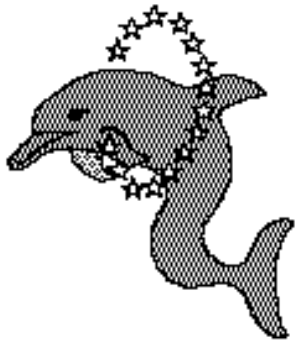


DIM
Delphi Information Management System

C. Gaspar, ECP Division, CERN Geneva
M. Dönszelmann, ECP Division, CERN Geneva

This manual gives a description of the Delphi Information Management system - DIM. The DIM allows User Interfaces and Control processes to access information from any of the 20 nodes of the Delphi Online System in a transparent way. Using DIM, Delphi User Interfaces can be run anywhere in the world providing that DECNET or TCP-IP are available.

Revision/Update Information: Version 2.0, Jan 1992

1 Introduction

The Online System of Delphi runs distributed over around 20 VAX systems and 70 FASTBUS OS9 processors. All these nodes produce different types of information that have to be gathered in order to be displayed or monitored.

To facilitate the transfer of information, an information management system was designed (DIM, Delphi Information Management). This system will transfer information, such as smi states, trigger rates, hv status etc, to all the destinations that are interested in it. This is called multicasting (as opposed to broadcasting, where the information from one source goes everywhere).

DIM is based on a client-server model. Servers will provide their information as a set of services. Once installed, these services can than be requested by the clients. A number of different requests are possible, like polling, periodic-updating, or monitored-updating.

Although information flows directly between a server and one or more clients, there is one central server that keeps track of what information is available and where it is available. Such a server is called a Name Server.

DIM is made fully recoverable. This means that if one of the parts (client, server, name server) goes down or crashes, when it restarts the system will get all its information back again.

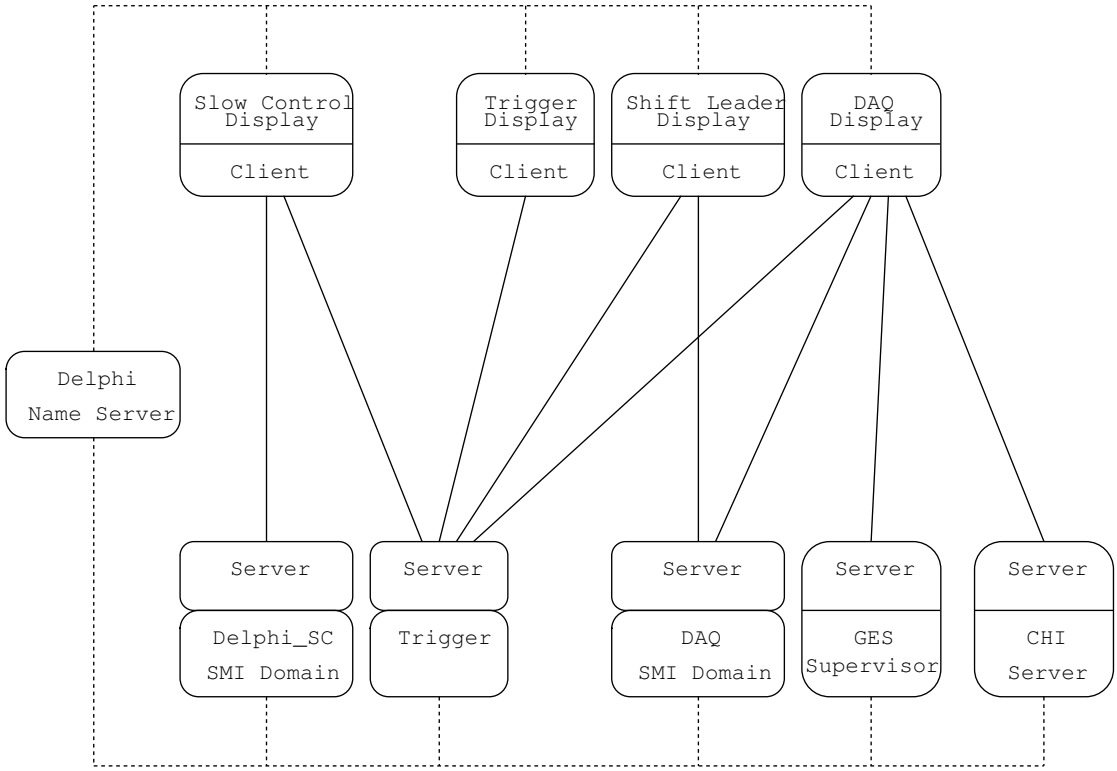
Since most information within Delphi is available in local memory sections, a general server is provided, that maps these sections and provides the information as services. This general server is fully configurable.

The DIM system is very easy to integrate. Any process wanting to became a server issues a call to the DIM package and all it's information will be available.

2 Information Distribution System

The mechanism of an information distribution system can best be explained by an example, see figure Figure 2–1. The system consists of a set of servers and clients, that can be point-to-point connected with each other. There is one special server, the name server, which has (temporary) connections with all servers and clients. The name server works like a telephone dictionary. It keeps track of where servers reside and what they have to offer.

Figure 2–1 Example of an Information Distribution System



A server registrates within the name server at startup. It declares its available information as named services. A client inquires the name server, what information is available and where to get it. Knowing the place where to get the information, it will make a direct connection to the server (no name server needed anymore).

More than one client can registrate an interest in a certain service. The server itself keeps track of who is interested, and hence sends the information only to these clients. This will keep the flow of information to the necessary minimum.

The DIM system tries to pack as much packets as possible. This means that information from different services, but from the same server, will be send together to one client.

The order in which different parts (server, clients, name server) of the DIM system are started is not important. All programs are designed to recover after crashing. In some cases the system might recover only slowly (see implementation).

Clients are linked with the client package. The interface to the DIM system on the client side is part of the client program.

Servers are foreseen in two different configurations. The example shows that the trigger runs as a stand-alone program, creating a global section. A server, connected to this global section, runs as a separate process. The GES supervisor however incorporates the server functionality in its own program.

In the following paragraphs the functionality of the name server, server and client are described in more detail.

2.1 The Name Server

The name server functions as a telephone dictionary, except that it holds different and more information. It searches items by name. A number of attributes can be associated with every item. Each attribute has a typed value.

The name space, as well as the attribute name space, is fully free, except for any name starting with "/DNS/" or "DNS.", since these are used to keep statistics (see below). Names could look like "/id/smi_sc/status" or "trigger.status".

The name server can respond to the client with either values of different attributes of a certain service or a list of services can be provided that conforms certain criteria, like <"node" equals VXDEOP>.

As the name server starts, its tables are empty. They are filled by servers informing the name server about their services. A double declaration of a certain service is not allowed, unless the server who first declared this service is not available anymore. In case of double declaration, the server will be notified by an "error" signal, upon which this server has to die.

If a client inquires about a service that does not exist yet, the name server will registrate the request. It will then, as soon as the service is available, contact all interested clients to provide them with the necessary information.

The name server is itself also a normal server. It offers a set of statistics on communications. These statistics are gathered once in a while from all the servers and are generally available.

If the nameserver dies while other servers are running, it will, after it started again, behave as a slow name server. All servers will reestablish contact and provide information to the name server again. Also all clients, that did not get a response yet (due to a non-available service for instance), will recontact the name server again.

2.2 The Server

A server will contact the name server, to declare its services. It regularly keeps in contact with the name server to inform it that its services are still available and to give some statistics.

The server exits with a fatal error in case it tries to declare a service that is already available with another server.

The server is contacted by clients to send information. If this connection is lost, the server does nothing. It just waits until the client reconnects.

If the server dies it tries to inform the name server that its services are no longer available. If this does not work (because of a node crash), the name server will know eventually, because the regular statistics will not come in. After the server recovered, everything works as normal again.

2.3 The Client

A client (linked in with the information program) has the ability to search for different types of information in the name server. If a client knows the service it is interested in, it will ask the name server how to contact the server. It then makes a connection to the server and requests for the information. Depending upon the implementation of the server program, this can be once only information, regular updates or updates based on monitoring.

If the client does not succeed to get in touch with the name server it will keep on retrying until the name server is up.

The client sends the request to the name server and does not wait for an answer, the reply might came immediately if the service is available or later as soon as it becames available. On reception of the reply from the name server the client will directly contact the server. If the client does not succeed in calling the server, it will call the name server again.

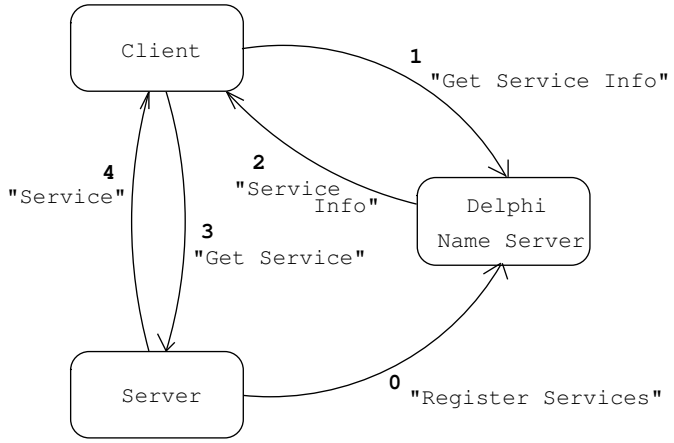
In all cases the client will succeed in calling for the information, but the actual update/information might come later (asynchronously).

If the client-server link goes down, the client will try to reestablish it once. If it does not succeed, a call for information is done with the name server, to find out if the server died, or if it has moved to another node. A regular answer can be expected from the name server or a declaration of interest is queued.

If the client crashes, it tries to cancel its requests with both the server(s) and the name server. If it does not succeed, both will get to know anyway, since the links will be broken. After restarting it will redeclare its interest in the different services.

The figure Figure 2–2 shows the data flow between a server, a client and the name server

Figure 2–2 Dim connections diagram

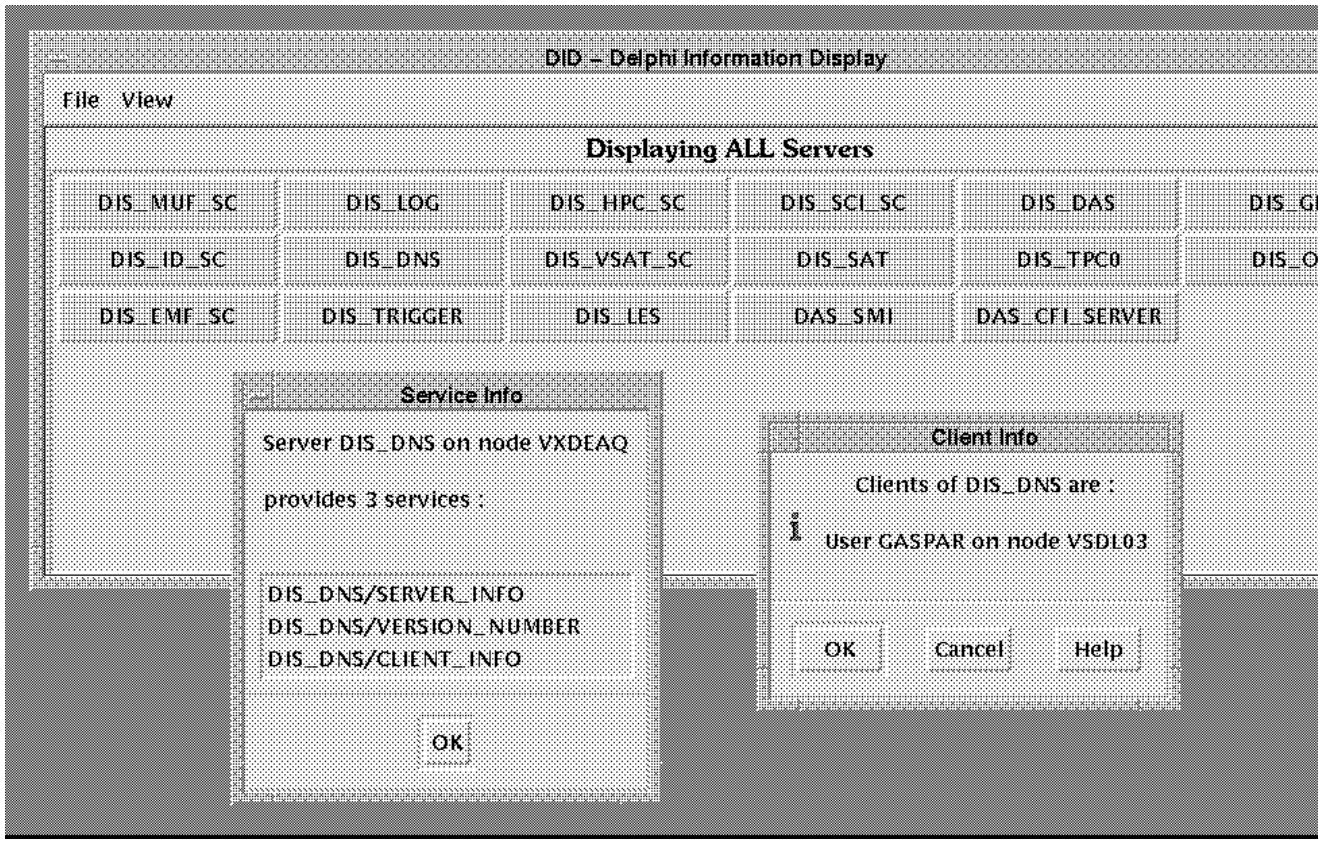


2.4 DID - DIM Information Display

The DID is a MOTIF utility that allows the visualization of the state of the system. It provides information about all the servers running at a certain moment, the services they offer and the clients connected to them.

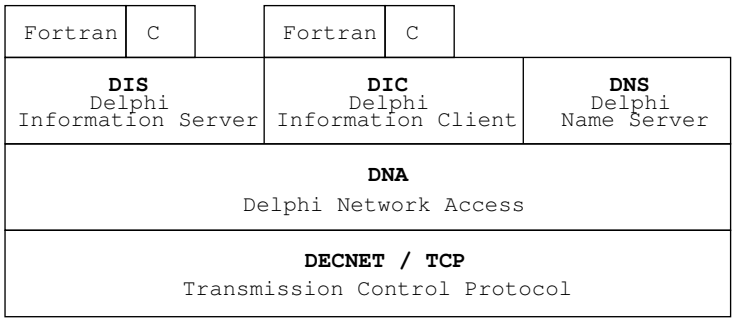
On the figure <REFERENCE>(fig_did) The services provided by the Name Server can be seen, as well as the clients connected to it, the only client beeing at that moment this User Interface.

Figure 2–3 DID User Interface



3 Implementation

Figure 3–1 Layering of DIM



3.1 DNA - Delphi Network Access

DNA - Implements Network access routines for a client/server based model. DNA can handle multiple connections asynchrounously. The DNA package can work over TCP/IP or DECNET.

The DNA layer uses the VMS QIO facilities to open, read, write and close connections. A special care has been taken to avoid the system to block, so all calls to DNA will complete asynchrounously.

3.2 DIS - Delphi Information Server

The Server functionality is available as a library, the routines contained in this library can be called by already existing programs wanting to perform as a DIM server and/or by the general server.

This DIS library provides routines to declare services to be provided by the server, there are two types of services : information services and command services. Information services flow in the direction server -> client, command services flow in the direction client -> server. Both types of services have to be declared by the server.

For a complete list of the server routines please refer to Appendix Appendix A.

3.3 DIC - Delphi Information Client

The Client functionality is available as a library, the routines contained in this library can be called by programs wanting to perform as DIM clients, for ex. User Interfaces.

The DIC library allows clients to specify in wich services they are interested and by wich mechanism they want to get them, there are three possible ways, receive a service once, receive it every specified amount of time or receive it when there is a change. When a service arrives the data can just be stored in a client buffer or a client routine can be called. The clients can also send commands to servers, in this case no data is received back from the server.

For a complete list of the client routines please refer to Appendix Appendix B.

3.4 DNS - Delphi Name server

The Name Server is a stand alone package, it provides the funcionality discribed in the previous chapter. It also behaves as a DIM server in the sense that it provides services allowing a User interface or any client to get information about the servers and the services available in the system at a certain time.

3.5 Library Interface

The Server and the Client library provide both, a C and a Fortran interface.

A Server Library

A.1 Routine Description

The Server library implements most of the server functions. These routines will be called by already existing programs wanting to perform as a DIM server and/or by the general server.

dis_add_service

Add an information service to the list of provided services.

FORMAT

dis_add_service *name, type, address, size, user_routine, tag*

ARGUMENTS

*char *name*
Service name, same name used by client when requesting a service.

int type
Type of service, not used for the moment.

*int *address*
If the data provided by this service is stored in memory (global section, array, etc) and is to be sent as it is to the client, it's address.

int size
The size in bytes of the data to be passed to the client if the previous parameter has been specified.

*void *user_routine*
The address of a routine to be executed when it's time to send the data to a client, this routine will provide the data, the parameters of this routine will be specified later in this document.

int tag
A parameter to be sent to the user_routine in order to identify the service.

RETURNS

VMS Usage: **service_id**
type: **unsigned int**
access: **write only**
mechanism: **by value**

The service identifier, to be used when (if) updating the service (by calling dis_update_service).

DESCRIPTION

This routine has to be called once for every service provided by the server. There are two ways of providing the service : specifying the address and size of the data to be sent to the client or specifying the address of a routine that will provide the data.

dis_add_cmnd

Add a command service to the list of provided services.

FORMAT

dis_add_cmnd *name, type, cmnd_user_routine, tag*

ARGUMENTS

*char *name*

Service name, same name used by client when requesting the execution of a command.

int type

Type of service, not used for the moment.

*void *cmnd_user_routine*

The address of the routine to be executed when a command request is received from a client, the parameters of this routine will be specified later in this document. This parameter is optional, if it is specified as zero, the request will be queued and can be recovered later by calling the routine dis_get_next_cmnd.

int tag

A parameter to be passed to the cmnd_user_routine or to dis_get_next_ cmnd in order to identify the command service.

DESCRIPTION

This routine has to be called once for every command service provided by the server.

dis_get_next_cmnd

Get a command from the list of waiting command requests.

FORMAT

dis_get_next_cmnd *tag, buffer, size*

ARGUMENTS

*int *tag*

This parameter returns the tag given to dis_add_cmnd, it allows the identification of the command.

*int *buffer*

A buffer address where the command request is to be copied.

*int *size*

On entry this parameter contains the size of the buffer, on exit it contains the real size of the command request.

RETURNS

VMS Usage: **return_code**
type: **unsigned int**
access: **write only**
mechanism: **by value**

This routine returns 0 if no command request is waiting for execution, -1 if the command request doesn't fit on the specified buffer (truncated) and 1 if the command as been copied successfully into the user buffer.

DESCRIPTION

This routine has to be called by the user if no cmnd_user_routine address has been specified in dis_add_cmnd. It allows the user to get and execute the commands requested by a client.

dis_start_serving

Start handling client requests.

FORMAT

dis_start_serving *task_name*

ARGUMENTS

int task_name

The name of the task corresponds to the VMS object name, it will be used by the name server in order to find the server. This name should be unique on the node where the server is running.

RETURNS

VMS Usage: **return_code**
type: **int**
access: **write only**
mechanism: **by value**

Returns 1 if all the involved operations worked fine and 0 if an error occured.

DESCRIPTION

This routine will register within the name_server all the services declared with dis_add_service and dis_add_cmnd.

It will set up the server so that it will handle client requests.

It will also set up all the mechanisms so that when a service has to be sent to a client the user_routine that provides that service will be called or the data will be taken from the address and size provided in dis_add_service and sent to the client.

dis_update_service

Report the change on a service to interested clients.

FORMAT

dis_update_service *unsigned int service_id*

ARGUMENTS

unsigned int service_id
The service_id returned by dis_add_service.

DESCRIPTION

This routine should be called by the server program when there is a change for a given service (when possible).

This routine will check whether there are clients interested in this service and if they have requested to receive the data uppon change (monitored service) the data will be sent.

user_routine

Routine written by the user in order to provide a service.

FORMAT

user_routine *tag, address, size*

ARGUMENTS

int tag
A parameter in order to identify the service, the tag given to dis_add_ service.

int **address
Should return the address of the data to be sent to the client.

int *size
Should return the size of the data to be sent to the client

DESCRIPTION

This routine should be declared in dis_add_service if not using the address, size parameters, it will be called whenever the service has to be sent to the client.

This routine should provide the necessary gathering or computing of data and store it in a buffer in order to be sent as a service to a client.

cmnd_user_routine

Routine written by the user in order to execute a command when a command request is received from a client.

FORMAT

cmnd_user_routine *tag, address, size*

ARGUMENTS

int tag

A parameter in order to identify the command, the tag given to dis_add_cmnd.

*int *address*

The address of the buffer containing the command.

int size

The size of the command data.

DESCRIPTION

This routine should be declared in dis_add_cmnd for immediate execution on reception of a command request.

EXAMPLES

1

The following example implements a server. This server contains an example of each type of server routine. This server provides two information services and two command services with all the possible options.

```
#include <dis.h>

int buffer[] = { 0,1,2,3,4,5,6,7,8,9 };

void build_service(tag, address, size)
int tag;
int **address;
int *size;
{

 *address = buffer;
 *size = sizeof(buffer);
}

int service_id;
```

```
void execute_cmnd(tag, buffer, size)
int tag;
int *buffer;
int size;
{
 if(tag == 1)
 {
  printf("TEST_SYNCH Command %s received\n",buffer);
  dis_update_service(service_id);
 }
 if(tag == 2)
  printf("TEST_ASYNCH Command %s received\n",buffer);
}

main()
{
int tag;
int local_buffer[10];
int size = 40;

 dis_add_service("TEST_BY_BUFFER", 0, buffer, 40, 0, 0);
 service_id = dis_add_service("TEST_BY_ROUTINE", 0, 0, 0, build_service, 0
 dis_add_cmnd("TEST_SYNCH", 0, execute_cmnd, 1);
 dis_add_cmnd("TEST_ASYNCH", 0, 0, 2);
 dis_start_serving("DIS_TEST");
 while(1)
 {
  while(dis_get_next_cmnd(&tag, local_buffer, &size))
   execute_cmnd(tag, local_buffer, size);
  sleep(10);
 }
}
```

B Client Library

B.1 Routine Description

The Client library will implements most of the client functions. These routines will be called by programs wanting to perform as a DIM client, for ex.display programs.

dic_info_service

Request an information service from a server

FORMAT

dic_info_service *name, type, timeout, address, size, user_routine, tag, fill_address, fill_size*

ARGUMENTS

*char *name*
Service name, same name used by server when declaring the service.

int type
Type of service, constants defined are: ONCE_ONLY, TIMED or MONITORED.

int timeout
For a tIMED service "timeout" indicates the time interval the server should use to send new data, for ONCE_ONLY or MONITORED services it indicates the time after which the service is considered to have failed.

*int *address*
Address of the buffer where to store the data when the service returns from the server.

int size
The size in bytes of the previous buffer.

*void *user_routine*
The address of a routine to be executed when new data is returned from the server, the parameters of this routine will be specified later in this document.

int tag
A parameter to be sent to the user_routine in order to identify the service that has completed.

*int *fill_address*
Address of a buffer containing data to be stored in the service buffer in case the service doesn't succeed.

int fill_size
The size in bytes of the previous buffer.

RETURNS

VMS Usage: **service_id**
type: **unsigned int**
access: **write only**
mechanism: **by value**

The service identifier, to be used when (if) removing the service.

DESCRIPTION

This routine first contacts the name server in order to get the address of the server where the requested service is available, it then sends the request to the server. the Service can be of three types: ONCE_ONLY, TIMED or MONITORED. When the response to a service comes back from a server the client buffer is filled with the data and the user_routine (if specified) is executed.

After a timeout (timeout parameter for ONCE_ONLY and MONITORED services and 2*timeout parameter for TIMED services) the service is considered failed, the information in fill_address is copied into the client buffer and the user_routine is called.

If the server is not responding the client recontacts the name server and the name server will wake up the client as soon as the server is up.

dic_cmnd_service

Request the execution of a command by a server

FORMAT

dic_cmnd_service *name, type, address, size*

ARGUMENTS

char *name
Service name, same name used by server when declaring the service.

int *address
Address of the buffer containing the command data.

int size
The size in bytes of the previous buffer.

DESCRIPTION

This routine requests the execution of a command by a server, address and size contain the command parameters, the server is not supposed to report back the completion of the command.

If the server is not responding the name server takes care of queuing the request and the command will be sent as soon as the server is up.

This routine allows clients (user interfaces for ex.) to send commands to be executed by the servers.

dic_release_service

Called by a client when a service is not needed anymore

FORMAT

dic_release_service *service_id*

ARGUMENTS

unsigned service_id
The service_id returned by dic_info service.

DESCRIPTION

This routine tells the server not to update this service anymore and destroys all the references to it.

user_routine

Routine written by the user, called when new data arrives from a server. The user routine can be called with two or three parameters, please read carefully the parameter description.

FORMAT

user_routine *tag, buffer, size*

ARGUMENTS

int tag

A parameter in order to identify the service, the tag given to dic_info_ service.

*int *buffer*

This parameter exists only if the parameter "ADDRESS" has NOT been specified when calling dic_info_service. It contains the address of the data received from the server. This buffer address is valid only during the execution of this routine. If the data contained in this buffer has to be used later it's the responsability of the user to copy it to another buffer.

int size

The size in bytes of the data actually sent by the server.

DESCRIPTION

This routine is called on reception of data from a server, it can be used for ex. by user_interfaces in order to update the screen whith the new set of data.

EXAMPLES

1

The following example implements a client. This client contains an example of each type of client routine. The client requests one TIMED service (every ten seconds), one MONITORED service and executes two types of commands on the server.

```
#include <dic.h>

int buffer[10];
int no_link = -1;
int version;

rout(tag, size)
int tag,size;
{
int i;
```

```
 if(buffer[0] == -1)
  printf("Service TEST_BY_BUFFER not available\n");
 else
 {
  printf("received service TEST_BY_BUFFER\n\t");
  for(i=0;i<10;i++)
   printf("%d ",buffer[i]);
  printf("\n");
 }
 printf("\n");
}

rout1(tag, size)
int tag,size;
{
 if(buffer[0] == -1)
  printf("Service TEST_BY_ROUTINE not available\n");
 else
  printf("received service TEST_BY_ROUTINE\n");
}

main()
{
 int index = 0;
 char str[80];

 dic_info_service("TEST_BY_BUFFER",TIMED,10,buffer,40,rout,0,&no_link,4);
 dic_info_service("TEST_BY_ROUTINE",MONITORED,60, 0, 0, rout1,0,&no_link,4
 while(1)
 {
  sprintf(str,"SYNCH_CMND_%d\n",index);
  dic_cmnd_service("TEST_SYNCH",str,strlen(str)+1);
  sleep(2);
  sprintf(str,"ASYNCH_CMND_%d\n",index);
  dic_cmnd_service("TEST_ASYNCH",str,strlen(str)+1);
  index++;
  sleep(3);
 }
}
```