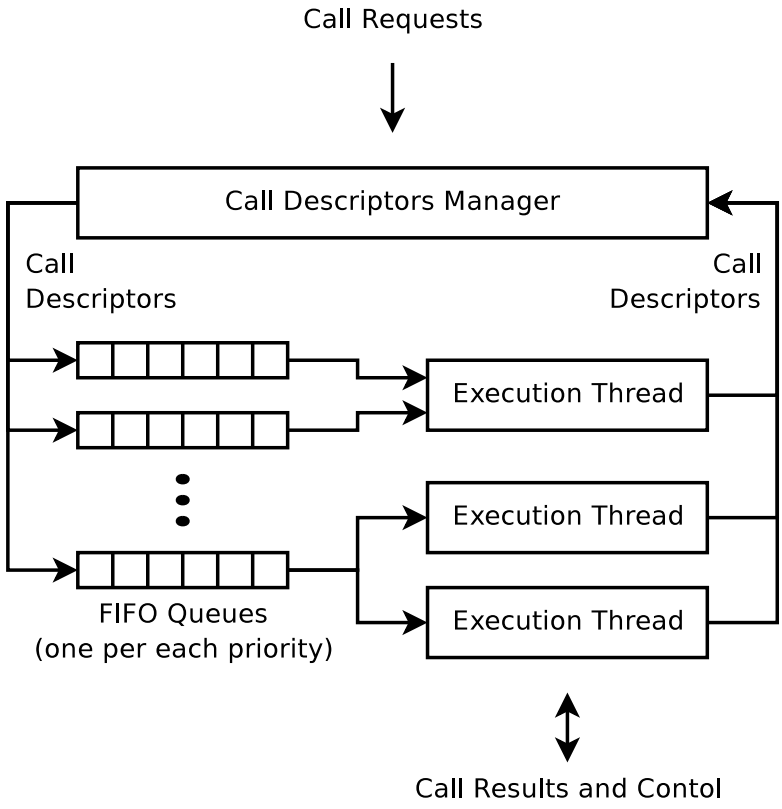

Figure 3: Real-Time Supervisor internals

back after the *high* priority call ends.

Priority to thread mapping may be other than one to one. The user can assign more than one thread to one priority (execution will be parallelized) and more than one priority to one thread. In the second case, thread will always execute the call with the highest available priority. Nevertheless, parallelized execution can be a possible source of problems because earlier query execution does not mean earlier completion.

The execution of the higher priority call may finish after the one with a lower priority depending on complexity of the given task and the amount of the data that need to be processed. This can cause errors when a low priority call uses data produced by preceding it in a short time high priority call (see Figure 4). The *Real-Time Supervisor* solves this problem by delaying the execution of calls on selected priority levels until all calls with higher or equal priorities finish.

The *Real-Time Supervisor* is working with full functionality only in multithreading environment, EPP can be used on platforms which do not support multitasking with the following limitations:

- only one Prolog call at the same time is possible,
- synchronous calls are directly executed and priorities are not supported,
- asynchronous call priorities support is limited,
- asynchronous calls are executed through a pooling mechanism.

3.4 Programming Interface

Programming interface exported by EPP can be divided into three groups:

1. functions providing direct access to Prolog engine internals (API exported by PAL),

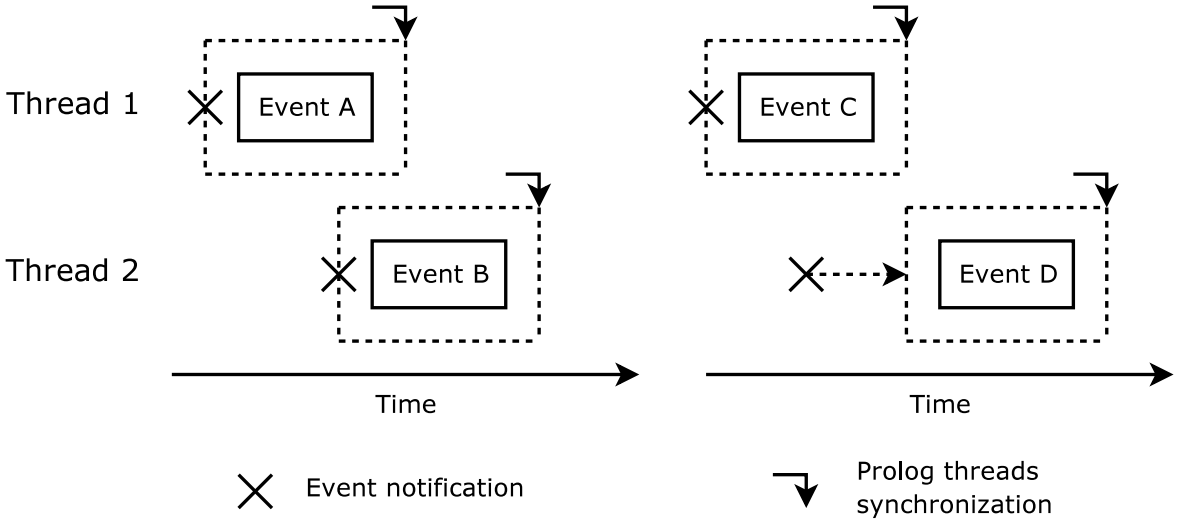

Figure 4: Prolog Threads synchronization and Delayed Execution example.

2. functions providing access to Prolog engine through Embedded Prolog Platform components,

3. functions for Embedded Prolog Platform management.

It is important to distinguish between different function types - some calls are valid only when the Prolog interpreter is in a specific state and its improper usage can destabilize the Prolog's interference engine. In regard to these restrictions, different function types have different prefixes in EPP.

All functions and data types with `pl_` prefix provide the direct access to the Prolog interpreter internals and should be used with extreme caution. Calls with names starting with `epp_` are exported by EPP and allow unrestricted and safe Prolog usage. The `EPP_` prefix is used for platform management calls.

3.5 Prolog management API

```
int pl_init_from_state(const char *statefile)
```

Initializes the Prolog Engine from a given *statefile*. This call MUST be invoked at the application startup before any other call to Prolog and/or EPP. It MUST be invoked from the main application thread also. *Statefile* can be generated from the Prolog code by the *epp-compile* utility.

```
int pl_shutdown(void)
```

Shutdowns the Prolog engine and EPP. This call SHOULD be invoked just before the application exit.

```
int pl_register_predicate(const char *name, unsigned int arity,
pl_predicate (*function)())
```

Registers a given *function* as the Prolog predicate *name/arity*. Examples of predicates written in C can be found on Listing 2.

```
int pl_console(void)
```

Starts the Prolog console if used interpreter has one. The console provides direct access to Prolog and can be used for application debugging and monitoring (see Listing 2).

```
int pl_attach_engine(void)
```

Attaches the Prolog engine to a current thread. This call MUST be invoked at the start of each application thread which uses `pl_*` calls for communication with Prolog, except the main thread (the one which invoked `int pl_init_from_state()`).

Directive	C/C++ data type
l / L	long / long *
f / F	double / double *
s / S	const char * / char **
p / P	void * / void **
_	(none)

Table 1: Parameters specification directives mapping to C data types.

`int pl_detach_engine(void)`
Detaches the Prolog engine from the current thread. This call MUST be invoked at the end of each application thread which uses `pl_*` calls for communication with Prolog, except the main thread (the one which invokes `pl_shutdown()`).

`int pl_is_engine_attached(void)`
Returns `PL_TRUE` if a calling thread has the Prolog engine attached.

`int pl_is_engine_idle(void)`
Returns `PL_TRUE` if a calling thread has the Prolog engine attached and it is not executing a query.

3.6 Prolog query API

The Prolog Abstraction Layer provides simple, but powerful interface for making the Prolog queries. Call convention is similar to standard C `printf()` function. Each predicate argument must be specified as exactly one character (representing data type) in *params_spec* string. The case of these characters represents the data transfer direction. Lowercase characters represent C to Prolog data conversion and uppercase Prolog to C. The list of supported data types with representation parameters specification can be found in Table 1.
`int pl_open_query(pl_query *query, const char *predicate, const char *params_spec, ...)`
`int pl_vopen_query(pl_query *query, const char *predicate, const char *params_spec, va_list args)`
The above function opens full featured Prolog query. Parameters are as follow:

- `pl_query` – the pointer to `epp_query` data type holding the query state,
- `predicate` – the predicate name to call,
- `params` – the *params* string specification as presented in Table 1.

```
1 pl_query  q;
2 volatile long r;
3
4 // predicate(+,-)
5 pl_open_query(&q, "predicate", "sL", "Argument 1", &r);
6 while(pl_next_solution(&q))
7   printf("Query Result: %l\n", r);
8 pl_close_query(&q);
```

Listing 1: Prolog query example

`int pl_next_solution(pl_query *query)`
Requests next solution from the given *query*.

`void pl_close_query(pl_query *query)`
Closes the given *query*. After calling this function, no other results related to specified *query* can be obtained from the Prolog engine.
`int pl_call(const char *predicate, const char *params_spec, ...)`
`int pl_vcall(const char *predicate, const char *params_spec, va_list args)`
Invokes a simple Prolog query - only the first solution is returned. Parameters are as follows:

- `predicate` – the predicate name to call,
- `pl_query` – the pointer to `epp_query` data type holding the query state,
- `params_spec` – the *params_spec* string specification as presented in table 1.

For every parameter specified in *params_spec* string an appropriate C variable must be provided as a list of names or via *va_list* structure.

It is forbidden to execute any of the presented functions in the definitions of the Prolog predicates.

3.7 Prolog predicate API

```
int pl_term2long(long *i, pl_term t)
int pl_term2string(const char **s, pl_term t)
int pl_term2float(double *d, pl_term t)
int pl_term2pointer(void **p, pl_term t)
```
Converts Prolog term to C data type. Returns `PL_TRUE` if the conversion was successful. Can be used only inside predicates.

```
int pl_long2term(pl_term t, long i)
int pl_string2term(pl_term t, const char *s)
int pl_float2term(pl_term t, double d)
int pl_pointer2term(pl_term t, void *p)
```
Converts C data type to Prolog term. Returns `PL_TRUE` if the conversion was successful. Can be used only inside predicates.

```
int pl_longUterm(pl_term t, long i)
int pl_stringUterm(pl_term t, const char *s)
int pl_floatUterm(pl_term t, double d)
int pl_pointerUterm(pl_term t, void *p)
```
Unification functions. Returns `PL_TRUE` if the unification was successful. Can be used only inside predicates.

3.8 EPP query API

```
int epp_call(unsigned int priority, const char *predicate, const char
*params_spec, ...)
int epp_vcall(unsigned int priority, const char *predicate, const char
*params_spec, va_list args)
int epp_open_query(epp_query *query, unsigned int priority, const char
*predicate, const char *params_spec, ...)
int epp_vopen_query(epp_query *query, unsigned int priority, const char
*predicate, const char *params_spec, va_list args)
int epp_next_solution(epp_query *query)
```

`void epp_close_query(epp_query *query)`
These calls are similar to the Prolog query API. If a priority is equal to `EPP_DIRECT_CALL`, then queries are executed directly in the current thread. Elsewhere queries are routed through the Real-Time Supervisor with a given priority. With `EPP_DIRECT_CALL` priority, there are no differences in the execution with `pl_call(...)` and `pl_vcall(...)` functions.

```
int epp_acall(epp_notify_function callback, void *userdata, unsigned
int priority, const char *predicate, const char *params_spec, ...)
int epp_vacall(epp_notify_function callback, void *userdata, unsigned
int priority, const char *predicate, const char *params_spec, va_list
args)
```
The above function prototypes represent asynchronous query API. The parameters are as follows:

- `callback` – the callback function name,
- `userdata` – the pointer to user data,
- `priority` – the priority of call,
- `predicate` – the predicate name to call,
- `params_spec` – the *params_spec* string specification as presented in table 1.

3.9 EPP management API

`unsigned int EPP_Process(unsigned int maxevents, unsigned int min, unsigned int max, unsigned int flags)`
Executes `maxevents` queries queued in Real-Time Supervisor with priorities between `min` and `max` in current thread. This call will block waiting for new queries unless `EPP_NO_BLOCK` flag was specified. Number of executed queries is returned.

`void EPP_ProcessUnblockAll(void)`
Unblocks all `EPP_Process()` calls waiting for new queries.

3.10 Examples

The Example from Listing 2 shows how to create Prolog predicate in C. The `hello_predicate` function is defined. This function will be called after running associated predicate in the Prolog code. The function must return `pl_predicate` value and may only take `pl_term` type as an argument. The function from the example takes one string argument and prints it to the standard output. In the *main* function `pl_register_predicate` is called which connects the C language function pointer to the Prolog predicate name. This example also shows how to run a console in one of the threads. In more complicated applications, a console may by attached to one of many Prolog executing threads, allowing for debugging and monitoring the program.

Listing 3 presents an asynchronous call example. In the `main` function the Prolog engine is initialized and `epp_acall` function is called. All solutions are returned by the registered earlier `callback` function. The *callback* function may return two values:

- `EPP_NEXT_SOLUTION` – then next solution is generated if it exists,
- `EPP_CLOSE_QUERY` – then no other solutions are returned.

The execution of the `epp_acall` function does not block the program execution. The call to the Prolog engine is started in `EPP_Process` function. Before starting `EPP_Process` function, more

```
1  #include <stdio.h>
2  #include <malloc.h>
3  #include <epp/epp.h>
4
5  pl_predicate hello_predicate(pl_term a) {
6      const char *str;
7
8      if (pl_term2string(&str, a) == PL_TRUE) {
9    printf("Hello %s !\n", str); free(str);
10   return(PL_TRUE);
11     }
12     return(PL_FALSE);
13 }
14
15 int main(int argc, char *argv[]) {
16     pl_register_predicate("hello", 1, hello_predicate);
17     pl_init_from_state(*argv);
18     pl_console();
19     pl_shutdown();
20     return(0);
21 }
```

Listing 2: Prolog predicate written in C

than one call may by queued. Every solution returned by the Prolog call execution runs an appropriate callback function once.

The Prolog code related to this example (see Listing 4) are simple three predicates with integer data.

The example from the Listing 5 is functionally identical to the previous one. The major difference is how the Prolog engine is executed. In this code, Prolog runs from a different thread and executes all queued calls according to `EPP_Process` function specification. According to that scheme, more than one thread may by created and more than one `EPP_Process` functions may be executed simultaneously. A proper choice of the `EPP_Process` function parameters allow for prioritized Prolog goals execution. The `EPP_ProcessUnblockAll` function (see line 33) allows computation to start in all created threads.

4 The Hexor Control Software design

The *Hexor* robot is controlled from the host computer via wireless serial link. The control adapter is connected to the *USB* and recognized by the Operating System as an additional serial port. Under Windows OS it is *COMX:* and under Linux */dev/ttyUSBX*. Control adapter is based on FTDI (FT232BM) RS232 to USB converter and other operating systems also support it(FreeBSD, MacOS).

Communication between the Hexor robot and the host computer is always initiated by the latter. Communication protocol is based on frames presented in Figure 5. The upper diagram presents the frame without arguments. The lower diagram shows one with arguments.

The CRC of every frame id computed by algorithm from Listing 6. The size of the variable *frame* should be at least one byte bigger than the size of the frame with data. The CRC of the frames received from the robot is computed by the same algorithm.

4.1 Hexor control commands

The Hexor robot recognizes control commands presented in Table 2. The *ACK* in the response column means that the robot sends the frame with no data and the same command code.

```c
#include <stdio.h>
#include <epp/epp.h>

int callback(int retval, void *userdata) {
    static unsigned int call = 1;
    long *i = (long*) userdata;

    if (retval == PL_TRUE) {
  printf("<CALL %u> TRUE:\t %li\n", call++, *i);
    } else if (retval == PL_FALSE) {
  printf("<CALL %u> FALSE\n", call++);
    } else {
  printf("<CALL %u> ERROR\n", call++);
    }
    return(EPP_NEXT_SOLUTION);
}

int main(int argc, char *argv[]) {
    long i;
    pl_init_from_state(*argv);
    epp_acall(callback, &i, 0, "counter", "I", &i);
    EPP_Process(-1, 0, 0, EPP_NO_BLOCK);
    pl_shutdown();
    return(0);
}
```

Listing 3: Asynchronous call C code

```
counter(1).
counter(2).
counter(3).
```

Listing 4: Asynchronous call Prolog code

The *move* command requires an additional data presented in Table 3. Data sent with this command determines whether the robot moves in a particular direction or stops. After receiving a movement command the robot starts to move in a specified direction until the next movement related command is received or sensors detects an obstacle.

The *read sonar and thermometer* command returns two bytes of data. First of them is a sonar reading whch is proportional to the distance from obstacles. The sonar effective range is between 3 and 300 cm. The second value is a temperature measured by a digital thermometer mounted on Hexor's main board. The measurement is in the Celsius degrees scale.

The *set head position* command requires two parameters. The first one is a horizontal position. The effective values for that parameter vary from 0x44 to 0xf5 and they correspond to the maximum left and maximum right position. The second parameter is a vertical position. Setting this value to 240 levels the camera head.

The *control external devices* command allows for turning on and off the power for additional devices. Data sent with this command is a bit mask that represents new power state. If the bit is set, the device will be turned on. The detailed bit description is presented in Table 4.

The *read thermometer* command returns current temperature measured by the sensor. Description of the data is analogical to *read sonar and thermometer*.

The *send new robot id* command permits to change the robot's personal id. The data sent with this command is a byte value representing a new robot's id. After that operation, the new robot's id should be used in communication.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <epp/epp.h>

void *thread(void *unused) {
    pl_attach_engine();
    EPP_Process(-1, 0, 1, 0);
    pl_detach_engine();
    return(NULL);
}

int callback(int retval, void *userdata) {
    static unsigned int call = 1;
    long *i = (long*) userdata;
    if (retval == PL_TRUE) {
  printf("<CALL %u> TRUE:\t %li\n", call++, *i);
    } else if (retval == PL_FALSE) {
  printf("<CALL %u> FALSE\n", call++);
    } else {
  printf("<CALL %u> ERROR\n", call++);
    }
    return(EPP_NEXT_SOLUTION);
}

int main(int argc, char *argv[]) {
    pthread_t t;
    long i;
    pl_init_from_state(*argv);
    pthread_create(&t, NULL, thread, NULL);
    epp_acall(callback, &i, 0, "counter", "I", &i);
    usleep(100*1000);
    EPP_ProcessUnblockAll();
    usleep(100*1000);
    pl_shutdown();
    return(0);
}
```

Listing 5: Asynchronous call C code with multithreading

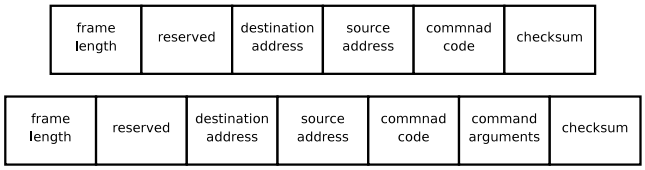

Figure 5: The Hexor Communication Frames.

The *read sensors* command returns the current state of all *on/off* sensors including infrared and touch ones. The meaning of the sensors state byte is described in Table 5. The sensor is active when the corresponding bit is cleared.

4.2 C layer

The C Layer is a helper library providing functions for communicating with Hexor robot via serial link. The main goal of the library is to provide a reliable bidirectional link between the robot and the

```
1  void calculateCRC(unsigned char *frame)
2  {
3      unsigned char crc = 0;
4      int x = 0;
5      int len = frame[0];
6      const char crctab[256] =
7          { 0x00, 0x5E, 0xBC, 0xE2, 0x61, 0x3F, 0xDD, 0x83, 0xC2, 0x9C,
8      .. /* 256 values */
9            0x0A, 0x54, 0xD7, 0x89, 0x6B, 0x35 };
10     frame[0]++;
11     for (; x < (len); x++) {
12         crc = frame[x] ^ crc;
13         crc = crctab[crc];
14     }
15     frame[len] = crc;
16 }
```

Listing 6: CRC value computation

Command description	Command code	Data IN	Response
move	0x20	direction id	ACK
read sonar and thermometer	0x21	none	sonar and temperature readings
set head position	0x23	X,Y coordinates	ACK
control external devices	0x24	new devices state	ACK
send new robot id	0x26	new robot id byte	ACK
read sensors	0x27	none	sensors state
read thermometer	0x28	none	temperature readings

Table 2: Hexor Control Commands

host computer. Hexor robot is connected with the host via a wireless serial link. This requires a stable and fault tolerant middleware as a communication layer.

To allow the programmer to control the robot from multithreaded engine and provide error reporting facility, a few elements were taken into consideration during the development of the control library. First of all, sending a command to the robot is a complex activity. It consists of sending a frame with a command to robot and receiving a proper answer frame from the machine. That means a robot may have received a proper frame with new command to execute, it may have started executing it and a problem occurred while transmitting return frame to a host computer. Because of that, it is safe to send a command that brings the robot to a known state (for example the *stop* command) after receiving an error.To allow using library in multithreaded programs, only one command sequence may be executed at a time. A command sequence is sending a command and receiving a proper response or a timeout. To detect communication failures, the timeout is set before sending the frame with a command to the robot. The timeout equals two seconds and it is determined by the algorithm that runs on the robot (see Figure 1). If a proper communication frame is received before timeout the command sequence is considered to be successful, otherwise it is a failure.

Direction	Stop	Forward	Backward	Left	Right
Data	0x00	0x01	0x04	0x02	0x03

Table 3: Move command data description

Bit no.	7	6	5	4	3	2	1	0
Description	NC	NC	NC	NC	NC	Camera	Power connector 2	Power connector 1

Table 4: Power control word bit description

Bit no.	7	6	5	4	3	2	1	0
Description	none	left touch	right touch	front IR	left back IR	none	middle back IR	right back IR

Table 5: ReadIR bit description

4.3 EPP layer

The EPP layer holds the *main* function that starts execution of a program and all definitions of Prolog predicates presented in section 4.4. This is also a place for preparing the multithreaded Prolog engine execution. The detailed description of chosen functions and instructions how to add another predicates to the Prolog API are presented below.

The delay predicate function presented on Listing 7 stops executing thread for specified amount of seconds. The function header is prepared according to EPP specification from section 3. It returns *pl_predicate* value and all arguments are of type *pl_term*. Internally the function uses the *usleep* procedure [3]. The return value of the delay predicate is *true* only if correct parameters are specified (see line 5) and *usleep* was not interrupted (see line 6).

```
1  pl_predicate delay_predicate(pl_term a)
2  {
3      long d;
4
5      if (pl_term2long(&d, a) == PL_TRUE) {
6          if (usleep(d*1000000) == 0);
7          return (PL_TRUE);
8      }
9      return (PL_FALSE);
10 }
```

Listing 7: Delay predicate definition

The *read_sensors_predicate* function from Listing 8 returns temperature and sonar readings from specified unit as Prolog terms. It presents the usage of the robot control library API with EPP API and also how to cope with transferring in/out parameters form C layer to Prolog layer and back. As a result, the new predicate is created for controlling the robot from the Prolog level. Similarly, other predicates may be constructed, related to hardware devices. The predicate returns *true* only if library function and all parameter conversions execute properly.

Listing 9 presents the *main* function of a program. It presents how to register defined C predicates to the Prolog engine, initialize the Prolog interpreter, set up multiple threads execution and firmly shutdown computation. In lines 4-8 the new predicates are registered. In line 9 the Prolog code linked with executable file is inserted into interpreter. In line 11 additional thread is created. In line 13 the Prolog goal is executed in the current thread. Line 15 shuts down the execution of the Prolog interpreter. Detailed descriptions of the presented functions may be found in section 3 and [3]. Additional threads may and may not be related to the Prolog interpreter execution. If they run Prolog goals, the execution will be as presented in section 3.

```
1  pl_predicate read_sensors_predicate(pl_term u, pl_term a, pl_term b)
2  {
3      long temp, sonar;
4      long unit;
5      if (pl_term2long(&unit, u) == PL_TRUE) {
6          if (read_sensors(unit, &temp, &sonar) == 0 && pl_longUterm(a, temp)
7              && pl_longUterm(b, sonar)) {
8              return (PL_TRUE);
9          }
10     }
11     return (PL_FALSE);
12 }
```

Listing 8: Read Sensors predicate definition

```
1  int main(int argc, char *argv[])
2  {
3      pthread_t t;
4      pl_register_predicate("forward", 1, forward_predicate);
5      pl_register_predicate("delay", 1, delay_predicate);
6      /* .. */
7      pl_register_predicate("readSensors", 3, read_sensors_predicate);
8      pl_register_predicate("camera", 3, camera_predicate);
9      pl_init_from_state( *argv);
10
11     pthread_create(&t, NULL, thread, NULL);
12     // pl_console();
13     pl_call("runme1", NULL);
14     pthread_join(t, NULL);
15     pl_shutdown();
16     return (0);
17 }
```

Listing 9: Main function

4.4 Prolog API

The API functions provided by presented in the previous sections software layers are presented below. API functions are written according to prolog syntax, where parameters prefixed with + are input parameters and those with leading − are output ones. In every command the *unit* parameter appears, which is the id number of the designated robot. All movement related commands start the execution of a specified movement until next movement command is received or the robot locates obstacles with its sensors.

1. `forward(+unit)` – start moving forward,
2. `backward(+unit)` – start moving backwards,
3. `left(+unit)` – start turning left,
4. `right(+unit)` – start turning right,
5. `stop(+unit)` – stop robot,
6. `camera(+unit,+x,+y)` – x,y coordinates of sonar and camera Y=240 level, X= 68 - 248 (max left - max right)

7. `cameraOn(+unit)` – start broadcasting data from camera,
8. `cameraOff(+unit)` – stop broadcasting data from camera,
9. `readSensors(+unit,-temp,-sonar)` – read temperature and sonar data from robot,
10. `readIR(+unit,-state)` – read touch and IR sensors state from robot,
11. `delay(+seconds)` - delay specified amount of time.

Every predicate returns "0" if a command was successfully sent to the robot (see section 4.2). Otherwise, a nonzero value is returned and a programmer may take appropriate steps to react to errors.

5 Examples

Here are some simple examples presented to demonstrate the use of the programmers API introduced in Section 4.4.

The first example (Listing 10) presents how to achieve "one step" functions for a robot. According to Hexor's software presented in section 2, sending any movement command results in starting following that direction until another movement related command is received, or a stop command is sent. Thus, sending a movement command followed immediately by *stop* command results in executing a single step by a robot.

```
onestepforward(U):- forward(U), stop(U).
onestepleft(U):- left(U), stop(U).
```

Listing 10: One step example

The example from Listing 11 presents how to read sensors from Prolog. Lines 1 to 3 set up predicates for the program execution. The predicate *counter* holds data used for positioning the sonar head. The predicate *unit* holds the robot unit number and should be set to correct value. The *runme* predicate is executed from EPP level and it is a main goal for scan thread. In the presented example, it only fires another goal – *start*. The *start* predicate turns on the camera, sets the initial camera and the sonar head position and starts *sensors* predicate which actually preforms a scanning task.

The *sensors* predicates has the following goals: the first one, starting in line 6, checks if the camera's head has reached the maximum right horizontal position. If so, it returns the camera's head to the maximum left position. The second *sensors* predicate sets a new head position, reads and displays the sensor data. The values from sonar, IR and touch sensors are raw data. A sonar reading is proportional to the distance from the obstacles. Values returned by the readIR predicate should be interpreted as shown in Table 5.

The program performs scanning in infinite loop. It may be interrupted by a user or by a communication failure.

The example from Listing 12 presents a simple walking demo. The *unit* predicate holds the robot unit number, just like in the previous example. Also *runme* predicate has the same function as in the example from Listing 11. The *gohexor* predicate performs a walking task. It starts the camera and executes the movement in all directions for five seconds. In the end program writes *DONE* on the screen and turns off the camera. The second *gohexor* predicate is executed when a communication error occurs. This example, except for demonstrating programmers API related with movement, also shows tow to cope with errors.

```
1  :-dynamic counter/1.
2  counter(68).
3  unit(11).
4  runme:- start.
5  start:-      unit(U),cameraOn(U),camera(U,68,240),sensors.
6  sensors:-    counter(T),T > 240, retract(counter(_)),
7        assert(counter(68)),!,sensors.
8  sensors:-    unit(U),counter(T),X is T + 10,
9        retract(counter(_)),assert(counter(X)),camera(U,X,240),
10             readSensors(U,A,B),
11             readIR(U,C),
12             write('temp: '),write(A),
13             write(' sonar: '),write(B),
14             write(' ir: '),write(C),
15             nl,!,sensors.
```

Listing 11: Sensor scan example

```
1  unit(11).
2  runme:- gohexor.
3  gohexor:- unit(U),cameraOn(U),forward(U),
4            writeln('forward'),delay(5),
5            backward(U),writeln('backward'),delay(5),
6            left(U),writeln('left'),delay(5),
7            right(U),writeln('right'),delay(5),
8            forward(U),writeln('forward'),delay(5),
9            stop(U),writeln('stop'),writeln('DONE'),cameraOff(U).
10 gohexor:- writeln('ERROR'),cameraOff(U).
```

Listing 12: Sensor scan example

6 Summary

The report presented control software architecture developed for *Hexor* mobile robot in detail. All layers were introduced, starting from the lowest one which holds communication procedures, through EPP component and EPP layer, ending on some usage examples. Initial concept of presented design was not to replace original Hexor's software, but to provide Prolog level API compatible with the original one.

Currently another project is in its development stage. A new software will be downloaded to the robot, instead of original *Basic* code based on embedded real time operating system. This should allow the robot to be asynchronous, and together with EPP, to achieve fully event driven control architecture.

References

[1] Atmel. *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash ATmega128 ATmega128L*, rev. 2467g-avr-09/02 edition, 2002.

[2] Atmel. *8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash ATmega8 ATmega8L*, 2486qs-avr-10/06 edition, 2006.

[3] IEEE et al. Posix programmer's manual. Unix Man Pages.

[4] P. Matyasik and G. J. Nalepa. Knowledge-based control of reactive systems with multi-layer architecture. In A. Napieralski., editor, *MIXDES 2007 : MIXed DESign of integrated circuits and*

systems : proceedings of the 14th international conference : Ciechocinek, Poland, 21–23 June, 2007, pages 667–672. Łódź : Technical University of Łódź. Department of Microelectronics and Computer Science, june 2007.

[5] P. Matyasik, G. J. Nalepa, and P. Zięcik. Prolog-based real-time intelligent control of the hexor mobile robot. In J. Hertzberg, M. Beetz, and R. Englert, editors, *KI 2007 : advances in Artificial Intelligence : 30th annual German conference on AI, KI 2007 : Osnabruck, Germany, September 10–13, 2007 : proceedings*, volume LNAI 4667 of *Lecture Notes in Computer Science. Lecture Notes in Artificial Intelligence*, pages 485–488, Berlin, Heidelberg, september 2007. Springer-Verlag.

[6] MCS Electronics. *BASCOM-AVR*, ver. 1.11.6.3 edition.

[7] G. J. Nalepa and P. Zięcik. Integrated embedded prolog platform for rule-based control systems. In A. Napieralski, editor, *MIXed DESign of integrated circuits and systems*, Łódź, 2006. Technical University of Łódź. Department of Microelectronics and Computer Science.

[8] Stenzel. *HexorII robot manual*, 2006.