A Real Time Environment for Expert Control

Årzén, Karl-Erik

1987

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7314)/1–15/(1988)

A Real Time Environment for Expert Control

Karl-Erik Årzén

Department of Automatic Control
Lund Institute of Technology
December 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
December 1987

Document Number
CODEN: LUTFD2/(TFRT-7314)/1–15/(1987)

Author(s)
Karl-Erik Årzén

Supervisor

Sponsoring organisation

Title and subtitle
A Real Time Environment for Expert Control.

Abstract

A real time environment for experiments with expert control is presented. Different methods for communication between processes on VAX/VMS are discussed. Special attention is paid to communicate between Lisp and Pascal. The Lisp dialect used is Franz Lisp together with EUNICE.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
15

Recipient's notes

Security classification

A REAL TIME ENVIRONMENT FOR EXPERT CONTROL

Karl-Erik Årzén
Department of Automatic Control
Lund Institute of Technology, Sweden

Abstract: A real time environment for experiments with expert control is presented. Different methods for communication between processes on VAX/VMS are discussed. Special attention is paid to communication between Lisp and Pascal. The Lisp dialect used is Franz Lisp together with EUNICE.

1 INTRODUCTION

This report presents a real-time environment suitable as a test bench for experiments with expert control. Expert control refers to a control system where an expert system is used to execute a collection of control algorithms. The reader is assumed to be familiar with the general concepts of AI and automatic control.

Expert control is presented and explained in chapter 2. Chapter 3 deals with a suitable software structure and in chapter 4 the available hardware and software tools are presented. In chapter 5 the real-time environment is presented and in chapter 6 the impact of this environment of Lisp is displayed.

2 EXPERT CONTROL

The notion of expert control was proposed by Åström and Anton (1984). The idea is to remove much of the branching and logic statements from the control algorithms and implement them in an expert system. With this approach it is also possible to augment the control algorithm by incorporation of heuristics and rules of thumb about control in general. A different kind of controller is thus achieved that can include functions difficult to implement in ordinary control algorithms. It is in particular possible to include tuning rules and to achieve a self tuning regulator more general than the existing ones, Åström (1983).

Most of the existing expert system are written in Lisp, Prolog or some of their derivatives. The reasons for this are not only historical. Lisp is well suited for symbolic data processing. Its interactive environment also simplifies program development. The latter is particularly important for large, experimental programs. Control algorithms, however, are preferably written in a traditional, numerically oriented programming language such as Fortran or Pascal. It is thus desirable to use several languages in the implementation of an expert control systems. In this particular implementation the expert system is written in Lisp, and the control algorithms are written in Pascal. These two parts must be able to communicate with each other in some way.

Control is a typical real time problem and ordinary controllers are implemented with this in mind. Communicating concurrent processes are used and care is taken to achieve mutual exclusion and to avoid real time problems such as dead-lock. This concurrency must be taken into consideration in the expert controller. Different priorities are needed for the different parts of the controller. The Pascal system which executes the control algorithms must have a high priority. The Lisp part which searches among large rule sets and is very time consuming could be given lower priority.

3 A PROPOSED STRUCTURE.

A software structure for expert control was proposed in Årzén (1986). A slightly modified version of this can be seen in figure 1. It consists of three processes, Lisp-Expert, Lisp-Io and Pascal-Controller. The Pascal-Controller executes the control algorithms. It consists of a control loop and a library of numerical algorithms. Lisp-Expert is the actual expert system with a database and rule sets. These two processes need to communicate with each other. To start or stop an algorithm or to change the parameters of an algorithm are typical messages from Lisp-Expert. Return messages can be sent when a change in behaviour of the controlled plant has been detected by some algorithm.

Lisp-Io is needed for the man-machine communication. It should handle manual changes of parameters in the controller. Another aspect is that the expert system must be able to explain what it is doing, e.g. which algorithms that are currently running and what the state of knowledge is of the controlled plant. By implementing this process as well in Lisp, substantial simplifications are obtained in the communication with Lisp-Expert. This is explained in detail in chapter 6.

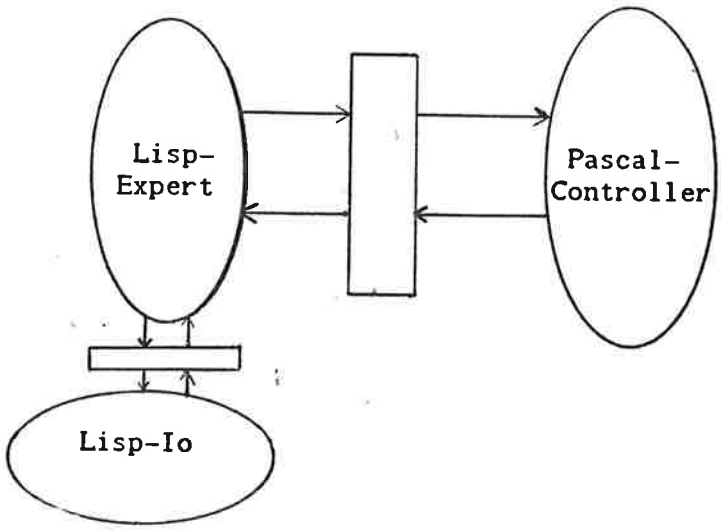

Figure 1. Proposed process structure. The ellipses represent processes and the rectangles represent communication media.

4 AVAILABLE HARDWARE AND SOFTWARE

A VAX 11/780 computer running under VMS is available for the project. The different processes can be implemented as subprocesses having different priorities. The operating system can be used for scheduling. A process can be delayed by calling appropriate system services.

The operating system VMS has three different methods for communication between processes, eventflags, mailboxes, and global sections. Common event flags provide a means for event synchronization of processes. The main operations are setting a flag and waiting for a flag to be set. This method can, however, not be used for exchanging data and it is thus not suitable for our application.

A mailbox is a record-oriented virtual I/O device that can be used by cooperating processes for exchange of messages. There are system services to create, write to and read from a mailbox. A mailbox can hold multiple messages, which are stored on a first-in first-out (FIFO) basis. Mailboxes can be used to implement the ADA rendezvous primitives as shown by Tengvall (1982).

A global section is an area of memory containing data or code that can be shared between communicating processes. Global sections are the most general communication method since it allows processes to have common variables. An implementation of ADA rendezvous primitives with global sections has been done in Elmqvist and Essebo (1982). Using global sections with Pascal works well, since there is a direct correspondence between variables and memory cells. A Lisp symbol, that corresponds to a variable in Pascal, does not, however, correspond directly to a memory cell. Lisp has an internal data structure for each symbol. It is perhaps possible to overcome this in some way and use global sections together with Lisp but my intention have been to keep the internal structure of the Lisp interpreter.

Franz Lisp by Foderaro and Sklower (1981), is the dialect available on our VAX. This Lisp is quite general. It contains most of the common software tools such as structure editor, file package, compiler, debugger, trace and step possibilities, etc. One problem however, is that it is designed for the operating system UNIX ,Unix (1981). EUNICE, Kashtan (1982), is a software package that allows users to run program written for UNIX on VAX/VMS with little or no modification. EUNICE functions as an interface between UNIX and VMS that performs the necessary translations. One of the major differences between VMS and UNIX is their different file format. VMS has a record oriented format and UNIX has a stream oriented format. This is handled by EUNICE, so that each program sees its appropriate format.

I/O data is transfered between Franz Lisp and external media through a port structure. The standard input and output ports are the terminal, but a port can also be connected to a file or to a pipe. A pipe is an UNIX method for communication between different programs.

Franz Lisp allows the user to write his own procedures and functions. These can be written either in C, Pascal or Fortran. The value returned from a function can be an integer number, a real number or an arbitrary, valid Lisp object. The compiled user written functions are dynamically loaded into the Lisp. They can be called and used just like ordinary Lisp functions. Using C as the language has one

major advantage. Since Franz Lisp is written in C, it is possible to use the same data structures to provide valid Lisp objects. It is only necessary to include the Franz Lisp declarations in the user written functions. It is also possible to call both the UNIX system services emulated by EUNICE and the VMS system services using C.

5 THE REAL TIME SYSTEM

The processes can be implemented as VMS subprocesses. Since global sections require an extensive modification of the internal structure of Franz Lisp, mailboxes are the only feasible method for communication. The actual software structure is illustrated in figure 2. Two mailboxes are used as communication media. Lisp-Io uses the same mailbox as Pascal-Controller for communication with Lisp-Expert. There may possibly be one extra mailbox for communication from Lisp-Expert to Lisp-Io, but since most of the messages in this direction will be printed on the terminal, the two processes can share the terminal.

Mailboxes are created with calls to the system service SYS$CREMBX. This can theoretically be done either from Pascal-VMS or from Lisp-EUNICE. In this implementation it is done from EUNICE. The mailboxes can be created in a user written C function which is called from Lisp-Expert. The code for this function can be seen in the appendix 1. The function creates the two mailboxes, starts Pascal-Controller using the system service SYS$CREPRC, and writes the mailbox names in the group logical table so that Lisp-Io can access them. The mailboxes are connected to UNIX files which are then connected to two Lisp ports. The value returned from the function is a list consisting of the two ports. When Pascal-Controller is initialized its standard input and output are set to the two mailboxes.

A mailbox may be accessed from three different levels. The lowest level is the system service level. The service for accessing a mailbox is SYS$QIO. The next level is the VMS Record Management Service (RMS). On this level the mailbox is associated with a file. Access is done with get and put operations. RMS internally uses system services to implement these operations. The third level to access a mailbox is from a high level language. From this level the mailbox can not be separated from an ordinary file. All access is done with ordinary read and write statements.

Much is gained by only using the highest level for accessing the mailbox. The mailbox mechanism becomes a very general way of communication. A message is simply a line of text if the mailbox is associated with a text file. This high level association defaults to a synchronous communication. A process that writes a message into a mailbox waits until the other process has read the message. This is not suitable for my purposes. The time critical process Pascal-Controller can not wait for the time consuming Lisp-Expert to read the message. There is, however, a possibility to change a timeout parameter in the VMS Record Access Block (RAB) so that the communication will be asynchronous. Each VMS file has a associated RAB on the RMS level. It contains information about access of the records in the file. This RAB block can also be accessed from EUNICE because each "UNIX" file in EUNICE has an internal structure that among other things contains the VMS RAB block. Details are shown in the appendices.

Another problem is that Pascal-Controller must have a possibility to check if it has got something to read before it actually reads it. If not, the Pascal-Controller will be halted until a message is sent and the control of the plant will stop. A possibility to check whether a mailbox is empty or not is to use the system service SYS$GETDEV which brings information about different devices. One of the parameters brought back is the number of messages in the box. This can be used as a check before reading.

Another demand is that messages should be inserted into Inbox according to their priorities and not by a FIFO scheduling. This is needed to assure that an alarm message is handled before some other less important message. This can be solved by letting Lisp-Expert have an internal mailbox structure. Instead of reading the first message from Inbox it starts with emptying Inbox and inserting the messages in its own box according to priority before it reads the first message from its own box.

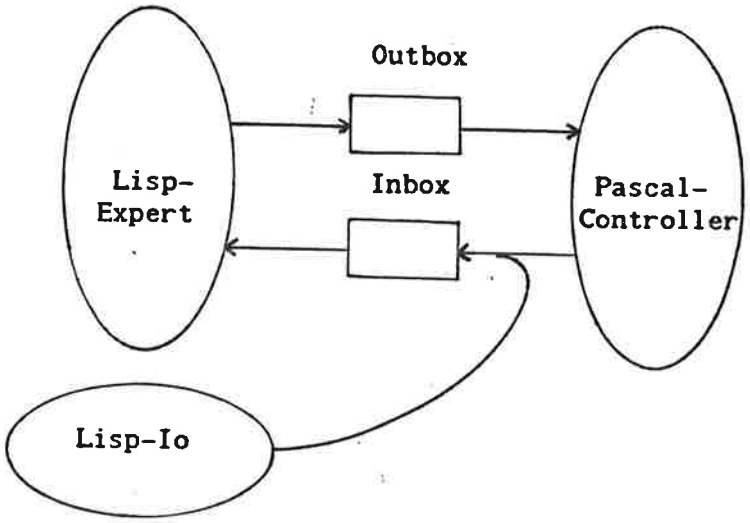

Figure 2. Process structure. The ellipses represent processes and the rectangles represent mailboxes.

Using a text file as a mailbox has a very powerful impact on the communication between the two Lisp processes. It means that the message format is totally free. A message can be an arbitrary Lisp expression. Since Pascal is a compiled language, the message format between Lisp-Expert and Pascal-Controller must be predefined and cannot be altered during execution.

When Lisp-Expert receives a message it can simply test if it is a Lisp expression and if so evaluate it.

(if (Lispexpression message) then (eval message))

The test for legal Lisp expressions can be based on the presence of parenthesis at the beginning and end. It is also very simple to print and read a Lisp expression to a port. Franz Lisp has built-in functions that prints and reads Lisp expressions to ports. They have the format

(print mess outport)

(read mess inport)

To change some symbol in the other process, one needs only send an assignment statement. For example

(print '(setq symbol 'newvalue) outport)

This free message format makes it possible to evaluate Lisp functions in a real time process almost as if it was running interactively.

In particular it is possible to change the code of the executing process on-line. There are three different ways to do this. One is to send a new function definition for an existing function in the other process. Another is to edit the existing function, print it on a file and then send a message to the other process to load this file. The third way is to edit the function, print it on a file, compile the file and then send a message to the process to load this compiled file. This is possible since interpreted and compiled code can be mixed in Lisp. All functions in the process can, in fact, be changed except the function which involve the reading and evaluation. This is no serious limitation since reading and evaluation can be confined to small well defined code sections.

One objection against this on-line changing of code in a real time process is that it is extremely dangerous. The process will stop if an error occurs during evaluation. There is, however, a way around this problem. It is possible to evaluate the message in an environment that catches an error and then returns without stopping. This is easily done with the Lisp function (errset expression). This function evaluates expression and if an error occurs then the function will return the value nil. This can be used as a test for successful evaluation.

This preservation of the interaction is extremely important in a test bench for expert control. It makes it possible to inspect, change and trace the expert system on-line.

CONCLUSIONS

A real time environment has been presented that has all the qualifications to be a basis for further experiments with expert control. An interesting communication method between parallel Lisp processes has been presented that preserves the interaction of Lisp.

REFERENCES

Årzén, K-E (1986): Expert systems for process control, CODEN: LUTFD2/TFRT-7315, Department of Automatic Control, Lund Institute of Technology, Lund Sweden.

Åström, K J (1983): Implementation of an auto-tuner using expert system ideas, Internal report Dep. of Automatic Control Lund Institute of Technology.

Åström K J and J.J. Anton (1984): Expert Control, Proc. 9'th IFAC World Congress, Budapest, Hungary

Digital (1983): VAX/VMS Manuals, Digital Equipments Corp.

Foderaro J K and Sklower K L (1981): The Franz Lisp Manual.

Kashtan D L (1982): EUNICE: A system for porting UNIX programs to VAX/VMS Artificial Intelligence Center, SRI International, Menlo Park Ca.

Tengvall F (1982): Rendezvous primitives for intertask communication on Vax/VMS, CODEN:LUTFD2/TFRT-7234, Dep. of Automatic Control Lund Institute of Technology.

Unix (1981): UNIX Programmer's manual, Computer Science Division, Dep. of Elec. Engineering and Computer Science, University of California, Berkeley California.

```
ENVIR.REP
/*
 *  User written C function that creates two mailboxes
 *  and starts a Pascal program with these as standard
 *  input and output. Returns a list with the two ports.
 *
 */


/*
 *  Inclusion of prefix files e.g. global.h that contains
 *  Franz Lisp's type declaration
 */

 include </usr/src/cmd/lisp/franz/h/global.h>
 include </usr/include/vms/dibdef.h>
 include </usr/include/vms/ssdef.h>
 include </usr/include/eunice/eunice.h>
 include <stdio.h>


lispval
crembx()


{ struct {int size; char *ptr; } inbox, outbox, imagename;
  struct {int size; char *ptr; } In_Descr, Out_Descr, logname;
  int Status,  fd1, fd2;
  short int unitnr1, unitnr2;
  int pidnr;
  char mbx_name1[9];
  register char *cp1, *cp2;
  register int i, j;
  FILE *inport, *outport;
  lispval list;
  char mbx_name2[9];

#define CHECK_STATUS  if(!((Status =
#define END_CHECK     ) & 1)) printf("ERROR");

inbox.ptr = "INBOX";
inbox.size = 5;
outbox.ptr = "OUTBOX";
outbox.size = 6;
imagename.ptr = "PASCONTR";
imagename.size = 8;
logname.ptr = "PASCONTR";
logname.size = 8;
```

```
/*
 *  Two mailboxes are created.
 *
 */


fd1 = creat(&inbox,0777,"ipc",256,"tmp",&unitnr1);
fd2 = creat(&outbox,0777,"ipc",256,"tmp",&unitnr2);


/*
 *  The mailbox names are created from the unitnumbers
 *  and entered in the group logical table.
 *
 */

   i = unitnr2;
  cp2 = &mbx_name2[9];
  *cp2-- = '\0';
  *cp2-- = ':';
  while(i) {
           *cp2-- = (i % 10) + '0';
           i /= 10;
           }
  *cp2-- = 'A' ; *cp2-- = 'B' ; *cp2-- = 'M' ; *cp2 = '_';


  Out_Descr.ptr = cp2;
  Out_Descr.size = 9;
  strcpyn(Out_Descr.ptr,cp2,9);

CHECK_STATUS
 sys$crelog(1,&outbox,&Out_Descr,0)
END_CHECK


   j = unitnr1;
  cp1 = &mbx_name1[9];
  *cp1-- = '\0';
  *cp1-- = ':';
  while(j) {
           *cp1-- = (j % 10) + '0';
           j /= 10;
           }
  *cp1-- = 'A' ; *cp1-- = 'B' ; *cp1-- = 'M' ; *cp1 = '_';


  In_Descr.ptr = cp1;
  In_Descr.size = 9;
  strcpyn(In_Descr.ptr,cp1,9);

CHECK_STATUS
 sys$crelog(1,&inbox,&In_Descr,0)
END_CHECK
```

```
 /*
  *   The process PASCONTR is started with the mailboxes
  *   as standard input and output.
  *
  */

 CHECK_STATUS
 sys$creprc(&pidnr,&imagename,&Out_Descr,&In_Descr,&In_Descr,
            0,0,&logname,4,0,0,0)
 END_CHECK


 /*
  *   The mailboxes are connected to UNIX files and opened.
  */

 inport = fdopen(fd1,"r");
 outport = fdopen(fd2,"w");

 /*
  *  The rab structure of outport is changed to allow
  *  asynchronous communication.
  *
  */

 FD_FAB_Pointer[fd2]->rab.rab$l_rop |= RAB$M_TMO;

 /*
  *   The UNIX files are associated with LISP ports
  *   and a list with the ports as car and cdr is
  *   returned.
  */

 ioname[PN(outport)] = (lispval) inewstr(Out_Descr.ptr);
 ioname[PN(inport)] = (lispval) inewstr(In_Descr.ptr);


 list = newdot();
 list->d.car = P(inport);
 list->d.cdr = P(outport);
 return(list);

}
```

```
[inherit('SYS$LIBRARY:STARLET')] program pascontr(input,output);

type Ptr_to_RAB = ^RAB$TYPE;
     unsafefile = [unsafe] file of char;


var  RAB : Ptr_to_RAB;

FUNCTION PAS$RAB( var f:unsafefile) : Ptr_to_RAB;EXTERN;

{ ------------------------------------------------------------------------ }

function MoreInBox(name:packed array [integer] of char):boolean;

{
  Returns true if there are more messages in the mailbox name.
}


var   status:integer;
      dbuff:packed array[1..DIB$K_LENGTH] of char;

begin
status := $GETDEV(DEVNAM:=name,PRIBUF:=dbuff);
if not odd(status) then writeln('MoreInBox:',status);
MoreInBox := ord(dbuff[9]) > 0;
end { MoreInBox };

{ ------------------------------------------------------------------------ }


{ Communication is changed to asynchronous }

begin
RAB:=PAS$RAB(output);
with RAB^ do
  begin
  RAB$V_TMO:=true;
  end;

{ Main loop in PASCONTR }

while true do
  begin
  if moreinbox('SYS$INPUT') then readmess;
  control;
  end;
end.
```

This C written procedure is compiled separately with the command

%cc -c crembx.c

This results in an object file that can be loaded into Franz Lisp on line with the command

> (cfasl 'crembx.o '_crembx 'crembx "function").

A call to crembx will look like.

```
> (crembx)
(%_MBA:1256 . %_MBA:1257)
```