Informes Técnicos Ciemat

755
Junio 1995

Exporting Variables in a Hierarchically Distributed Control System

J.C. Díaz Martín
L. Martínez Laso

Dirección de Tecnología

Toda correspondencia en relación con este trabajo debe dirigirse al Servicio de Información y Documentación, Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas, Ciudad Universitaria, 28040-MADRID, ESPAÑA.

Las solicitudes de ejemplares deben dirigirse a este mismo Servicio.

Los descriptores se han seleccionado del Thesauro del DOE para describir las materias que contiene este informe con vistas a su recuperación. La catalogación se ha hecho utilizando el documento DOE/TIC-4602 (Rev. 1) Descriptive Cataloguing On-Line, y la clasificación de acuerdo con el documento DOE/TIC.4584-R7 Subject Categories and Scope publicados por el Office of Scientific and Technical Information del Departamento de Energía de los Estados Unidos.

Depósito Legal: M-14226-1995
NIPO: 238-95-010-2
ISSN: 0214-087X

Editorial CIEMAT

CLASIFICACIÓN DOE Y DESCRIPTORES

700480
REMOTE SENSING, CONTROL SYSTEMAS COMPUTERIZED CONTROL SYSTEMS, OPTIMIZATION, TOKAMAK DEVICES, COMPUTERS, HELIAC STELLARATORS.

"Exporting Variables in a Hierarchically Distributed Control System"

Díaz Martín, J.C.; Martínez Laso, L.
Advanced Technology Laboratory. CIEMAT.
26 pp. 5 figs. 6 refs.

Abstract

We describe the Remote Variable Access Service (RVAS), a network service developed and used in the distributed control and monitoring system of the TJ-II Heliac, which is under construction at CIEMAT (Madrid, Spain) and devoted to plasma studies in the nuclear fusion field. The architecture of the TJ-II control system consists of one central Sun workstation Sparc 10 and several autonomous subsystems based on VME crates with embedded processors running the OS-9 (V.24) real time operating system. The RVAS service allows state variables in local control processes running in subsystems to be exported to remote processes running in the central control workstation. Thus we extend the concept of exporting of file systems in UNIX machines to variables in processes running in different machines.

"Exportando variables en un sistema de control distribuido jerárquicamente"

Díaz Martín, J.C.; Martínez Laso, L.
Laboratorio de Tecnología Avanzada. CIEMAT.
26.pp. 5 figs. 6 refs.

Resumen

Describimos en este trabajo el servicio de acceso remoto a variables a través de red RVAS, que hemos desarrollado y que es utilizado en el sistema de control y supervisión del Heliac TJ-II, en construcción en el CIEMAT (Madrid, España), dedicado al estudio del plasma en el campo de la fusión nuclear. Este sistema de control está formado por una estación de trabajo Sun Sparc 10 actuando como consola de control central y una serie de subsistemas autónomos 680x0/WMEbus bajo el sistema operativo de tiempo real OS-9 (V.24). El servicio RVAS permite la exportación de variables de procesos de control local en los distintos subsistemas a procesos remotos en la estación central mediante mecanismos de red. De esta forma extendemos el concepto de exportación de sistemas de ficheros al de exportación de variables en procesos remotos en ejecución entre diferentes computadoras.

Exporting Variables in a Hierarchically Distributed Control System

Juan Carlos Díaz Martín and Luis Martínez Laso
Advanced Technology Laboratory. CIEMAT.

Abstract

We describe the Remote Variable Access Service (RVAS), a network service developed and used in the distributed control and monitoring system of the TJ-II Heliac, which is under construction at CIEMAT (Madrid, Spain) and devoted to plasma studies in the nuclear fusion field. The architecture of the TJ-II control system consists of one central Sun workstation Sparc 10 and several autonomous subsystems based on VME crates with embedded processors running the OS-9 (V.24) real time operating system. The RVAS service allows state variables in local control processes running in subsystems to be exported to remote processes running in the central control workstation. Thus we extend the concept of exporting of file systems in UNIX machines to variables in processes running in different machines.

Resumen

Describimos en este trabajo el servicio de acceso remoto a variables a través de red RVAS, que hemos desarrollado y que es utilizado en el sistema de control y supervisión del Heliac TJ-II, en construcción en el CIEMAT (Madrid, España), dedicado al estudio del plasma en el campo de la fusión nuclear. Este sistema de control está formado por una estación de trabajo Sun Sparc 10 actuando como consola de control central y una serie de subsistemas autónomos 680x0/VMEbus bajo el sistema operativo de tiempo real OS-9 (V.24). El servicio RVAS permite la exportación de variables de procesos de control local en los distintos subsistemas a procesos remotos en la estación central mediante mecanismos de red. De esta forma extendemos el concepto de exportación de sistemas de ficheros al de exportación de variables en procesos remotos en ejecución entre diferentes computadoras.

1 Introduction

The TJ-II is a medium size helical axis stellerator with four periods, 1.5 m maximum radius and 1 T nominal magnetic field which is under advanced construction status at CIEMAT (Madrid, Spain) [1]. This flexible device, able to generate a wide range of magnetic configurations, can address a complete scientific programme on the Physics of advanced Stellerators. The control of this machine is carried out by a network of Motorola 68K computers embedded on VME crates,

running under the real time operating system OS-9. This architecture has been considered the best solution for a complet integration and modularity. Each of these machines or subsystems performs the local control of an independent function such as power supply, vacuum pumping, cooling, heating, diagnostics or data acquisition [2]. An ethernet TCP-IP LAN allows the intercommunication for all the involved processors. The configuration of the TJ-II Heliac control system is schematically shown in Fig. 1. Its modular hardware structure can be easily modified and updated.

We have devised a highly structured software for the TJ-II control system. This allows, on one hand, the modularity and flexibility needed in a complex system and, on the other hand, the order that a big interdisciplinar project demands, where different people, tasks and computers must perform as a whole. Developments in network and distributed computing achieved during last years, such as RPC and NFS, have enjoyed a wide acceptance and availability in today operating systems. We have incorporated the RPC services to our software design of the control and monitoring system of the TJ-II as the natural way to get things working together in this physically distributed environment.

Conceptually, we understand the TJ-II control system as a hierarchical structure of machines (S_1, S_2, ...,S_M), processes (P_1, P_2, ..., P_N, ...) running on each of them and the variables (V_1, V_2, ..., V_K, ...) inside these processes as Fig 2 shows. We define the control and monitoring system as a subhierarchy of the whole hierachical structure build only over the control processes and variables that determine the TJ-II machine status and performance. In this way the operator at a central control machine can abstract the control and monitoring system from the local complexities of the different subsystems considering only the subset of variables that are relevant to him.

Processes that run in the central control machine perform the so called *global control*, which involves the remote access to these variables as well as the human interaction to the system through a graphic interface. Variables of *local control* processes in the subsystems have to be readily available to the global control processes, even if global and local processes run on heterogeneous computing enviroments, Sun/SunOS and Motorola 68K/OS9 (V2.4) respectively,

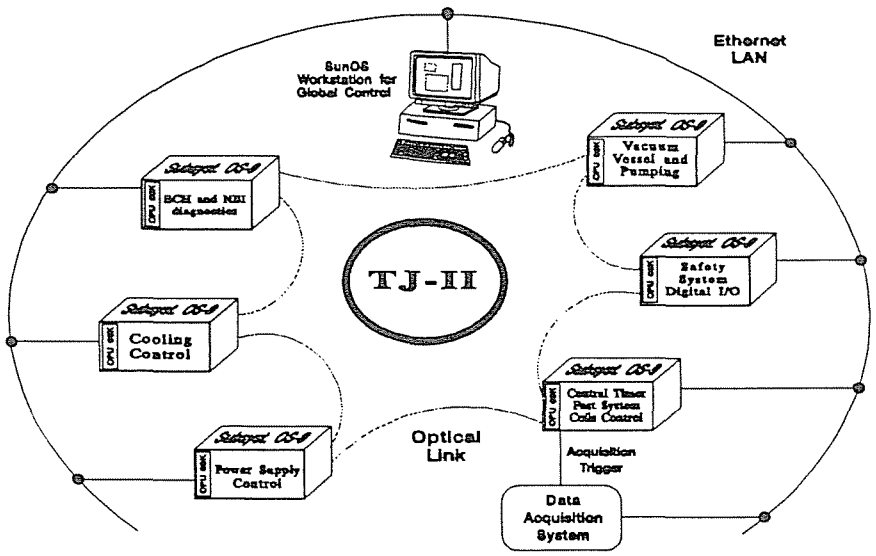

Fig. 1 Configuration of the TJ-II Control System as a set of autonomous subsystems

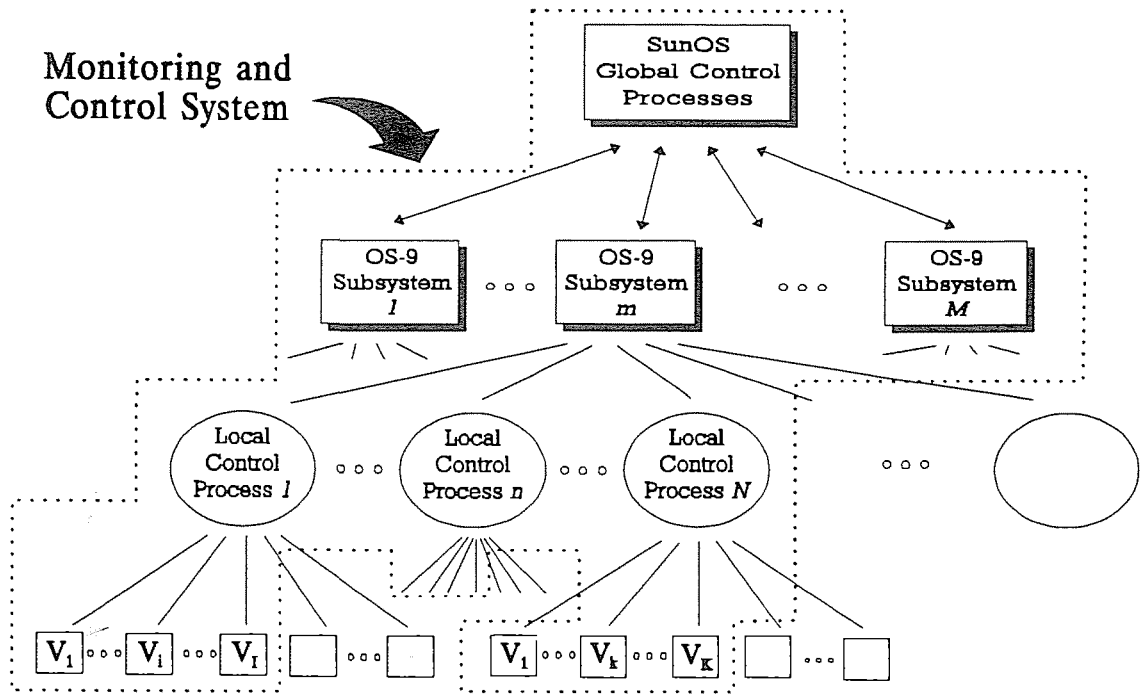

Fig. 2 Monitoring and control hierarchy

so a link between them is needed. The RVAS network service provides such a link. RVAS stands for Remote Variable Access Service and has been built on the concept of *exported variable*. The word exportation has an analog meaning to the one used in distributed file systems jargon, where a server machine exports a file system to client machines, that mount the file system through network services. The client machine sees their mounted files as if they were local ones. NFS is a well known network service that performs this task. To extend the exported filesystem concept to variables in running processes we define an exported variable as the one belonging to the address space of a running process, that is accessible from remote processes through network mechanisms. Under this definition, any process running on a client machine can have two kinds of global variables: the conventional local ones and the remote ones exported to other machines. In our approach, a process accesses its remote variables by issuing a remote procedure call (RPC). This decision obeis to two reasons: firstly, RPC supplies a robust and reliable data transfer and liberates the programmer from machine-dependent data conversion tasks and secondly, we can take advantage of its high-level philosophy of remote resident routine with the aim of developing an extremely compact, clear and elegant RVAS interface in the global control processes. Machines that run exporter processes are named variable servers and machines that run processes accesing to exported variables are named variable client machines. In the the TJ-II control and monitoring system the OS-9 local subsystems play the variable server role. Global processes in the Sun machine are the clients of such servers.

The client-server computing philosophy is also applied to the human interaction to the TJ-II. The X-windows server in the central machine builds a graphic interface that provides to the human operator with a state scenary of the whole TJ-II and a confortable environment in the tasks of parameter setting that involves the control of the TJ-II. Global control processes monitor and control the TJ-II status through the RVAS service and interact the human operator through the X-windows system services. Fig. 3 illustrates the whole architecture of the control system, showing the layout of applications, services and underlying network protocols employed.

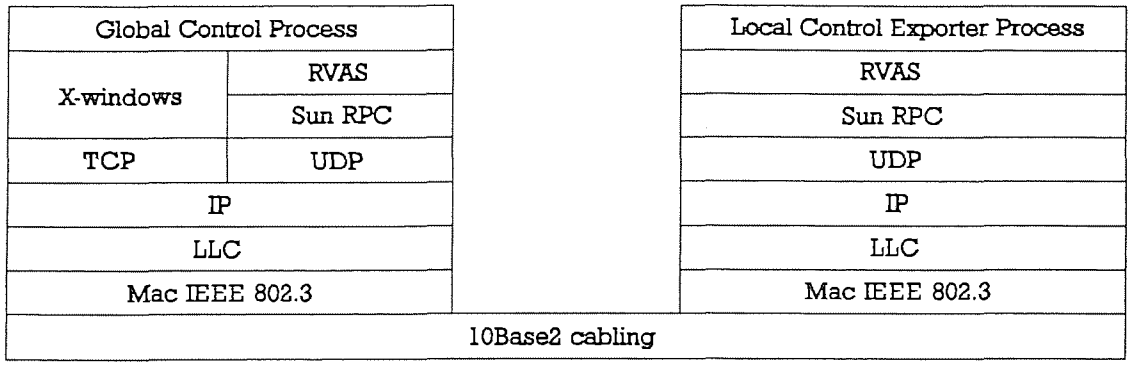

Fig. 3 Layout of the network protocols and services of the TJ-II monitoring and control system.

The different aspects of the RVAS service are described in the following sections. An overview is given next. Third section presents the RVAS elements from the point of view of the user. Information on how to get and install the RVAS software is given in section four. The technical aspects and details on the implementation are described in section five, only interesting for people that need to know some more about everything and enjoying learning how things work inside.

2 RVAS overview

The main objective and the most relevant aspect of this work is to achieve the before mentioned availability of global variables in OS-9 real time local control processes in an efficient way and as transparently as possible to the programmer of these processes. This availability of variables by other processes can be implemented by a common memory module or structure, shared by both the exporter and the client processes of the variables. Any shared structure requires management code (mutual exclusion guarantee, etc) in every process that access to it, in particular the local exporter processes. This code is, however, expensive in terms of time consuming, because two system calls are at least required to use any mutual exclusion mechanisme. Two system calls for each access represents a heavy load for busy local control processes. Moreover, coding the access to shared structures means an added effort to the local control process programmer and a potential source of troubles, always difficult to debug. This two arguments, the strong execution time overhead introduced and the added complexity of the code has moved us to discard the shared module solution and consider other approach where the concept of exported variable plays a central role. It avoids the concurrent access problem and keeps the exporter process simple and fast. The access to an exported variable will be made in the exporter in the usual way, addressing its local memory context, and in the client processes by invoking a service. A daemon centralizes the service requests and serves them in sequential order.

The RVAS service is implemented by three conceptual software elements. The first one is the client application program interface (API) to RVAS, a library of functions built on the low level Sun RPC primitives that ease RVAS use to the programmer. Client programs invoke these functions in order to access remote variables. The second one is a server process called RVAS daemon (rvasd) that deals with client requests. One rvasd daemon runs in each RVAS server

machine. It performs signal and data exchange with the exporter processes involved in the requests in order to serve them. The third element is an exportation library linked to each OS-9 exporter process at compilation time. The exporter library implements the communication between the RVAS daemon and each of the exporter processes.

When a variable of the controlled system has to be read or updated, the global control process involved (client, from the RVAS point of view) invokes the variable exportation service by calls to the RVAS API library. This interface consists of four primitives, named RVAS_GetVar(), RVAS_PutVar(), RVAS_TestActive() and RVAS_PrintError(). The programmer must pass to RVAS_GetVar() the internet name of the machine where the addressed exporter process is running, the identifier of the process in the machine and the identifier of the exported variable in the process. A timeout of return is also given in milliseconds. RVAS_GetVar() will return the value of the variable if everything worked properly. RVAS_PutVar() accept, besides, the new value for the variable involved. The formal definition of the four primitives is given in section 3.

Global control processes access exporter processes in a given OS-9 subsystem through the *rvasd* daemon running in such subsystem. *rvasd* has been implemented as a Sun RPC server. In that way, when a client process issues a RVAS API primitive for accessing an exported variable, the RPC software that implements the call carries the parameters through underlying network protocols and supplies them to the RVAS daemon, that is charged to execute the call: performs a handshake with the involved exporter process in order to read/write the requested variable. The interprocess communication mechanisms provided by OS-9 are used between the exporter process and the RPC daemon to implement such a handshake. Details on this point can be found in fifth section, where a detailed technical description of the RVAS implementation is given.

3 Use of RVAS

On the client side, a process accesses remote variables through a simple C library interface. Though it has been developed on a Sun machine under SunOS, RVAS can run on any machine under any operating system, provided the Sun RPC software is available. On the server side, only the OS-9 (V.24) environment is supported. Of course, the server software, both RPC daemon and exporter processes, must be running before any issue of client requests.

3.1 The client side

Client processes use the functions defined in RVAS API library in order to avoid the direct use of the RPC interface. That keeps client code simple and clear. The definitions of the current API functions are the following ones.

The call to RVAS_GetVar function is :

```
Void    *RVAS_GetVar(Server,ExporterCode,VarCode,TimeOut)
Char    *Server;
int     ExportedCode;
int     VariableCode;
u_int   TimeOut;
```

RVAS_GetVar() takes the internet machine name where the RVAS daemon runs, the code of the exporter process, the code of the target variable in the exporter process and the timeout in

milliseconds. A suitable default value is used when TimeOut is zero. It returns a pointer to a memory buffer where the required variable is stored. On error, Rvas_GetVar() returns NULL and the variable RVAS_errno is set to the error code.

The call to RVAS_PutVar function is :

```
Void    *RVAS_PutVar(Server,ExporterCode,VariableCode,ValuePtr,TimeOut)
Char    *Server;
int     ExportedCode;
int     VariableCode;
void    *ValuePtr;
int     TimeOut;
```

RVAS_PutVar() takes the internet machine name where the RVAS daemon runs, the code of the exporter process, the code of the target variable in the exporter process, a void pointer to the new value for the variable to be written and the timeout in milliseconds. A suitable default value is used when TimeOut is zero. It returns a pointer to a memory buffer with undefined contents. On error, Rvas_PutVar() returns NULL and the variable RVAS_errno is set to the error code.

The call to RVAS_TestActive function is :

```
Void    *RVAS_TestActive(Server,ExporterCode,TimeOut)
Char    *Server;
int     ExportedCode;
u_int   TimeOut;
```

RVAS_TestActive() takes the internet machine name where the RVAS daemon runs, the code of the exporter process and the timeout in milliseconds. A suitable default value is used when TimeOut is zero. It returns a pointer to a boolean value with the result of the check if the RVAS daemon is running or not. On error, Rvas_TestActive() returns NULL and the variable RVAS_errno is set to the error code.

The call to RVAS_PrintError function is :

```
Void    *RVAS_PrintError(Text)
Char    *Text;
```

RVAS_PrintError() takes an ASCII string text and prints it and the error code to the standart error output.

The RVAS software for the clients consists of an #include header file rvas.h and a static library called librvas.a. The procedure and details on how to set up properly the RVAS software are given in the installation section of this document. rvas.h declares the functions of the RVAS API library and defines the error codes returned. An example of RVAS client process is available on the distribution software and another one is shown in appendix A, where two remote variables (of codes 100 and 101) of the exporter process number 0 in the OS-9 machine `gtavml.ciem-at.es` are accessed. To compile any RVAS client process the option `-lrvas` has to be provided to the compiler. More details are given in section four.

3.2 The server side

A template has to be imposed to the source code of any C process to become an exporter. In this section we present the transformation of a general C program that exports several variables in order to fit the RVAS template. Only C-coded OS-9 processes are considered here, though the discussion can easily extended to other languages. Three types of exported variables are currently supported by RVAS, char, int and float. Let us consider the general control program as:

```
#include's
#define's

char        a, b, c, d, e, ... ;
int         i, j, k, l, m, n, ... ;
float       p, q, ..., y, z, ... ;
other_types ...

main()
{
    intercept(InterceptRutine);
    Control_Body();
}

InterceptRutine(signal)
{
    int  signal;

    switch(signal)
    {
        case SIGNAL_1:
            ...
            break;
            ...

        case SIGNAL_n:
            ...
            break;
        default:
            ...
    }
}
```

Let's assume that the variables a, c, k and y are to be exported. The pattern becomes the following one, where bold means added code.

```
#include's
#include    <RVASsignal.h>
#include    <RVAScatcher.h>

#define's

char    b, d, e;
int     i, j, l, m, n;
float   p, q, ..., x, z;
        ...

/* Exported variables, segregated from the rest */
char    a, c;
int     k;
float   y;

int     MyEPC = 0;                  /*    Exporter Process Code                  */
char    *ExportCharPtr = &a;        /*    Address of first char     exported variable  */
int     *ExportIntPtr  = &k;        /*    Address of first int   exported variable  */
float   *ExportFloatPtr= &y;        /*    Address of first float    exported variable  */

    main()
{
    intercept(InterceptRutine);
    RVASbind();
    Control_Body();
    RVASliberate();
```

```
}

    InterceptRutine(signal)
{
    int  signal;

    TheSignal = signal;
    switch(signal)
    {
        case SIGNAL_1:
            break;
            ...

        case SIGNAL_n:
            break;

        case RVAS_BIND_GRANT:
            break;
        case RVAS_BIND_DENY:
            printf("Can't export variables\n");
            break;
        case RVAS_READ_REQUEST:
            ServeRead();
            break;
        case RVAS_WRITE_REQUEST:
            ServeWrite();
            break;
    }
}
```

The most relevant aspect of the change is that the body of the program remains untouched. No instruction or call has been modified, although a segregation of the global variables to be exported is needed. Only five additional elements appear now:

1) Two include files have been added, RVASsignal.h and RVAScatcher.h
2) Four new signal cases have been considered in the signal intercept rutine.
3) Four new variables have been defined. MyEPC, that keeps the code of the process as an exporter one, and ExportCharPtr, ExportIntPtr, ExportFloatPtr three pointers to the three blocks of exported variables, each for a type of variable, all of them for exporter library communication purposes. The pointers are initialized to the address of the first variable of each block.
4) A new sentence, 'TheSignal = signal;', has been included in the signal handler routine.
5) Two new calls, RVASbind() and RVASliberate(), are invoked. Both RVASbind() and RVASliberate() take no arguments and return an integer value: 0 on success and -1 on error.

Every exported variable in an exporter process needs an identification code in order to be referenced in client processes. Char variables codes fall in [0-99] range, int variables in [100-199] range and float variables in [200-299] range. Consecutive variables of a given type are given consecutive codes. The variable a in the example is assigned the code 0, c is assigned the code 1, k the code 100 and y the code 200. Appendix B shows an example of an exporter process. It holds the remote variables accessed by the RVAS client program given in appendix A. An exporter process is compiled by linking the exportation library. The OS-9 link step of the compilation process is rather unfriendly, therefore a makefile is provided with the installation software to ease things. This makefile is supplied in appendix C.

4 RVAS distribution and installation

The RVAS client and server software is public and distributed in the compressed files rvasSvrOs9.tar.Z and rvasCltSunOS.tar.Z, available in the /home/pub/rvas directory in the internet

machine `gtasn3.ciemat.es` (130.206.11.19) and accesible via anonymous FTP. The creation of a RVAS directory from where invoke FTP in both client and server machines is recommended. In these files, the headers, libraries and makefiles are included for the OS-9 (server) and SunOS (client) operating systems.

The server software is available in the file rvasSvrOS9.tar, in Motorola 680x0 binary format for OS-9. The ytar program is provided for untaring it with the commands sequence:

```
Prompt% cd ... /RVAS
Prompt% load -d ytar
Prompt% ytar x rvasSvrOS9.tar
```

The file rvasSvrOS9.tar is exploded as follows:

```
EXAMPLE
     SOURCE
          aplicontrol.c
     BIN
     RELS
     makefile
EXPORTLIB
     LIB
          exportlib.l
SERVER
     BIN
          rvasd
```

The RVAS daemon must be up and running before the first bind request from any exporter process. Once installed, the following procedure must be executed to start the RVAS daemon:

```
Prompt> chd ... /RVAS/SERVER/BIN
Prompt> load -d rsvasd
Prompt> rvasd&
```

The inclusion of these commands in the startup file is recommended to get rvasd always ready and the RVAS service available. The file aplicontrol.c is an example of a simple exporter process that exports two integer variables. The makefile compiles it and links the exportlib.l library. It allows the test of the RVAS facilities. This is the command sequence that runs it.

```
Prompt> chd ... /RVAS/EXAMPLE
Prompt> make
Prompt> chd BIN
Prompt> load -d aplicontrol
Prompt> aplicontrol&
```

On the client side, rvasCltSunOS.tar.Z contains four files: the C source code of the RVAS API library for Unix machines, a C source code RVAS client example program, a header file for RVAS clients and a readme file, where directives to install the RVAS API library and to compile RVAS client applications can be found.The file rvasClntSunOS.tar is exploded as follows:

```
librvas.c
XviewClientExample.c
readme
rvas.h
Makefile
DOC
```

The following procedure must be executed to install rvas.h in the include directory of the system in the SunOS machine.

```
Prompt% cd ... /RVAS
Prompt% make all
Prompt% su
     Password:
Prompt{root}% make install
Prompt{root}% exit
Prompt%
```

Our RVAS client example, XviewClientExample.c, also uses the XView set of widgets, so three more libraries have to be included in the compilation command:

```
Prompt% cc XviewClientExample.c -I/usr/local/include -L/usr/local/lib
-lrvas -lxview -lolgx -lX11
```

The DOC directory keeps a WordPerfect 6.0 formatted copy of this RVAS tutorial document. Once everything runs properly, you can enjoy RVAS. Any comment or suggestion will be welcome to `jdiaz@gtasn3.ciemat.es` or `laso@dec.ciemat.es`.

5 RVAS implementation

In this document we have presented till now the information needed for any user to include RVAS in his programs. An API to RVAS in the client machine and an exportation library in the server machine have been provided to prevent the user from the complexities of the explicit use of the RPC low level calls and the OS-9 interprocess communication facilities. Technical details on the internal structure of those libraries and the RVAS daemon are described in this section. Figure 4 shows a scheme of the different elements and communication mechanisms used in our application.

5.1 The RVAS API

The RVAS API library is built on the low level Sun RPC services [3]. Our application is distributed between a SunOS operating system running on the Sun SPARC architecture and the real time OS-9 operating system running on the 68K Motorola microprocessors. Sun RPC, available on both systems, hides this heterogeneity providing a system independent end-to-end data path between the parts. Sun RPC services have a three layer architecture. We use the lowest one, that offers time monitoring on the RPC calls execution, allowing the programmer to set a timeout that gives the maximum return deadline of each RPC call. This characteristic is more than adequate in the real time distributed environment were the RVAS is to be used. The RPC low level interface is however rather complex and intrincate, therefore an straight programming interface to RVAS has been developed.

5.2 Inter-process communication in OS-9

The communication between the RVAS daemon and the exporter process, using the exporting library, has been implemented "ad hoc" making use of four available process intercommunication facilities in OS-9: signals, shared data modules, events and named pipes, also

called FIFO's in the Unix literature [5]. In each of the RVAS server machines, the communication between the daemon and the exporters is carried out through a signal sending protocol. These signals are defined in the export.h file, which is included in the daemon and exporter source code. An exporter process sends a signal to the server in order to establish connection and another one to disconnect from the exportation service. The daemon confirms or denies the connection by means of another signal. When a RVAS client process request accesses to an exported variable, the request gets first to the daemon and then it issues a signal to notify such request to the implied exporter process. An OS-9 shared data module is a buffer of memory managed by the system, shared by two or more processes and known by a public name. Two data modules are shared by the daemon and the exporter processes in RVAS. Their names are: Pids and RPCparameters. In OS-9 V2.4, when a process receives a signal, it has no way to know the process identification number (pid) of the sender, therefore, no reply is possible. We make use of the data module Pids in order to avoid this problem, making the signal sender process to leave there its pid. The Pids module is a three integer structure which format is:

```
typedef struct {
    int PidServer;
    int PidExporter;
    int NPExporter;
}StructPids;
```

When the RVAS daemon starts up, it leaves its pid in the PidServer field so that it becomes available to any exporter process willing to send signals to it. Any exporter process must leave its pid in the PidExporter field before sending the connection signal RVAS_BIND_REQUEST to the RVAS daemon. Once received, the daemon reads the PidExporter field in the shared module Pids in order to find out the process asking for binding and grants it by sending to it the RVAS_BIND_GRANT signal. The exporter sleeps after sending the signal RVAS_BIND_REQUEST waiting the RVAS_BIND_GRANT signal from the RVAS daemon.

Let us consider the case when a second exporter process gets a CPU time slice after the pid of the first exporter process has been written to the shared module, but before the RVAS_BIND_REQUEST signal had been issued. Let us assume this second exporter also tries to

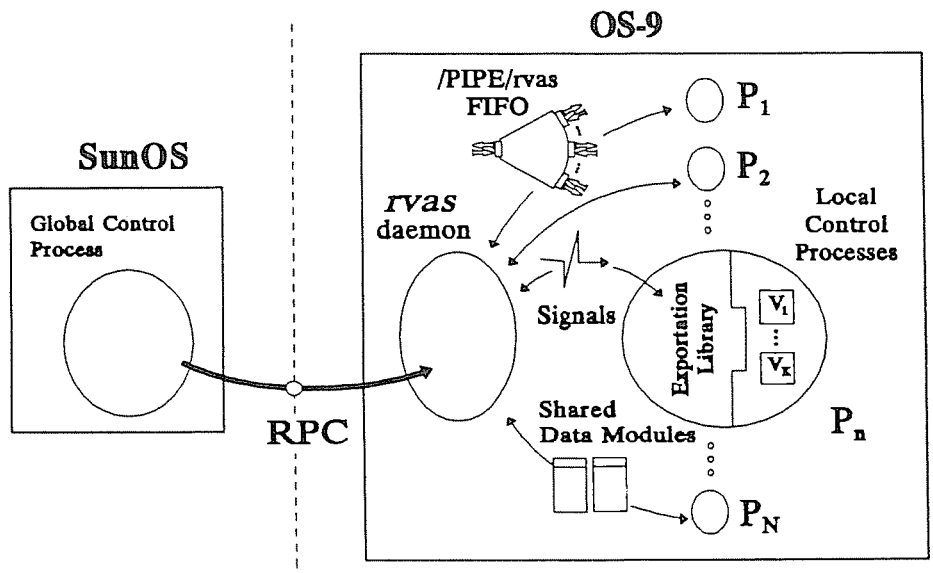

Fig. 4 Diagram of OS-9 inter-process communication mechanismes used.

establish connection with the RVAS daemon. It overwrites the Pid of the first exporter and send the RVAS_BIND_REQUEST signal. At this moment, two signal are enqueued at the daemon process descriptor. When dispatched, the second exporter process will be replayed twice and the first of them will wait forever. In order to avoid this problem, a semaphore has been implemented over the named GatePids OS-9 event and the bind/liberate code has been packed in a critical region guarded by this semaphore. An OS-9 event is a public long integer variable to which any process can link for reading, writting, or waiting for a variable to be in a given range of values.

We have already mentioned that the SunOS client process reads/writes a remote variable by issuing a RPC call of two/three parameters: the code of the exporter process, the code of the exported variable inside the process and, on writting, the new variable value. The pertinent RVAS daemon receives the RPC call and redirects it to the exporter process given as the first parameter. This task is performed in three steps. In the first place, the RPC server daemon accesses the RPCparameters shared data module and copies there the parameters of the call been serviced. The RPCparameters module is described as the three integer structure given by:

```
typedef struct {
        int  Process;
        int  Variable;
        int  NewValue;
} StructRPCparameters;
```

It is not needed to consider here a mutual exclusion access mechanism due to the iterative nature of the RPC server, that does not serve a call until it has ended with the previous one [5]. Once the parameters copy is made in the RPCparameters data module, the second step is performed, by sending the signal RVAS_READ_REQUEST (read a variable) or RVAS_WRITE_REQUEST (write a variable) to the exporter process. The last step is to reply to the RVAS client.

A named pipe is an interprocess communication mechanism that generalizes the concept of pipe. It allows a multidirectional data flow between processes in a first byte written, first byte read basis. The reading or writting of an exported variable from a remote client generates a traffic of bytes between exporter process and the RVAS daemon. This data flow is handled by the named pipe or FIFO /PIPE/rvas and its function is schematically presented in Fig. 5. Two descriptors, one for reading and another one for writing, are required per OS-9 process using the named pipe. Each exporter process opens its descriptors to /PIPE/rvas in the before mentioned RVASbind() procedure.

5.3 The RVAS server daemon

Any server respond, in general terms, to same execution pattern. It waits for requests, serves them and goes back to wait. The execution loop of a RPC server daemon is as follows. It waits RPC requests from a client in an UDP socket, blocked in the select() call in the OS-9 RPC library function scv_run(). Eventually, the data ready condition arrives in any of the sockets managed by RPC. Then the daemon awakes and executes the required procedure. The RVAS daemon, however, not only dispatch RPC clients, but also the bind/liberate requests from OS-9 exporter processes. Unlike the RPC requests, that are managed by the select() system call, the exporter petitions are implemented by means of signals. If any signal arrives from an exporter process while the deamon is in select(), the select() call is aborted and the signal intercept routine is inmediatelly run. After it returns, the execution goes on in the instruction following the interrupted select() call, the svc_run() loop makes the RVAS daemon issue a select() call again.

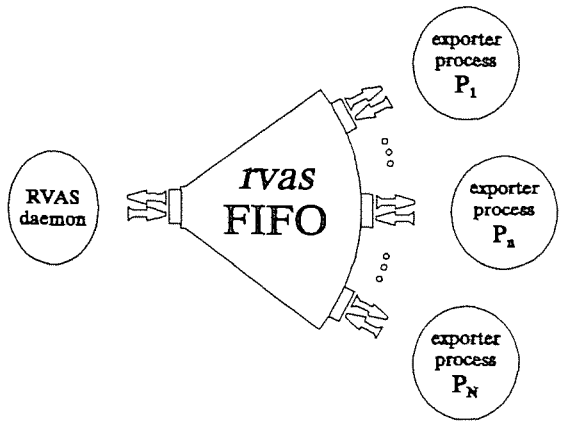

Fig. 5 The RPC daemon and exporter processes communicate exported variables through the /PIPE/rvas FIFO.

The source code of scv_run() can be found in reference [6]. The RVAS daemon is the link between clients and exporters. In what follows, we will show how it is known by both, how it registers the exporters that clients can contact with and, at last, how clients requests are managed and redirected to the implied exporter.

The RVAS daemon is an RPC server and, as such, it has to be known by a public identificator. Known this key, any local or remote process may become client of the service paid by RVAS. The identificator consists of two integers, called program number and version number, ([555552, 13] in our case). The RPC software is in charge of translating this public identificator to a suitable UDP port in order to interface the UDP/IP protocol stack on which Sun RPC is built. In that way, client processes are unaware of UDP port numbers. Like the RPC daemon, as a service provider, is known to clients by a public identifier, it, as an OS-9 process, must be known by every exporter process in the machine. Logically, this local identifier is its Process Identification Number (pid). Publication() is the first routine called in the server daemon source code. It creates the shared data module Pids and puts there its pid, where it will always be available to any process willing to bind it in order to export variables. Pids also keeps the pid and code of the exporter process that is currently trying to bind to the daemon. The event GatePids is also created in Publication() to synchronize simultaneous bind procedures from exporter processes.

The RVAS daemon intercepts two signals, both from exporter processes when they start and stop execution respectively: RVAS_BIND_REQUEST and RVAS_LIBERATE_REQUEST. Two routines, ServeBind() and ServeLiberate(), handle them respectively. Let's assume that a writing request comes. It carries three parameters: the code of the exporter process (EPC), the code of the exported variable to be addressed and the new value to be written. Once the daemon gets the parameters, it needs to know the pid of the exporter in order to send a RVAS_WRITE_REQUEST signal to it, therefore a translation from EPC to pid is needed. The table EpcToPid[] does this mapping. As is easy to understand, the routines ServeBind() and ServeLiberate() are the ones that manage EpcToPid[]. The whole story is as follows: When any process that exported variables begins to execute, it fetches the pid of the RVAS daemon from the Pids shared data module and, after that, it gets blocked at the semaphore on the event GatePids, waiting for it to be released. Then it enters a critical region where it leaves its pid in the Pids shared data module, send the RVAS_BIND_REQUEST signal to the RVAS

daemon and goes to sleep. The daemon, now surely blocked in an UDP socket, catch the signal and enters in its intercept ServeBind() routine. ServeBind() reads the pair (pid, EPC) from Pids and creates an entry in EpcToPid[]: The entry given by EPC will contains the pid. Once the entry is created, the slept process that sent the RVAS_BIND_REQUEST signal becomes registered as an exporter, the signal RVAS_BIND_GRANT is sent to it and ServeBind() returns. Any signal wakes an OS-9 process, so that RVAS_BIND_GRANT makes the slept process to resume the execution, exits the critical region and, once registered as exported, begins its control task.

A Sun RPC server can handle one or more procedures. Each of them are identified by an unsigned integer. Our RVAS server daemon supports three procedures: RPCread(), RPCwrite() and RPCactives(), with, respectively, 1, 2 and 3 procedure codes. RPCread() and RPCwrite() use the EpcToPid[] table to get the pid of the exporter process whose EPC comes as a parameter of the received RPC call. Later, they put the parameters in the shared data module RPCparameters in order to make them available to the target exporter process and, finally, send a signal to it, either RVAS_READ_REQUEST or RVAS_WRITE_REQUEST respectively. On reading, the daemon gets blocked in the named pipe /PIPE/rvas waiting for the value of the required variable. Once read, RPC software is in charge of making this value as the return value of the RPC call. On writting, the daemon just writes the new value to /PIPE/rvas. A void value is returned by both routines. RPCactives() has a management role, reporting clients about the exporter processes connected to RVAS.

5.4 The RVAS exporter process

It is needless to say that any just started exporter process must let the RVAS daemon know about its presence in order to link to RVAS client processes, therefore some kind of binding procedure must be executed by the exporter. The exporter process must also inform the daemon before exiting. The exported code should run unmodified between both calls: once binded, it enters its own code and runs as expected by its programmer, unaware of the fact that his address space will be invaded by the RVAS daemon. Multitasking oriented CPUs protect the adressing context of the processes by hardware fences, generally implemented in memory management units. RVAS breaks the fence that guards the exporter process addressing context by the resource provided by the OS-9 operating system: the signal system call. The RVAS daemon will signal the exporter process whenever it be required to do it. Upon reception, the busy exporter will momentary put its work apart and will perform on behalf of the RVAS daemon. After this short interrupt, things go on as if nothing were happened. Perhaps a global variable will register a new value. This discussion takes us to a software glue that consists basically of two elements, on one hand, a pair of procedures, RVASbind() and RVASliberate(), that respectively set up and liberate the bind to the RVAS daemon and, on the other hand, a pair of signal catching routines, ServeRead() and ServeWrite(), that dispatch the requests from the RVAS daemon once the binding procedure runs successfully. These routines have been implemented in an object library that we name the *exportation library*. The details on the internal structure of these functions are presented in section 5.5.

5.5 The RVAS exportation library

As mentioned above, the role of the exportation library is to make a global variable available to other process, i.e., to export it, and to hide to the programmer of the exporter process the details that implement this new attribute (exported) of some of his variables. A high

degree of transparency has been achieved up to date, but still the programmer has to observe some syntax rules about the order on which the exported variables are declared and other related questions that were addressed in section 3.2. Work is in progress in order to simplify even more the bind to RVAS in new versions. The exportation library is a four entry points linkable module. The entries are RVASbind(), RVASliberate(), ServeRead() and ServeWrite();

We describe in this section some details on how these functions are implemented. The application programmer explicitily invokes the two first ones, while is unaware of the last two ones, called in signal catching routines. When an exporter process wants to bind to the RVAS daemon, it invokes the function RVASbind(). The steps to perform this connection are the following: first, the exporter process writes its pid in the field PidExporter in the shared data module Pids, second, it sends the signal RVAS_BIND_REQUEST to the daemon asking for connection and, third, waits for the signal RVAS_BIND_GRANT that grants the connection. These three actions must be carried out in an atomic fashion and this is assured by the use of the event GatePids. It implements a semaphore that regulates the signal interchange between the RVAS daemon and the exporters as above explained in section 5.2. RVASliberate() performs the tasks needed to release the connection of the exporter process to the RVAS daemon. The ServeRead() and ServeWrite() routines allow the exporter process to catch the signals RVAS_READ_REQUEST and RVAS_WRITE_REQUEST issued by the RVAS daemon when required by a RVAS client process. The ServeRead() and ServeWrite() catching routines have a minimum impact on the exporter application performance. ServeRead() reads from the shared data module RPCparameters the code of the addressed exported variable. The type of the variable, char, int or float, is determined from the code of the exported variable. The exportation library keeps a pointer per type, each pointing to the first variable of each type. The pointer and the order number determine the address of the exported variable. The variable is read from the /PIPE/rvas and the intercept routine returns to control code execution. The computing load that the intercept routine represent to the exporter process is practically reduced to a system call. ServeWrite() intercepts the signal RVAS_WRITE_REQUEST and the same considerations apply here.

6 Acknowledgments

The authors are particularly indebted to Luis Pacios Rodríguez and his team, who rely on us to help on the software aspects of the TJ-II control system. We thank Pedro Olmos Moreno for his support and his careful read of the manuscript. Our special thanks to Antonio Muñoz Roldán, for his valuable suggestions on software installation and his always useful comments. In addition, we thank José Carlos Díaz Carrero for his frienship and his infinite patience on our questions on UNIX and the X window system.

7 References

[1] Alejandre, C. *et al.* "TJ-II Proyect: a Flexible Helliac Stellerator", Fusion Technology, Vol. 17, No. 1, p. 131, January 1990.

[2] Pacios, L. *et al.* "Control System for the TJ-II Helliac", Proceedings of the RT93 Conference, Vancouver, Canada, 1993.

[3] Corbin, J.C., "The Art of Distributed Aplications", Springer-Verlag, 1991.

[4] Dayan, P.S., "The OS-9 Guru. 1 - The Facts", Galactic Industrial Limited, Mountjoy Research Center, Stockton Road, Durham, DH1 3UR, United Kingdom, 1992.

[5] Stevens, W. R., "Unix Network Programming", Prentice-Hall Software Series, Prentice-Hall, 1990.

[6] Microware, "The OS-9/OS-9000 Network File System/Remote Procedure Call User's Manual", Microware Systems Corporation, 1900 N.W. 114th Street, Des Moines, Iowa 50325-7077. 1992

Appendix A

```
1   /*
2   /*  Ver.     Date          Remarks                                                    Author
3   /*  -----    --------      -------------------------------------------------------    ---------------
4   /*  1.0      02/02/94      This is an example of RVAS client program. It makes use of the    Juan Carlos Diaz Martin
5   /*                         RVAS API calls.  in order to check the exporter process that are
6   /*                         running in the machine "gtevm1", to read a remote variable in the
7   /*                         exporter process of EPC 0 and set a second variable in this same
8   /*                         exporter process.
9   */
10  #include <stdio.h>
11  #include <rvas/rves.h>
12
13  /*-------------------------------------------------------------------------------
14  */
15      main()
16  {
17      Actives();
18      EvenSpy();
19      EvenRangeChange();
20  }
21
22  /*-------------------------------------------------------------------------------
23  */
24      Actives()
25  {
26      int             index;
27      int             *ResultPtr;
28
29      printf("\nPids  of Active OS-9 control processes:\n");
30      printf("-----------------------------------------\n");
31      for(index = 0; index < 10; index++)
32      {
33          if( NULL == (ResultPtr = RVAS_TestActive("gtavm1", index, 200)) )
34          {
35              RVAS_PrintError("Actives()");
36              return;
37          }
38          if(0 != *ResultPtr)
39              printf("Exporter %d is active, PID %d\n", index, *ResultPtr);
40          else
41              printf("Exporter %d is not active\n", index);
42      }
43  }
44
45  /*-------------------------------------------------------------------------------
46  */
47      EvenSpy()
48  {
49      int         *ResultPtr;
50
51      /*
52      /*  Ask for the value of the integer variable 100, in the exporter process 0 in the subsystem "gtavm1"
53      */
54      if(NULL == (ResultPtr = RVAS_GetVar("gtavm1", 0, 100, 600)))
55      {
56          RVAS_PrintError("EvenSpy()");
57          return;
58      }
59      pritf("gtavm1, EPC 0,  variable 100: %d\n", *ResultPtr);
60      return;
61  }
62
63  /*-------------------------------------------------------------------------------
64  */
65      EvenRangeChange()
66  {
67      void            *ResultPtr;
68      static int      time = 0;
69
70      time++;
71      /*
72      /*  Write the value of the integer variable 101, in the exporter process 0 in the subsystem "gtavm1"
73      */
74      if(NULL == (ResultPtr = RVAS_PutVar("gtavm1", 0, 101, (void*)&time, 600)))
75      {
76          RVAS_PrintError("EvenRangeChange()");
77          return;
78      }
79  }
```

Appendix *B*

```
1   /*
2   /*  Ver.    Date            Remarks                                                    Author
3   /*  ----    ------          ---------------------------------------------------------  ------------
4   /*  1.0     05/10/93        This program updates the integer variable 'EvenCounter' that acts   Juan Carlos Díaz Martín
5   /*                          as a circular counter. The also exported variable 'Range' drives the
6   /*                          range of counting. RVAS client process can read the value of the
7   /*                          cunter and set up the range of conting.
8   */
9   #include <stdio.h>
10  #include <RVASsignal.h>
11  #include <RVAScatcher.h>
12
13  /*-------------------------------------------------------------------
14  /*
15  /*  EXPORTER PROCESS CODE (EPC)
16  */
17  int     MyNPE = 0;          /*      Setting of the EPC      */
18
19  /*-------------------------------------------------------------------
20  /*
21  /*  Declaration of exported variables
22  /*
23  /*  Code        Name            Access Type
24  /*  ----        ------          -----------
25  /*  100             EvenCounter     R
26  /*  101             Range           R/W
27  */
28
29  /*-------------------------------------------------------------------
30  /*
31  /*  int type exported variables
32  */
33  int     EvenCounter = 0;
34  int     Range = 0;
35  int     *ExportIntPtr = &EvenCounter;
36
37  /*-------------------------------------------------------------------
38  /*
39  /*  Declaration of functions
40  */
41  int  MyInterceptRoutine();
42  void    ControlBody();
43
44  /*-------------------------------------------------------------------
45  /*
46  /*  Signal intercept routine
47  */
48  int  MyInterceptRoutine(signal)
49  int  signal;
50  {
51      TheSignal = signal;
52      switch(signal)
53      {
54          case 2:  /* Control-E */
55          case 3:  /* Control-C */
56              RVASliberate();
57              exit(signal);
58          case RVAS_BIND_GRANT:
59              break;
60          case RVAS_BIND_DENY:
61              printf("Can't export variables\n");
62              break;
63          case RVAS_READ_REQUEST:
64              ServeRead();
65              break;
66          case RVAS_WRITE_REQUEST:
67              ServeWrite();
68              break;
69          default:
70              return(-1);
71      }
72      return(0);
73  }
```

```
/*-------------------------------------------------------------------
/*
/*  Control Code. The 'EverCounter' variable acts as a circular counter. The variable 'Range' determines
/*  the [0-248] or [250-500] range of counting.
*/
void    ControlBody()
{
        while(1)
    {
        EvenCounter + = 2;
        switch(Range)
        {
            case 0:
                if(EvenCounter > 250)
                    EvenCounter - = 250;
                if(EvenCounter = = 248)
                    EvenCounter = 0;
                break;
            case 1:
                if(EvenCounter < = 250)
                    EvenCounter + = 250;
                if(EvenCounter = = 500)
                    EvenCounter = 250;
                break;
            default:
                ;
        }
    }
}

/*-------------------------------------------------------------------
/*
/*  Main Program
*/
main()
{
    Intercept(MyInterceptRoutine);
    switch(RVASbind())
    {
        case 0:
            printf("Successful binding to RVAS\n");
            break;
        case -1:
            printf("Couldn't bind to RVAS\n");
            break;
        default:
            printf("Unknown error returned from RVASbind()\n");
            break;
    }
    ControlBody();
    switch(RVASliberate())
    {
        case 0:
            printf("Successful liberation from RVAS\n");
            break;
        case -1:
            printf("Couldn't liberate from RVAS.\n");
            break;
        default:
            printf("RVASliberate returned unknown error\n");
            exit(0);
    }
}
```

Appendix *C*

```
1   #   Makefile OS-9 (V24)
2   #
3   #   Date        Remarks                                                    Author
4   #   ----------  ---------------------------------------------------------  ---------------------
5   #   06/10/93    Compilation of aplicontrol.c, an example of an OS-9 exporter   Juan Carlos Diaz Martin
6   #               program. The exportation library exportlib.l is linked to it.
7   #
8   DD =            /h0
9   TEMP =          /r0
10  ODIR =          ./BIN
11  RDIR =          ./RELS
12  SDIR =          ./SOURCE
13  EXPTDIR =       .
14  EXPTSDIR =      ../EXPORTLIB/SOURCE
15  EXPTRDIR =      ../EXPORTLIB/RELS
16  EXPTLDIR =      ../EXPORTLIB/LIB
17  RPCLIB =        $(DD)/nfs/lib/rpclib.l
18  RPCDEFS =       $(DD)/nfs/defs
19  INETDEFS =      $(DD)/isp/defs
20  MISDEFS =       ../DEFS
21  INETLIB =       $(DD)/isp/lib
22  CFLAGS =        -t=$(TEMP) -v=$(DD)/defs -v=$(INETDEFS) -v=$(RPCDEFS) -v=$(MISDEFS)
23  COMMLIB =       -l=$(RPCLIB) -l=$(INETLIB)/netdb.l -l=$(INETLIB)/socklib.l
24  #
25  #
26  $(ODIR)/aplicontrol:    $(RDIR)/aplicontrol.r $(EXPTLDIR)/exportlib.l
27      l68 /DD/LIB/cstart.r $(RDIR)/aplicontrol.r \
28              -l=$(EXPTLDIR)/exportlib.l \
29              $(COMMLIB) \
30              -l=/DD/LIB/clib.l -l=/DD/LIB/sys.l \
31              -O=$(ODIR)/aplicontrol
32
33  $(RDIR)/aplicontrol.r:  $(SDIR)/aplicontrol.c
34      cc $(CFLAGS) $(SDIR)/aplicontrol.c  -r=$(RDIR)
```