The DFN Remote Procedure Call Tool

Reference Manual
Vol.I + II

Rolf Rabenseifner
Jens Bönisch, Frank Deuble, Chris Dodge, Bernd Hesse, Mandana Karimian
Jan Kayser, Werner Kollak, Hans Dieter Reimann

December, 15, 1994

Release 1.0.60

Rechenzentrum der Universität Stuttgart
Prof. Dr.-Ing. R. Rühle
Allmandring 30
70550 Stuttgart
(Tel.: 0711-685-5530, FAX 0711-682357)

Sponsored by the
Verein zur Förderung eines Deutschen Forschungsnetzes (DFN),
and the Federal Minister for Research and Technology (BMFT)
under the grant numbers TK 558 VA 005.1 and VA 005.3
'Distributed Applications over Highspeed Data Communication'

Internal documentation name

dfnrpc/1.0/src/doc/*.tex (Text)
MAC/rabenseifner/DFNRPC_1.0/pics/* (Fig.-Src)
dfnrpc/1.0/src/doc/pics/*.showcase (Fig.-Src)
dfnrpc/1.0/src/doc/pics/*.eps (EPS-Abb.)

$DFNRPCROOT/dfnrpc/1.0/dfnrpc_e.ps (Printfile, english)
$DFNRPCROOT/dfnrpc/1.0/dfnrpc.ps (Printfile, Vol. I, german)
$DFNRPCROOT/dfnrpc/1.0/dfnrpc_obsolete.ps (Printfile, Vol. II, german)

Inhalt

Projectleader: Rolf Rabenseifner. Projectstart: 1. Sept. 1988

Sponsored by
Verein zur Förderung eines Deutschen Forschungsnetzes - DFN (German Research Network) and the Federal Minister for Research and Technology with the grant number TK 558 VA 005 *"Verteilte Anwendungen über Hochgeschwindigkeitsdatenkommunikation"* (Distributed Applications over Highspeed Datacommunication) and TK 558 VA 005.3 *"Erweitertes RPC-Tool für verteilte Anwendungen"* (Improved RPC-Tool for distributed Applications).

1 Introduction

The future of scientific and technical computing lies predominantly in the appropriate distribution of computational tasks between workstations and super-computers. This type of decentralized data processing structure has been highlighted in the recommendation of the Baden Württemberg science council, the DFG and the third general IT plan of Baden-Württemberg. The implication for the scientist is that the majority of his work is done on a desktop workstation, but when required he must have direct access to a more powerful computer (server). The use of workstations, with their high performance graphics is now becoming essential for the visualisation of scientific or technical data.

To maintain and develop the quality of our research at an international level, the use of state of the art, high performance computer systems is vital, but it is not economically realisable to place a "super-workstation" on every desktop. The only practical alternative is to provide access to central **Compute-Servers** over **Highspeed Data Networks (HDN)**. Today's compute-servers may be either vector computers, parallel computers or workstations clusters, but whichever type of system is used, fast display of the computational output is only possible with high speed data transfer.

The inauguration of the 140 MBit glass-fibre connection between Stuttgart and Karlsruhe recently demonstrated the possibilities that will opened to scientists over the next few years. However, the use of currently available software for coupling compute-servers with workstations over high speed data networks remains the domain of specialists. Distributed computing of this form is significantly more complex than more traditional methods of remote computer access, for example, via a simple terminal program. Therefore, it is necessary to develop tools for the scientific community which will allow scientists to display the results from jobs running on high powered compute-servers simply and quickly on their workstations.

The Remote Procedure Call Tool of the German Research Network (**DFN-RPC**) is such a system, which can be used to easily distribute applications between the compute-server and workstation (client) as suits the particular application. Both the effort required by the DFN-RPC user, and the amount of technical knowledge about networks (communication protocols etc.) has been kept to a minimum. Test tools, such as debuggers, can be used with applications distributed with the DFN-RPCs, and the finished program can be so arranged that remote processes are started and stopped transparently.

The Remote Procedure Call Tool described in this handbook makes possible the straightforward **distribution of application programs** between workstations and compute servers.

The DFN-RPC tar file and documentation be found via ftp at:
ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/
or by using a WWW browser at:
ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html

2 Installing the DFN-RPC

IMPORTANT: Upper and lower case print is significant.

To install the DFN-RPC, take the following steps:

1. Prepare the system on which it is to be installed; disk space, compiler etc.
2. Collect the tar file.
   This is delivered, where possible, over the network.
3. Unpack the tar file.
   The DFN-RPC is normally delivered as source code, however all program documentation (comments) have been removed.
4. Setup the installation parameter set.
   Installation parameter sets for many computer and operating system types are delivered with in the tar file. Most of these can be directly used, or require at most minor modification.
5. Configure and run the navigation script dfnrpc_set.
   This enables simple navigation through the DFN-RPC directory tree both during the installation process, and later when using the software.
6. Install one of the configurations.
   All libraries and examples are compiled and linked.

Once this last step is finished, the installed is complete.

At the end of this chapter, two further aspects of the installation are discussed:

- Installing updates (e.g. for error correction),
- Installation on a fileserver.

2.1 Requirements

Permissions	Normal user write permission in a directory.
Disk space	35 - 55 MBytes (complete inst. including examples) In detail: 3 MBytes for the compressed tar file. 8 MBytes for the source (at least during the installation). 4 MBytes for an installation without examples. 20 - 40 MBytes for the examples. (Exact requirements dependent on operating system.)
Operating system	Posix (or VMS)
Compilers	Fortran and C
Transport Layer	TCP/IP (and/or ISO/OSI TP4, see below)

2.2 Collecting the Tar File

The DFN-RPC release 1.0.60 is delivered as a compressed tar file for Unix systems.

`dfn_rpc_1.0.60.tar.z`

Either *ftp* (in *binary* mode) or a WWW browser (e.g. Mosaic) can be used to collect the file from the following address:

ftp://ftp.uni-stuttgart.de/pub/rus/dfn_rpc/dfn_rpc_1.0.60.tar.Z

2.3 Unpacking the Tar File

DFN-RPC can be installed in any directory on the system, which will be referred to as DFNRPC-ROOT. In the following example command sequences, DFNRPCROOT is */usr/3rdparty/dfn*.

Unpacking the Tar File:

```
cd    /usr/3rdparty/dfn
zcat  /..._tar_file_directory_.../dfn_rpc_1.0.60.tar.Z | tar xvf -
```

The resulting subtree *dfnrpc/1.0/src* contains the complete DFN-RPC software as source code (without internal documentation or comments).

```
Files  in /usr/3rdparty/dfn/dfnrpc/1.0
                  / dfnrpc.ps        This handbook as postscript file
                  / dfnrpc_set       csh script for navigation through software

Files  in /usr/3rdparty/dfn/dfnrpc/1.0/src
                  / README           Read before undertaking further installation
                  / inst_SUN4        Installation script for a Sun 4
                  / inst_CRAY2       Installation script for a Cray-2
                  / install          Central installation script.
                  / install_...      Help files for installation.


Subdirectories in /usr/3rdparty/dfn/dfnrpc/1.0/src

 .../ makefiles   / bin_makefile     Generic makefile for sys/.../bin
                  / lib_makefile     Generic makefile for sys/.../lib
                  / include_makefile Generic makefile for sys/.../include
 .../ ct4         / ...              Source for DFN-RPC library
 .../ ipsm        / ...              Source for interactive process start module
 .../ stubgen     / ...              Source for the stub generator (part 1)
 .../ rpc         / dfnrpcgen        Source for the stub generator (part 2)
                  / rpcmacro.m4
                  / dfn...           Help files and utilities

                                    Examples:
 .../ examples/sum / README          Introductory example in Fortran
                  / source_makefile
                  / sum.rpi
 .../ examples/csum/ README          Introductory example in C
                  / source_makefile
                  / csum.rpi
                  / csum_lb.c
                  / csum_cb.c
                  / csum_sb.c
 .../ examples/awi / ...             Example from AWI
 .../ examples/ben / ...             DFN-RPC benchmarks
 .../ examples/cav / ...             Flow over a cavity
 .../ examples/cvttest / ...         Test for the conversion routines
 .../ examples/exa / ...             RPI example from this book
 .../ examples/moni / ...            Monitor example
```

```
.../ examples/para / ...          Parallel RPC demo program
.../ examples/ph1 / ...           Simple PHIGS example program
.../ examples/ph2 / ...           as ph1, but with manufacturer depend. init.
.../ examples/phi / ...           Distributed PHIGS
.../ examples/pipe / ...          Datapipes
.../ examples/rfa / ...           Remote file access
.../ examples/rpcpipe / ...       Example RPC and datapipes
.../ examples/test / ...          Data transmission test program
.../ examples/ua1a / ...          Example - remote write - interactive
.../ examples/ua1b / ...                                 - with NQS
.../ examples/ua1c / ...                                 - with rsh
.../ examples/ua1d / ...                                 - with IPSM
```

2.4 Platforms

Table 2.1 lists the platforms on which the DFN-RPC is available. The flags MANU, ARCH and OPER in the inst_... files correspond to the dfnrpcgen options -M, -A and -O.

The DFN-RPC can usually be installed on other compatible platforms with the nearest values of the options MANU, ARCH and OPER, for example on Unicos 8.0 the option ARCH=unicos_7.0 should be used, or with IRIX 5.2, it should be installed with ARCH=IRIX_5.1.

Additionally, the DFN-RPC supports the following alternatives (also see table. 2.2):

F2C_CC — Compilation of Fortran files using the public domain Fortran to C converter to create C files, then compiling them with a C compiler. When F2C_CC is not given, the machine standard Fortran is used. Also see release notes in section 2.11.6 on page 17.

ULTRA — Additional UltraNet native protocol stack if ULTRA=true. Otherwise, ULTRA=false must be set.

OSI — Additional OSI protocol stack. Values can be; false (no OSI installed), sunlink and sunnet (products from sun), cray (Cray OSI 1.0) and cray2 (Cray OSI 2.0). ULTRA and OSI are mutually exclusive.

PHIGS — Example programs using Phigs.

The Phigs examples are now not automatically compiled with the installation. OSI and PHIGS are no longer supported.

Flags in the inst files MANU	ARCH	OPER	Fortran compiler	supplied inst_...files	accompanying AFS-Arch.
alliant	FX	Concentrix_5.0	fortran	inst_ALLI	
apollo	DN10000	SR10.2	ftn	inst_APO	
cray	cray2	unicos_4.0	cf77		
		unicos_5.0	cf77		
		unicos_6.0	cf77		
		unicos_7.0	cf77		
	ymp	unicos_7.0	cf77	inst_CRAYYMP	c90_uni8
convex	C120[1]	ConvexOS_V7.1	fc	inst_CVX	
	C120ieee[2]	ConvexOS_V7.1	fc	inst_CVXI	
dec	vax	ultrix	fort	inst_VAX	vax_ul4
	dec	ultrix	f77	inst_DEC	pmax_ul43
		OSF_1	f77	inst_ALPHA	alpha_osf1
fujitsu	vp	uxpm	frt -Ab	inst_FUJITSU	
hp	pa_risc	hp_ux	f77	inst_HP	hp700_ux90
ibm	RS6000	AIX	xlf	inst_RS6	rs_aix32
intel	paragon	OSF_1	f77	inst_PARAGON	paragon_12
sg	iris3d	IRIX	f77		
	iris4d	IRIX_3.2	f77	inst_IRIS4	sgi_40
		IRIX_5.1	f77	inst_IRIS4_51	sgi_52
sun	sun3	SunOS_3.5	f77		
		SunOS_4.0.3	f77		
		SunOS_4.1	f77		sun3_411
	sun4	SunOS_3.5	f77		
		SunOS_4.0.3	f77		
		SunOS_4.1	f77	inst_SUN4	sun4_411
		SunOS_52	f77		sun4_52

Table 2.1: Supported DFN-RPC platforms

Flags in the inst files MANU	ARCH	OPER	Fortran compiler	supplied inst_...Files	**Additional options**
any	any	any	fc		F2C_CC=path for f2c and fc
sg	iris4d	IRIX_3.2	fc	inst_IRIS4F2C	F2C_CC= /client/bin
convex	C120	ConvexOS_V7.1	fc	inst_CVXU	ULTRA=true
sg	iris4d	IRIX_3.2	f77	inst_IRIS4U	ULTRA=true
cray	cray2	unicos_6.0	cf77		ULTRA=true
sun	sun3	SunOS_4.0.3	f77	inst_SUN3O	OSI=sunnet
sun	sun4	SunOS_4.1	f77		OSI=sunnet
cray	cray2	unicos_6.0	cf77		OSI=cray
cray	cray2	unicos_7.0	cf77	inst_CRAY2	OSI=cray2

Table 2.2: Optional platforms

Additionally, the following installation files are included:

inst_...P. Has PHIGS=true set.

[1] Computers with Convex native real number representation as default.

[2] Computers with IEEE real number representation as default.

Configuration name in dfnrpc_set	Installation parameter set	Configuration directory	All files are placed in the subtree during installation:
ALLI	inst_ALLI	alliant	.../dfnrpc/1.0/sys/alliant
ALLID	inst_ALLID	alliant	.../dfnrpc/1.0/sys_debug/alliant
ALPHA	inst_ALPHA	alpha	.../dfnrpc/1.0/sys/alpha
ALPHAD	inst_ALPHAD	alpha	.../dfnrpc/1.0/sys_debug/alpha
ALPHA_sw	inst_ALPHA_sw	alpha	/sw/axp_osf1/dfnrpc-1.0.60
APO	inst_APO	apollo	.../dfnrpc/1.0/sys/apollo
APOD	inst_APOD	apollo	.../dfnrpc/1.0/sys_debug/apollo
CRAY2	inst_CRAY2	cray2	.../dfnrpc/1.0/sys/cray2
CRAY2D	inst_CRAY2D	cray2	.../dfnrpc/1.0/sys_debug/cray2
CRAYYMP	inst_CRAYYMP	cray_ymp	.../dfnrpc/1.0/sys/cray_ymp
CRAYYMPD	inst_CRAYYMPD	cray_ymp	.../dfnrpc/1.0/sys_debug/cray_ymp
CRAYYMP_sw	inst_CRAYYMP_sw	cray_ymp	/sw/c90_uni8/dfnrpc-1.0.60
CVX	inst_CVX	convex	.../dfnrpc/1.0/sys/convex
CVXD	inst_CVXD	convex	.../dfnrpc/1.0/sys_debug/convex
CVXI	inst_CVXI	convex_ieee	.../dfnrpc/1.0/sys/convex_ieee
CVXID	inst_CVXID	convex_ieee	.../dfnrpc/1.0/sys_debug/convex_ieee
CVXI_sw	inst_CVXI_sw	convex_ieee	/sw/convex_os10/dfnrpc-1.0.60
CVXU	inst_CVXU	convexultra	.../dfnrpc/1.0/sys/convexultra
CVXUD	inst_CVXUD	convexultra	.../dfnrpc/1.0/sys_debug/convexultra
DEC	inst_DEC	dec	.../dfnrpc/1.0/sys/dec
DECD	inst_DECD	dec	.../dfnrpc/1.0/sys_debug/dec
DEC	inst_DECP	dec	.../dfnrpc/1.0/sys/dec
DECD	inst_DECPD	dec	.../dfnrpc/1.0/sys_debug/dec
DEC_sw	inst_DEC_sw	dec	/sw/pmax_u143/dfnrpc-1.0.60
FUJITSU	inst_FUJITSU	fujitsu	.../dfnrpc/1.0/sys/fujitsu
FUJITSUD	inst_FUJITSUD	fujitsu	.../dfnrpc/1.0/sys_debug/fujitsu
HP	inst_HP	hp	.../dfnrpc/1.0/sys/hp
HPD	inst_HPD	hp	.../dfnrpc/1.0/sys_debug/hp
HP_sw	inst_HP_sw	hp	/sw/hp700_ux90/dfnrpc-1.0.60
IRIS4	inst_IRIS4	iris4	.../dfnrpc/1.0/sys/iris4
IRIS4D	inst_IRIS4D	iris4	.../dfnrpc/1.0/sys_debug/iris4
IRIS4F2C	inst_IRIS4F2C	iris4f2c	.../dfnrpc/1.0/sys/iris4f2c
IRIS4DF2C	inst_IRIS4DF2C	iris4f2c	.../dfnrpc/1.0/sys_debug/iris4f2c
IRIS4	inst_IRIS4P	iris4	.../dfnrpc/1.0/sys/iris4
IRIS4D	inst_IRIS4PD	iris4	.../dfnrpc/1.0/sys_debug/iris4
IRIS4U	inst_IRIS4U	iris4ultra	.../dfnrpc/1.0/sys/iris4ultra
IRIS4UD	inst_IRIS4UD	iris4ultra	.../dfnrpc/1.0/sys_debug/iris4ultra
IRIS4U	inst_IRIS4UP	iris4ultra	.../dfnrpc/1.0/sys/iris4ultra
IRIS4UD	inst_IRIS4UPD	iris4ultra	.../dfnrpc/1.0/sys_debug/iris4ultra
IRIS4_51	inst_IRIS4_51	iris4_51	.../dfnrpc/1.0/sys/iris4_51
IRIS4_51D	inst_IRIS4_51D	iris4_51	.../dfnrpc/1.0/sys_debug/iris4_51
IRIS4_51_sw	inst_IRIS4_51_sw	iris4_51	/sw/sgi_51/dfnrpc-1.0.60
IRIS4_sw	inst_IRIS4_sw	iris4	/sw/sgi_40/dfnrpc-1.0.60
LINUX	inst_LINUX	linux	.../dfnrpc/1.0/sys/linux
LINUXD	inst_LINUXD	linux	.../dfnrpc/1.0/sys_debug/linux
LINUX_sw	inst_LINUX_sw	linux	/sw/linux_1/dfnrpc-1.0.60
PARAGON	inst_PARAGON	paragon	.../dfnrpc/1.0/sys/paragon
PARAGOND	inst_PARAGOND	paragon	.../dfnrpc/1.0/sys_debug/paragon
PARAGON_sw	inst_PARAGON_sw	paragon	/sw/paragon_12/dfnrpc-1.0.60
RS6	inst_RS6	ibmrs6	.../dfnrpc/1.0/sys/ibmrs6
RS6D	inst_RS6D	ibmrs6	.../dfnrpc/1.0/sys_debug/ibmrs6
RS6_sw	inst_RS6_sw	ibmrs6	/sw/rs_aix32/dfnrpc-1.0.60
SUN3O	inst_SUN3OP	sun3osi	.../dfnrpc/1.0/sys/sun3osi
SUN3OD	inst_SUN3OPD	sun3osi	.../dfnrpc/1.0/sys_debug/sun3osi
SUN3O	inst_SUN3O	sun3	.../dfnrpc/1.0/sys/sun3osi
SUN3OD	inst_SUN3OD	sun3osi	.../dfnrpc/1.0/sys_debug/sun3osi
SUN4	inst_SUN4	sun4	.../dfnrpc/1.0/sys/sun4
SUN4D	inst_SUN4D	sun4	.../dfnrpc/1.0/sys_debug/sun4
SUN4O	inst_SUN4OP	sun4osi	.../dfnrpc/1.0/sys/sun4osi
SUN4OD	inst_SUN4OPD	sun4osi	.../dfnrpc/1.0/sys_debug/sun4osi
SUN4	inst_SUN4P	sun4	.../dfnrpc/1.0/sys/sun4
SUN4D	inst_SUN4PD	sun4	.../dfnrpc/1.0/sys_debug/sun4
SUN4_52_sw	inst_SUN4_52_sw	sun4	/sw/sun4_52/dfnrpc-1.0.60
SUN4_sw	inst_SUN4_sw	sun4	/sw/sun4_411/dfnrpc-1.0.60
VAX	inst_VAX	vax	.../dfnrpc/1.0/sys/vax
VAXD	inst_VAXD	vax	.../dfnrpc/1.0/sys_debug/vax
VAX_sw	inst_VAX_sw	vax	/sw/vax_u14/dfnrpc-1.0.60
VMS	inst_VMS	vms	.../dfnrpc/1.0/sys/vms
VMSD	inst_VMSD	vms	.../dfnrpc/1.0/sys_debug/vms

Table 2.3: Supplied installation parameter sets

inst_...D	All files are compiled with debugging on.
inst_...all	The examples are compiled with the additional transfer encoding types of dec, vax and cray as well as the standard ieee.
inst_...sw	Installation in /sw, see "Central Installation" below. With this installation, absolute paths are placed in the makefiles. CONF thus has to start with '/', and cannot be longer than 32 characters.

The installation file for VMS can be given upon request.

2.5 A Custom Installation Parameter Set

It's possible to write a customised installation parameter set based on one of the supplied installations, however it is possible that the supplied installation files will be changed with new releases, so it is not recommended that these are edited directly. There are basically two names that can be set:

Configuration name	The configuration name is used in the shell script **dfnrpc_set**, which aids navigation through the software as described below. The installation parameter set filename is usually (but not always) **inst_**configuration-name.
Configuration directory	a) If the directory name given here does not begin with a '/', i.e. it is a **relative path**, then: Installation of the DFN-RPC creates the directories .../dfnrpc/1.0/**sys** and/or .../dfnrpc/1.0/**sys_debug** below the directory .../dfnrpc/1.0. For each configuration installed, a subdirectory will be created in the above **sys** directories, the name of which is chosen here. This is the so called **local installation**. b) If the directory name starts with a '/', i.e. it is an **absolute path**, then: The installation will be placed under this path. This is the so called **central installation**.

Table 2.3 on page 7 lists the supplied configuration sets.

Example inst_SUN4

```
#!/bin/csh
#
#    inst_SUN4   -   DFN-RPC Installation Parameter File
#    ---------------------------------------------------
#
setenv DFNRPCROOT `cd ../../..; pwd | sed -e 's|^/tmp_mnt||'`
setenv CONF sun4

setenv MANU sun
setenv ARCH sun4
setenv OPER SunOS_4.0.3
setenv ULTRA false
setenv ONLYSRV false
setenv PHIGS true
setenv OSI false
```

```
setenv TRANS_ieee true
setenv TRANS_cray false
setenv TRANS_dec  false
setenv TRANS_vax  false
setenv SYS_OPTIM true
setenv SYS_DEBUG false

setenv COMP_SYS  true
setenv COMP_EXA  true

umask  2
/bin/csh install
```

WARNING: There should be no extra spaces at the end of the line!

The installation parameters are explained below:

- **Filename inst_***SUN4*

  The second part of the filename is the so called *configuration name*. It can be freely chosen.

- **#!/bin/csh**

  This causes the script to be run from the C-shell (csh). This line should not be changed.

- **# inst_***SUN4* **- DFN-RPC Installation Parameter File**

  The configuration name should also be placed here in a comment.

- **setenv DFNRPCROOT 'cd ../../..; pwd | sed -e 's|^/tmp_mnt||''**

  This extracts the DFN-RPC root directory from the current directory, which, in this example is */usr/3rdparty/dfn*. The command must remove the various prefixes normally used with automounters. By default, the directory */tmp_mnt*, if found, will be removed.

- **setenv CONF** *sun4*

  Name for the configuration directory. All files created during the installation are placed in this directory, which in this example would be /usr/3rdparty/dfn/dfnrpc/1.0/sys/*sun4* or /usr/3rdparty/dfn/dfnrpc/1.0/sys_debug/*sun4*. The configuration directory name can be freely chosen, it can even be given as a directory tree, e.g. *sun/4/osi* (but only with MANU=cray, sg, sun).

With a central installation, an absolute path must be given (see also section 2.10 on page 14). The following lines are dependent on the computer configuration:

- **setenv MANU** *sun*

  Manufacturer: *alliant, apollo, convex, cray, dec, ibm, sg* (Silicon Graphics), *sun*

- **setenv ARCH** *sun4*

  Architecture: see table above.

- **setenv OPER** *SunOS_4.0.3*

  Operating system: see table above.

- **setenv ULTRA** *false*

  This flag indicates whether the ULTRA compatibility library should be used: *true* or *false*. *true* is only possible when the ULTRA compatability library is installed, and **OPER** = Concentrix_5.0, ConvexOS_V7.1,UNICOS_5.0 to 7.0, IRIX_3.2, SunOS_4.03 or SunOS_4.1.

- **setenv ONLYSRV** *false*

  The 'Compute Server Only' flag. This flag should be *true* on compute servers, that is on systems without a graphics console, otherwise *false*. When the flag is *true*, the visualisations sections of the example programs are not compiled. This option is now obsolete, and should be set to *false*.

- **setenv PHIGS** *false*

  This flag indicates whether PHIGS is installed or not: *true* or *false*. This is relevant for the examples cav, phi, ph1 and ph2 only. It is no longer supported, and is no longer automatically compiled with the installation.

  true is only possible with the following operating systems: **OPER** = SR10.2, Ultrix, AIX, IRIX, IRIX_3.2, SunOS_3.5 to SunOS_4.1. SunPHIGS 1.2 is necessary for the SUN.

- **setenv OSI** *false*

  The OSI flags indicates which OSI software is installed:

false	no OSI software
sunlink	SunLink OSI 6.0 or compatible (only possible with SunOS_4.03 and SunOS_4.1)
sunnet	SunNet OSI 7.0 or compatible (only possible with SunOS_4.03 and SunOS_4.1)
cray	cray OSI rel. 1.0 (only with **OPER** = UNICOS_6.0)
cray2	cray OSI rel. 2.0 (only with **OPER** = UNICOS_7.0)

  The OSI protocol stack is no longer supported.

- **setenv F2C_CC** *unused*

  The F2C_CC flag indicates whether the public domain converter *f2c* is used instead of the standard Fortran compiler. Possible values are:

  unused or no F2C_CC entry: the manufacturers proprietary Fortran compiler is used for installation.
  A directory, e.g. /usr/local/bin:
  Path under which *f2c* and *fc* are found. See section 2.11.6 on page 17.

The following flags are used to specify choices for the generation of the DFN-RPC code. Appropriate directories and makefiles are created as a result.

- **setenv TRANS_ieee** *true*

  When *true* is given, the example directories and makefiles for the transfer encoding *ieee* are created, which, for the example here would be:
  */usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/examples_***ieee**.

  When *false* is given, this transfer encoding is ignored during generation of the examples.

- **setenv TRANS_cray** *false*

  As above, but for cray transfer encoding. In the above example, when this is *true*, the directory ...**_cray** will be created.

- **setenv TRANS_dec** *false*

  As above, but for dec transfer encoding.

- **setenv TRANS_vax** *false*

  As above, but for vax transfer encoding.

- **setenv SYS_OPTIM** *true*

  true = generation should take place in the directory
  */usr/3rdparty/dfn/dfnrpc/1.0/***sys***/sun4 without* the debug option.

- **setenv SYS_DEBUG** *false*

  true = generation should take place in the directory
  /usr/3rdparty/dfn/dfnrpc/1.0/**sys**/*sun4 with* the debug option.

The following flags indicate which parts of the software are generated (i.e. compiled and linked) during installation.

- **setenv COMP_SYS** *true*
  DFN-RPC system compilation flag; *true* or *false*.
  When this is *true*, the following instructions are executed:

  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/include; make
  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/lib; make
  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/bin; make

  (if SYS_DEBUG or SYS_OPTIM are set.)
- **setenv COMP_EXA** *true*
  Examples compilation flag: *true* or *false*.
  If the flag is *true*, then the makefiles for the examples are executed:
  cd /usr/3rdparty/dfn/dfnrpc/1.0/sys(_debug)/sun4/examples_ieee/ua1a; make
  make
  ...
  (if SYS_DEBUG or SYS_OPTIM, and TRANS_ieee, ... are set.)
- **umask** *2*
  This flag ensures that all generated files are readable and writable by you and your group, and are also readable by the 'world'.
- **/bin/csh install**
  This line calls the installation script. It should not be changed.

The following parameters are set in the installation sets provided.

Configuration	TRANS_				SYS_		COMP_	
parameter set	ieee	cray	dec	vax	OPTIM	DEBUG	SYS	EXA
inst ...	true	false	false	false	true	false	true	true
inst ...D	true	false	false	false	false	true	false	false

The options MANU and ARCH are hardware specific flags (for example on SparcStations, ARCH = sun4). If in doubt about the OPER flag, select the lowest allowed value for the architecture. Also, if unsure about the other options (ULTRA, ONLYSRV, PHIGS and OSI), all should be set to *false*.

2.6 Navigating Through the DFN-RPC Software

Easy navigation through the software is useful when undertaking the installation. If the configuration has been customised, then the shell script dfnrpc_set in the directory */usr/3rdparty/dfn*/dfnrpc/1.0/src must be edited appropriately.

Approx. lines 119-122: A name (in capital letters) should be placed in the first of the following lines:

```
#4        SUN4             (or another configuration, see below.)
#4    Available configurations with optimized executables:
#4     ALLI ALPHA APO CVX CRAY2 DEC FUJITSU HP IRIS3 IRIS4 PARAGON
#4     RS6 SUN3 SUN4 VAX
```

Approx. lines 171-???: The configuration given above must be programmed in a similar manner to that given in the lines below for the configuration directory *sun4*. The directory name (here *sun4*) must be identical to the name given in the line *setenv CONF sun4* in the above installation script.

With a local installation (i.e. using a relative path in CONF), DFNRPCCONF must be set, however if it is a central installation (i.e. an absolute path in CONF), then DFNRPCCENTRAL must be set.

```
# The following lines represent a choice of different configurations.
alias ALLI    DFNRPCCONF sys/alliant
alias ALLID   DFNRPCCONF sys_debug/alliant
alias ALPHA   DFNRPCCONF sys/alpha
alias ALPHAD  DFNRPCCONF sys_debug/alpha
alias ALPHA_sw DFNRPCCENTRAL /sw/axp_osf1/dfnrpc-1.0.60
   ...
alias SUN4    DFNRPCCONF sys/sun4
alias SUN4D   DFNRPCCONF sys_debug/sun4
```

The following lines can then be placed in your .cshrc file, and actualised with 'source .cshrc':

```
setenv DFNRPCROOT /usr/3rdparty/dfn
setenv DFNRPCREL   1.0
setenv DFNRPCTRANS ieee
source $DFNRPCROOT/dfnrpc/$DFNRPCREL/src/dfnrpc_set
SUN4     (or the configuration that has been given above)
```

Once these five lines have been executed, the command *dfnh* can be used to obtain explanations of these scripts (also see section 4.11 on page 43).

2.7 Installing a Configuration

The following commands must be given:

```
dfns       or      cd /usr/3rdparty/dfn/dfnrpc/1.0/src
inst_SUN4  or      inst_<ConfigurationsName>
```

If **COMP_SYS** and **COMP_EXA** are both *true* set in the installation parameter set **inst_***ConfigurationsName*, then the installation is now complete.

If **COMP_SYS** is *false*, then the following makefiles must now be called by hand.

For this, the environment variables DFNRPCCONF and DFNRPCSRC must be set. If the script dfnrpc_set and its configuration (as described in the last section) has been used, then these variables are already correctly set. Alternatively, they can be set by hand, which for this example is:

```
setenv DFNRPCROOT     /usr/3rdparty/dfn
setenv DFNRPCREL      1.0
setenv DFNRPCTRANS    ieee
setenv DFNRPCSRC      $DFNRPCROOT/dfnrpc/$DFNRPCREL/src
setenv DFNRPCCONF     $DFNRPCROOT/dfnrpc/$DFNRPCREL/sys/sun4
                                               or    sys_debug/sun4
```

Once this has been done, the make process can be started, by changing to the appropriate directory and calling *make*:

```
dfni   or    cd $DFNRPCINCLUDE
make

dfnl   or    cd $DFNRPCLIB
make

dfnb   or    cd $DFNRPCBIN
make
```

If **COMP_EXA** is **false**, then the examples can be additionally made by hand:

```
sumb  or    cd $DFNRPCEXAMPLES/sum
make

csumb  or    cd $DFNRPCEXAMPLES/csum
make

etc.
```

2.8 Undertaking Error Correction

Currently, most error correction is undertaken by replacing a source file. Following this, the various *make* calls described in the previous section must be repeated.

2.9 Installation on a Fileserver

As each installation uses its own directory sub-tree, it is possible to place several installations in one DFNRPCROOT on a file server. The simplest case is when DFNRPCROOT is placed on a fileserver that is reachable by several different machines via NFS.[3] The advantage here is that only one copy of the DFN-RPC source is required on disk, and error correction can be more simply done.

[3] In this way, it is possible to port the whole DFN-RPC system to as many workstations and compute servers as is required by developers.

2.10 Central Installation

Using the DFN-RPC over NFS or AFS.

For some time now at RUS, a concept for maintaining the uniform organisation and availability of software packages has been in existence. Consequently, the amount of work required to maintain software (additional to the operating system) has been reduced. This section describes how the DFN-RPC software can be used in this uniform '/sw' installation environment via NFS or AFS. The machine types available, and whether they use NFS or AFS is shown in the table below. [6, 7]

For machine architectures not included in this list, it is naturally possible to install the DFN-RPC locally.

Any difference in DFN-RPC use via NFS or AFS is merely a result of how software is made available on the computer. Use of the software remains identical. The following sections describe the steps necessary for a workstation administrator to take when making the DFN-RPC software (as well as the software using the SW-installations concept) available on his computer.

via	<arch>	Hardware/Operating System
AFS	hp700_ux90	HP 9000 Series 700 running HP-UX 9.0
AFS	pmax_ul43	DECStation 2100, 3100 or 5000 (single processor) with Ultrix 4.3
NFS	axp_osf1[4]	DEC/Alpha running OSF/1 T2.0
AFS	vax_ul4	VAX systems running Ultrix 4.2 and 4.3
AFS	rs_aix32	IBM RS 6000/AIX 3.2, 3.2.1, 3.2.
AFS	sun4_411	Sun Sparc/Sun OS 4.1.1, 4.1.2, 4.1.3
AFS	sun4_52	Sun SparcStations running Solaris 2.3
NFS	sgi_40	Silicon Graphics/IRIX 4.0.x
AFS	sgi_52	Silicon Graphics running IRIX 5.2
NFS	c90_uni8	Cray-C90 running Unicos 8.0.x
NFS	paragon_12	Intel Paragon running OSF/1 Rel. 1.2
NFS[5]	convex_os10	ConvexOS, Rel. 10.1

For use via NFS, the following entry in /etc/fstab is required:

```
sw.rus.uni-stuttgart.de:/sw_nfs/sw/ ␣/sw␣nfs␣ro,intr,soft,timeo=50,retrans=2,bg␣0␣0
```

To mount, use the commands:

```
mkdir ␣ /sw
mount ␣ -vat ␣ nfs
```

Use via AFS:

```
ln ␣ -s ␣ /afs/rus/uni-stuttgart.de/sw ␣ /sw
```

The following steps are identical for both **NFS** and **AFS**:

```
ln ␣ -s ␣ /sw/clients/<arch> ␣ /client
```

The correct architecture must be given for <arch>.

The DFN-RPC software, (and all other software available via the SW-installations concept) can be found in the directory /client/bin. Libraries and include files necessary for DFN-RPC use are found in the directories /client/lib and /client/include respectively.

[4] Will be renamed as alpha_osf1.

[5] Not available from sw.rus.uni-stuttgart, but from ZDV, Uni. Tübingen.

Calling the DFN-RPC software must be done directly, not via environment variables as described in the handbook. Therefore, if /client/bin has been added to your PATH variable, the programs can be called directly (e.g. dfnrpcgen).

The DFN-RPC source is not available to the normal user via AFS/NFS, therefore navigation through the software, as described in section 4.3 of the handbook, is somewhat limited.

The directory /client/examples/dfnrpc/examples_ieee contains several examples of DFN-RPC use, which are described in the chapter 'Examples' on page 161.

The file /client/doc/dfnrpc-doc/dfnrpc.ps contains the DFN-RPC handbook in postscript form. Additionally, further information and publications related to the DFN-RPC can be found as postscript files in the directory /client/doc/dfnrpc-doc.

2.10.1 Differences Between Local and Central Installations

A central installation is defined as one where the value of CONF in the installation file inst_<platform> starts with a slash - "/".

- *dfnrpc_set in local and central installations:*

The start of dfnrpc_set remains unchanged:

```
setenv DFNRPCROOT /sw/src/dfnrpc-1.0.60
                  (or where you have installed the dfnrpc.)
setenv DFNRPCREL   1.0
setenv DFNRPCTRANS ieee
source $DFNRPCROOT/dfnrpc/$DFNRPCREL/dfnrpc_set
```

In a local installation, the line (for e.g.),

```
SUN4
```

or, for any installation without the debug option, the line

```
DFNRPCCONF sys/configuration_directory
```

must be called, where the `configuration_directory` must correspond to the entry CONF in the inst_... file.

With a central installation, the following must be given:

```
DFNRPCCENTRAL configuration_directory      or for e.g.
DFNRPCCENTRAL /sw/sgi_40/dfnrpc-1.0.60
```

so that various aliases are correctly set.

Additionally, in a central installation, the examples are collected together in the directory `$DNRPCCONF/examples`. In the installation on /sw, `/client/examples/dfnrpc/` points to this directory.

`dfnrpc_set` can be found in `$DFNRPCCONF/bin` or in `/client/bin`.

- *Paths included at compile time*

During installation, various paths are compiled into the code. These differ in the local and central installations.

File	used by	Search path in local /	central Install.
dfnrpc_stub.inc	INCLUDE stat. in the stubs	current directory	<CONF>/include
dfnrpcm4	dfnrpcgen	current$DFNRPCBIN	<CONF>/bin
rpcmacro_short.m4	dfnrpcm4	cur.$DFNRPCINCLUDE	<CONF>/include

dfnrpc.servers	Routine RPQUCI in application prog.	current directory	current directory
RPCAGMIB	Monitor program dfnrpcsrv and Stubs	current directory	current directory
dfnrpc.inc	Compiling the application prog.	current directory, or given by INCLUDE statement in application program	
libdfnrpc.a	Linking the application prog.	current$DFNRPCLIB in link command of the developer	<CONF>/lib in link command of the developer

During installation, <CONF> is resolved and compiled into the code, however, *current* ... values are resolved either during compilation of the application, or at application run time. With a local installation, the developer must always make sure that the environment variables $DFNRPCBIN, $DFNRPCINCLUDE and $DFNRPCLIB are correctly set. See also [17], page 18.

With a central installation, done within the /sw installation concept, the developer must replace <CONF> with the value /client. [6, 7]

2.10.2 Files in /sw – Links in /client

With the /sw-installation, various files are reachable over links in /client:

/client/bin/dfnrpcgen	the stubgenerator
/client/bin/dfnrpcsrv	the monitor MIB server
/client/bin/dfnrpcwin	the window orientiered monitor
/client/bin/dfnrpc_set	script for navigation
/client/bin/dfnrpcm4	to convert modified stubs
/client/bin/dfnrpcagt	(internal program for communication between stubs and dfnrpcsrv)
/client/include/dfnrpc.inc	Fortran include file
/client/include/dfnrpc_stub.inc	(stubs include file)
/client/lib/libdfnrpc.a	the DFN-RPC library
/client/examples/dfnrpc/examples_ieee	contains the examples: awi, ben, cav, con, crshexa, csum, cvttest, exa, moni, para, ph1, ph2, phi, pipe, rfa, rpcpipe, rshexa, test, ua1a, ua1b, ua1c and ua1d
/client/doc/dfnrpc-doc	DFN-RPC documentation.

2.11 Release Notes

In the following installations, there are some slight differences that should be noted:

2.11.1 Convex

If MANU=convex, it is necessary to use getsysinfo to determine which real number represntation the machine uses (native_default or ieee_default). The installation must be adjusted accordingly, either by setting ARCH=C120 or ARCH=C120ieee, or by using the correct installation script; inst_CVX or inst_CVXI.

If the DFN-RPC is used together with a C main program, then the environment variables FOR005 and FOR006 must be set at runtime, as shown in the example csum.

2.11.2 DEC RISC ULTRIX

From "DEC Fortran for Ultrix RISC Systems Version 3.1" onwards the compiler is capable of understanding various standard Fortran 77 features that are used by the DFN-RPC.

2.11.3 Fujitsu

With MANU=fujitsu, all Fortran applications and stubs with frt must be compiled with the -Ab option.

2.11.4 Apollo

The graphical monitor, and the X-Windows parts of the example "Test" cannot be used on Apollo machines.

2.11.5 Differences in X11 Installations

If X11 is installed in /usr/include (e.g. on Sun) instead of /usr/lib, then the following files must be edited so that the X11 libraries can be found: $DFNRPCSRC/makes/bin_makefile and $DFNRPCSRC/examples/test/source_makefile. Also, the appropriate include directories must be given with the -I option during the compilation of monbut.c and test.c.

2.11.6 Using *f2c* Instead of the Proprietry Fortran Compiler

If fc is to be used, then the following line must be set in the installation file:

setenv F2C_CC ≪ Directory where fc and f2c are found ≫

for example:

setenv F2C_CC /usr/local/bin

f2c also supports INTEGER*1 and LOGICAL*1,2, although this is not documented. INTEGER*1 has a range of 0..255, whereas f77 has a range of -128..127, which will cause an error if an INTEGER*1 is given a value in the range 128..255, compiled with f2c, sent as an INTEGER*2/4/8 and placed in an INTEGER or INTEGER*2/4/8 by the receiver. This will hapen with the test routines m2 and m3 when test_sb is compiled on a BYTE machine using f2c, and test_cb runs on a non-BYTE machine (e.g. Cray).

By default, fc uses the C compiler gcc from GNU. The DFN-RPC uses a modified fc that uses the proprietry C compiler (cc).

Please Note: The f2c release version 1993.04.28 or newer should be used. The value given for the environment variable F2C_CC is the path for f2c (converter) and fc (compiler = converter + gcc).

With the examples programmed in C, in the file dfnrpc/1.0/src/install.m4_h, the line CFLFLAGS = $(FCPATH)/../lib/libf2cF77.a $(FCPATH)/../lib/libf2cI77.a defines where the libraries are to be found (also see the penultimate line of fc).

If the version of f2c used does not know the option -Nl1000, then this must be changed in install.m4_h (approx. 9 line earlier).

Also see page 5 and page 10.

2.11.7 Differences Between Proprietary Fortran Compilers

As a result of hardware differences, the Fortran compiler may use different libraries during linking. Care must be taken here when linking C programs (as with examples csum and crshexa). The options listed in the file $DFNRPCSRC/install.m4_h for the architecture dependant CFLFLAGS may need to be edited.

3 sum and csum; Two Introductory Examples

3.1 Introduction

The following two examples show how an application can be distributed between two computers.

Examples **sum** and **csum** read two numbers from the keyboard, add them and then return the result. The input and output takes place on the first computer, while the addition is calculated by the second computer. For this to happen, computer one must first read the two numbers, then call the subroutine that adds the two numbers on computer two. Calling the addition routine on the remote computer will be done using the DFN-RPC.

sum.f:

```
program sum_client
real a,b,result
write(*,*)'please insert first number:'
read(*,*) a
write(*,*)'please insert second number:'
read(*,*) b
call add(a,b,result)
write(*,*)'the sum of ',a,' and ',b,
          ' is ',result
stop
end


subroutine add(a,b,result)
real a, b, result
write(*,*)'computing the sum of ',a,
          ' and ',b
result = a + b
return
end
```

csum_lb.c:

```
void add_(a,b,result)
  float *a,*b,*result;
 {
  printf("...computing the sum of \
         %f + %f...\n",*a,*b);
  *result = *a + *b;
 }

main()
 {
  float a,b,res;

  printf("Please insert first number  : ");
  scanf("%f",&a);
  printf("Please insert second number : ");
  scanf("%f",&b);
  add_(&a,&b,&res);
  printf("The sum of %1.8g plus %1.8g is \
         %1.8g\n",a,b,res);
 }
```

As the name "Remote Procedure Call" implies, only procedure or subroutine calls can be made across a network. Therefore all operations intended for the remote computer must be written as functions or subroutines.

3.2 From Local Call to Remote Procedure Call

To distribute a program with the DFN-RPC, the program should firstly be partitioned and secondly, the appropriate *client* and *server stubs* must be generated. A *Remote Procedure Interface Definition File* is required by the *Stub Generator* so that it knows the exact form the stubs should take in the application. Additionally, a couple of of initialization calls must be added for the *Object Start* (process start).

There are two ways to partition the program into the *Client* and *Server Parts.*

1. The source code is separated by hand, or
2. appropriate instructions are placed in the source code so that the stub generator can create the server and client parts from the extended original Fortran.

   In this case, the RPI definitions should also be placed in the Fortran source file. This is the "single source file principle", as explained in more detail below.

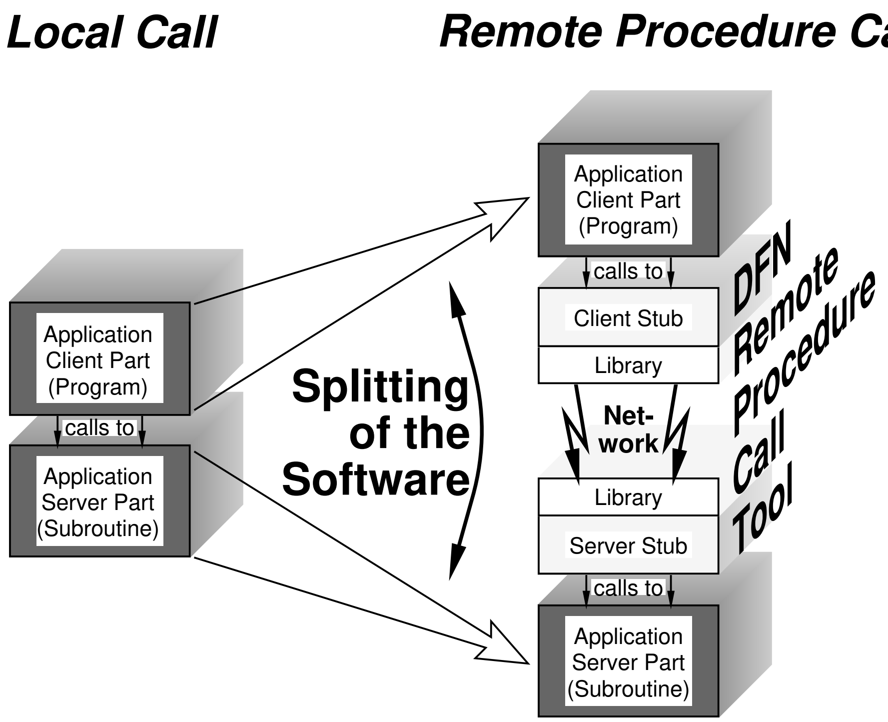


Abbildung 3.1: From 'Call' to 'Remote Procedure Call'

To generate the stubs, it is necessary to create a *Remote Procedure Interface (***RPI***) Definition*, that is, a description of the interface used for distributing the application. The RPI definition can either be placed in a separate file, or integrated into the Fortran source file (*single source file principle*) whereby one adds extra information to the argument list declarations[1] or place special comment lines.

The *Stub Generator* is the central tool of the DFN-RPC. It takes the RPI file as input, which may contain the following information:

- Information about program partitioning
- Fortran source code
- RPI definition

as described in detail in chapter 5 on page 49. Figure 3.2 gives a diagrammatical representation of the two main RPI definition methods.

Usage of the stub generator is given in chapter 7 on page 155.

Despite the transparent nature of the remote procedure call mechanism, the associated *object handling* cannot be completely hidden. The RPC mechanism itself, the transport services and the remote server process must all be correctly initialised.

The library routines necessary for this initialisation are described in sections 4.9 and 6 (p. 40f,107ff).

[1] To prevent confusion with the Fortran PARAMETER statement, the word argument is used for parameters.

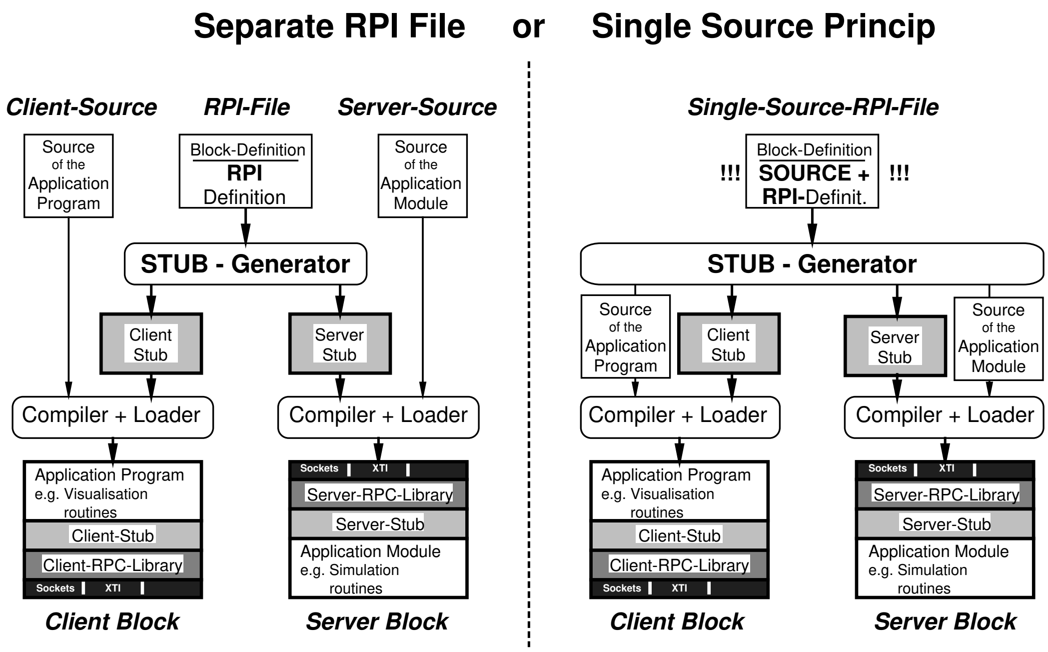


Abbildung 3.2: Separate RPI File or Single Source Principle

3.3 The Code for sum.rpi

The first computer in this example is the **client** on which the main part of the application runs. The second computer is the **server**, which runs a server program that executes a function called by the client. This example program is split into two sections; the code for the client part, and that for the server part. From this code we will create two modules, the client module "cm" which contains the main program, and the server module "sm" which comprises code to receive and process the remote routine call from the client, and the remote routine itself.

To distribute the program, a remote procedure interface (RPI) file must be created. In this example, we do this by extending the original Fortran source file.

A set of block definitions is placed at the start of the file, containing details of which modules are used to create the blocks. This information is necessary for the stub generator. Following this are the modules cm and sm themselves.

C*[BLK_BEG] cb, C*[MOD_USE] C*[MOD_USE] C*[BLK_END]	NAME='client block', CLIENT _SITE cm, LOCAL sm, CLIENT	In the **client block**, the word LOCAL causes the inclusion of the complete client module code cm in the block. The client stub is placed in cm through the word CLIENT .
C*[BLK_BEG] sb, C*[MOD_USE] C*[BLK_END]	NAME='server block', SERVER _SITE sm, SERVER	The **server block** contains the source code from sm and the server stubs.
C*[BLK_BEG] lb, C*[MOD_USE] C*[BLK_END]	NAME='local block', LOCAL _SITE cm, LOCAL sm, LOCAL	The **local block** is the un-distributed version of the program, containing the code from cm and sm without the RPC specific calls.

The compiled blocks are executable (part-) programs. The stubs are automatically created by the stub generator, and are used to send the RPC across the network.

Start of the client module cm.

Start of the client main program.

```
C*[MOD_BEG] cm,  NAME='sum client module'
         program sum_client
         real a,b,result
         integer iret
```

rpquci initialises the RPC on the client side, starts the server and returns the server object-ID *sobjid* (which is needed when more than one server is being used) and error flag *iret* to the client. If *iret*=0, then the RPC has been success - fully initialised. RPC specific instructions start with the C*RPC, and are ignored when the program is not to be distributed.

```
C*RPC    integer nserver,sobjid
C*RPC    write(*,*) ' sum_cb - this is the client of a'
C*RPC+     ' distributed fortran program'
C*RPC    nserver=1
C*RPC    call rpquci(nserver,sobjid,iret)
C*RPC    if(iret.ne.0) then
C*RPC      write(*,*)'sum_cb - error in rpc initialisation'
C*RPC      stop
C*RPC    endif
```

Read the arguments for the routine **add**, which is called by RPC. The call syntax appears as that of a local subroutine call.

```
         write(*,*)'please insert first number:'
         read(*,*) a
         write(*,*)'please insert second number:'
         read(*,*) b
         call add(a,b,result)
         write(*,*)'the sum of ',a,' and ',b,' is ',result
```

The library routine **rpqucf** closes the connection with the server and ends the client side RPC interface.

```
C*RPC    call rpqucf(iret)
C*RPC    if(iret.ne.0) then
C*RPC      write(*,*)'sum_cb - error in rpc finishing'
C*RPC    else
C*RPC      write(*,*)'sum_cb - finished'
C*RPC    endif
         stop
         end
C*[MOD_END]
```

Client main program end.

Client module end.

—

Server module start.

Server main program start.

To create the connection with the client, the server needs its address. If this isn't given as a command line argument, then **rpqusp** is called with an empty *master_address* string, causing it to prompt the user for the address. (Passing the address as a parameter is necessary when an automatic server start is required.)

```
C*[MOD_BEG] sm,  NAME='sum server_module'
C*RPC    program sum_server
C*RPC    integer iret
C*RPC    character*100 master_address
C*RPC
C*RPC    write(*,*)'sum_sb - this is the server of a '
C*RPC+   ' distributed fortran program '
C*RPC
C*RPC    if(iargc().lt.1) then
C*RPC      master_address= ''
C*RPC    else
C*RPC      call getarg(1,master_address)
C*RPC    endif
```

rpqusp initialises the RPC on the server side, creates the connection with the client, receives all RPC requests, and after client termination closes the connection.

```
C*RPC    call rpqusp(master_address,iret)
C*RPC    if(iret.ne.0) then
C*RPC      write(*,*)'rpc error - server stopped '
C*RPC      stop
C*RPC    endif
C*RPC    write(*,*)'sum_server - rpc correctly finished'
C*RPC    stop
C*RPC    end
```

Server main program end.

```
C*[RPI_BEG]
          subroutine add(a,b,result)
             real a
C*RPC+ [IN]
             real b
C*RPC+ [IN]
          real result
C*RPC+ [OUT]
C*[RPI_NON]
          write(*,*)'computing the result of ',a,' and ',b
          result=a+b
          return
          end
C*[RPI_END]
C*[MOD_END]
```

—
Start of the interface definition for the RPC routines.
With all function parameter declarations, extra information must be given to indicate whether it is an input or output parameter. This information is placed in a continuation line starting with the characters C*RPC+ so that this information is ignored in the un-distributed case.
The actual subroutine code is initiated with C*[RPI_NON], indicating to the stub generator that it should ignore the following lines.
This is where *result* is calculated.
Interface definition end.
Server module end.

3.4 Running the Example Program

Two windows are needed to run this example program. Start (in any order) the client program **sum_cb**[2] in one window, and **sum_sb** in the other. **sum_cb** prints out the *master_address* string, which must be given as input to the server program. The string contains the address of the client computer, the port number to be used for the connection and an encrypted password. Once the server has this information, the client-server connection is established.

Alternatively, the server can be started by using the remote shell (rsh) command, as shown in examples rshexa and crshexa (section 9.5 on page 166).

CLIENT:

```
host1$ sum_cb
sum_cb - this is the client of a distributed
                              fortran program
dfn-rpc - rpquci - quick client initialization
rpquci - please start server no. 1
master's address=

rpquci - connected with server 1
dfn-rpc - rpquci - client connected to 1 server
please insert first number:
3
please insert second number:
4

the sum of 3.0000 and 4.0000 is 7.0000
dfn-rpc - rpqucf - releasing all connections
dfn-rpc - rpqucf - client correctly finished
sum_cb - finished
host1$
```

SERVER:

```
host2$ sum_sb
sum_sb this is the server of a distributed
                              fortran program
dfn-rpc - rpqusp - quick server program
dfn-rpc - rpqusi - quick server initialization
rpqusi - please type in the master's address:

computing the sum of 3.0000 and 4.0000

dfn-rpc - rpquci - all rpcs done correctly
dfn-rpc - rpqusf - releasing the connection
sum_server - rpc correctly finished
host2$
```

[2] $DFNRPCEXAMPLES/sum/sum_cb
or /client/examples/dfnrpc/examples_ieee/sum/sum_cb

3.5 Compiling the Example sum

To compile the program sum, create a directory called mysum in your home directory. Go to 'mydir', and copy the file sum.rpi there.

The stub generator must now be used, and is found in the directory $DFNRPCBIN.

```
$DFNRPCBIN/dfnrpcgen -T ieee sum
f77 -o sum_lb sum_lb.f
f77 -o sum_cb sum_cb.f sum_cb_stb.f $DFNRPCLIB/libdfnrpc.a
f77 -o sum_sb sum_sb.f sum_sb_stb.f $DFNRPCLIB/libdfnrpc.a
```

This creates the following files:

sum_lb.f	the un-distributed program
sum_cb.f	the client part of the distributed program
sum_cb_stb.f	the client stubs for the remote procedure interface
sum_sb.f	the server part of the distributed program
sum_sb_stb.f	the server stubs for the remote procedure interface

The Fortran files need now to be compiled and linked, the client and server block programs being linked with their respective stub parts to create an executable program. The DFN-RPC library 'libdfnrpc.a' must also be linked with the two blocks.

3.6 The Makefile for sum.rpi

The makefiles for the example programs are architecture, operating system and compiler dependent, therefore they are generated automatically during the DFN-RPC installation from a machine independent source-makefile. As a result, the original makefiles contain some unused, additional variables and statements. This section explains the makefile operation, using a shorted makefile as an example.

```
# makefile for DFN-RPC example "sum"
# =================================
#
# Some hosts prefer SUFFIXES =, others prefer .SUFFIXES: (see below)
SUFFIXES   = .unused
.SUFFIXES:   .unused
```

Instead of using the built-in "make" rules, many of the dependencies and processing steps are explicitly given in the following lines:

```
#
# ...commands:
#
# $(FC)     = call to the fortran compiler (predefined macro)
# $(FLD)    = loader for fortran
FC          = f77
FLD         = $(FC)
SHELL       = /bin/sh
RM          = /bin/rm
```

The compiler and linker calls are set here, for which f77 is used in this example. To prevent errors resulting from undefined commands, the full paths are given for sh and rm.

```
#
# ...directories and files:
#
# From the shell environment the following variables will be imported:
# DFNRPCBIN DFNRPCLIB DFNRPCINCLUDE
#
# or in a central installation they must be defined here, e.g.:
# DFNRPCBIN      = /sw/sgi_40/dfnrpc-1.0.60/bin
# DFNRPCLIB      = /sw/sgi_40/dfnrpc-1.0.60/lib
# DFNRPCINCLUDE  = /sw/sgi_40/dfnrpc-1.0.60/include
# ...flags:
#         -c = only compile, no loader call
#         -g = debug       / -O = optimize (no debug)
FFLAGS  =  -c -g
LFLAGS  =  -g
#         -lbsd  =libraries for TCP
NETLIB  =  -lbsd
DFNRPCTRANS = ieee
```

The compiler and linker flags are set here. The -c option prevents f77 from linking the output object code, which is done in the next call without the -c option, so linking the object file with the necessary libraries. TCP library and transfer encoding flags are also set. If the three necessary path variables are not set, then the preceeding comment characters (#) must be removed.

```
#
# ...targets
#
modules:sum_lb sum_cb sum_sb
#
source:sum_lb.f sum_cb.f sum_sb.f sum_cb_stb.f sum_sb_stb.f
```

All three parts of the application are created by the call **make modules** or simply **make**. The command **make source** calls only the stub generator, which separates the source code and creates the stubs.

```
#
# Client part:
# ============
#
client: sum_cb
sum_cb: sum_cb.o sum_cb_stb.o $(DFNRPCLIB)/libdfnrpc.a
        echo "make module      sum_cb.o--->sum_cb"
        $(FLD) $(LFLAGS) -o sum_cb sum_cb.o sum_cb_stb.o \
               $(DFNRPCLIB)/libdfnrpc.a $(NETLIB)
sum_cb.o:sum_cb.f dfnrpc.inc
        echo "make: compilation   sum_cb.f--->sum_cb.o"
        $(FC) $(FFLAGS) sum_cb.f
sum_cb_stb.o:sum_cb_stb.f dfnrpc_stub.inc
        echo "make: compilation   sum_cb_stb.f--->sum_cb_stb.o"
        $(FC) $(FFLAGS) sum_cb_stb.f
```

Compiling and linking of the client block. f77 (used as both compiler and linker in this example) is given the appropriate flags for each call. The object files created from compilation of the client program sum_cb.f and the client stubs sum_cb_stb.f are linked together with the DFN-RPC and TCP libraries to form the client block. The necessary Fortran code is created by the stub generator (see above).

The server block is created similarly:

```
# Server part:
# ============
#
server: sum_sb
sum_sb:sum_sb.o sum_sb_stb.o $(DFNRPCLIB)/libdfnrpc.a
        echo "make module      sum_sb.o--->sum_sb"
        $(FLD) $(LFLAGS) -o sum_sb sum_sb.o sum_sb_stb.o \
               $(DFNRPCLIB)/libdfnrpc.a $(NETLIB)
sum_sb.o:sum_sb.f dfnrpc.inc
        echo "make: compilation   sum_sb.f--->sum_sb.o"
        $(FC) $(FFLAGS) sum_sb.f
sum_sb_stb.o:sum_sb_stb.f dfnrpc_stub.inc
        echo "make: compilation   sum_sb_stb.f--->sum_sb_stb.o"
        $(FC) $(FFLAGS) sum_sb_stb.f
#

# Not distributed version:
# ========================
#
local: sum_lb
sum_lb:sum_lb.o
        echo "make module         sum_lb.o--->sum_lb"
        $(FLD) $(LFLAGS) -o sum_lb sum_lb.o
sum_lb.o:sum_lb.f
        echo "make: compilation   sum_lb.f--->sum_lb.o"
        $(FC) $(FFLAGS) sum_lb.f
```

To create the local block, the compiler and linker are simply called with the appropriate flags. The source file is sum_lb.f.

```
# Sources :
# =========
#
sum_cb.f sum_cb_stb.f sum_sb.f sum_sb_stb.f sum_lb.f : sum.rpi $(DFNRPCBIN)/dfnrpcgen
        echo "stub generation     sum.rpi --->sum_*.f"
        $(DFNRPCBIN)/dfnrpcgen  -T $(DFNRPCTRANS) sum
```

The stub generator dfnrpcgen is called here, creating the Fortran source files from the RPI file. The above transfer encoding is given as parameter.

The dependency on dfnrpcgen given here in the first line ensures that when there are changes in the stub generator (e.g. a new release with optimized communications protocol), all stubs would be re-created.

```
dfnrpc.inc : $(DFNRPCINCLUDE)/dfnrpc.inc
        echo "make: get source    dfnrpc.inc"
        -$(RM) -f dfnrpc.inc ; echo reset_errorflag >> /dev/null
        ln -s $(DFNRPCINCLUDE)/dfnrpc.inc dfnrpc.inc
dfnrpc_stub.inc : $(DFNRPCINCLUDE)/dfnrpc_stub.inc
        echo "make: get source    dfnrpc_stub.inc"
        -$(RM) -f dfnrpc_stub.inc ; echo reset_errorflag >> /dev/null
        ln -s $(DFNRPCINCLUDE)/dfnrpc_stub.inc dfnrpc_stub.inc
```

Before compilation, links to the two include files dfnrpc.inc and dfnrpc_stub.inc must be set in the current directory. In case old links exist, the links are first deleted.

```
.SILENT:
```

Only the explanations given by "echo" and the compiler output appear on the screen.

3.7 csum – The Same in C

The example **csum** has exactly the same functionality as **sum**, but is written in C. When using the DFN-RPC in C programs, the RPI definition, which comprises the block defenition section and the call definitions for the distributed routines, must be placed in a separete file with the extension '.rpi'. The actual C source code is placed in seperate files; one file for each block. For the example **csum**, there are four source files: **csum.rpi**, **csum_cb.c**, **csum_sb.c** and **csum_lb.c**.

The Fortran routine names given in the RPI defenintion must be made callable by the C program, which is done by placing an underscore at the end of the (lowercase) function names in the C code.

csum.rpi

```
C*[BLK_BEG] cb,NAME='csum client block',CLIENT _SITE
C*[MOD_USE] sm, CLIENT
C*[BLK_END]
```

CLIENT causes the client stubs from sm and the C code form csum_cb.c to be merged to create this block.

```
C*[BLK_BEG] sb,NAME='csum server block',SERVER _SITE
C*[MOD_USE] sm, SERVER
C*[BLK_END]
```

The server block is created from the server stubs and the file csum_sb.c.
No defenition is required for the local block as it created entirley from the file csum_lb.c.

```
C*[MOD_BEG] sm, NAME='csum server routines'
C*[RPI_BEG]
 subroutine add (a,b,result)
 real a [IN]
 real b [IN]
 real result [OUT]

C*[RPI_END]
C*[MOD_END]
```

In the module defenition part of sm, the Fortran interface for the RPC called routines are given, together with the transfer direction foreach parameter.

csum_cb.c

```
main()
 {
  float a,b,res;
  int nserv,iret,sobjid[2];
  printf("csum_cb - This is the client of ");
              "a distributed C program\n");
  nserv = 1;
  rpquci_(&nserv,sobjid,&iret);
  if (iret != 0)
   {
    printf("csum_cb - error in rpc initialization\n");
    exit(1);
   }
  printf("Please insert first number : ");
  scanf("%f",&a);
  printf("Please insert second number : ");
  scanf("%f",&b);
  add_(&a,&b,&res);
  printf("The sum of %1.8g plus %1.8g is %1.8g\n",a,b,res);
  rpqucf_(&iret);
  if (iret != 0)
   {
    printf("csum_cb - error in rpc finishing\n");
    exit(1);
   }
  printf("csum_cb - finished\n");
  exit(0);
 }
```

– Initialise RPC
As **rpquci** is a Fortran routine, the name must be given with a following underscore character. All returned parameters must be passed as a pointer.

The parameters for the RPC routine must be passed as 'call by reference' values.
– Terminate RPC

csum_sb.c

```
main(argc, argv)
int argc; char *argv[];
 {
  int iret;
  int strlng = 1;
  char* str = " ";
  printf("csum_sb - This is the server of a ");
                "distributed C program\n");
  if (argc>1)
   {
    strlng = strlen(argv[1]);
    str = argv[1];
   }
  rpqusp_(str,&iret,strlng);
  if(iret != 0) exit(1);
  printf("csum_sb - finished\n");
 }
void add_(a,b,result)
float *a,*b,*result;
 {
  printf("csum_sb ...computing the sum of %1.8g + "
                          "%1.8g...\n",*a,*b);
  *result = *a + *b;
 }
```

When no argument is passed to the program, the master address string remains empty.

Otherwise, the length of the argument is checked and the master address string set to the argument's value.

To pass strings, a pointer to the string is placed at the string parameter position, and at the end of the parameter list, the string length as a 'call by value' parameter. With routines that are called via an RPC, care must be taken to ensure that all parameters passed to the routine are declared as pointers. In every other respect, the RPC routines are written as local routines.

Further Comments:
The syntax given above for calling Fortran routines from a C program (terminating the function name with an underscore) is actually system dependant. The chapter on C programming (on page 159) gives the exact method for many different systems.

On a Convex, if the proprietry compiler has been used, the following environment variables must be set before running the program:

setenv FOR005 SYS\$INPUT
setenv FOR006 SYS\$OUTPUT

Compiling csum

Before compiling csum, the necessary links to the include files must be set, as with sum.rpi. This is system dependant, more information on which is given in chpater 8 'Programming in C' on page 159. A list of the various Fortran compilers usable on different systems is given in table 2.1 on page 6. Alternativly, the Fortran to C converter f2c can be used - see section 2.11.6 on page 17, and also pages 5 and 10.

```
dfnrpcgen -n csum.rpi

f77 -c csum_cb_stb.f
cc -c csum_cb.c
cc -o csum_cb csum_cb.o csum_cb_stb.o $DFNRPCLIB/libdfnrpc.a \
   -l... (system dependant linker flags)

f77 -c csum_sb_stb.f
cc -c csum_sb.c
cc -o csum_sb csum_sb.o csum_sb_stb.o $DFNRPCLIB/libdfnrpc.a \
   -l... (system dependant linker flags)

cc -o csum_lb csum_lb.c
```

4 Overview of the DFN-RPC Software

The main features of the DFN-RPC tool are:

- synchronous RPCs
- asynchronous and buffered asynchronous RPCs
- parallel RPCs
- datapipes (message passing)
- stub generator
- global waiting point
- conversion library
- remote file access library
- graphical monitor tool
- integrated server process start
- automatic setup of communication networks between server processes.

This overview introduces the DFN-RPC, and explains aspects of its use:

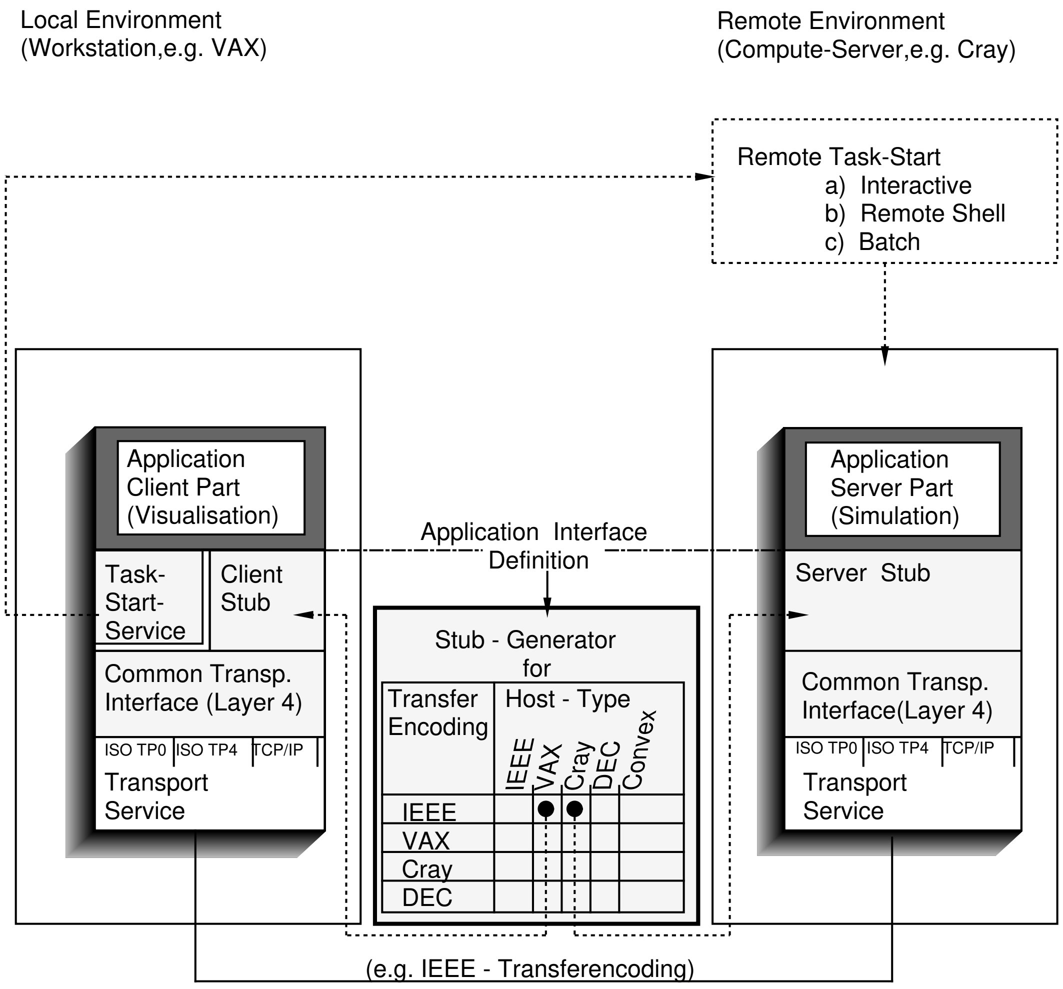


Abbildung 4.1: The DFN-RPC concept; an example scenario

The DFN-RPC modules are shown in *light grey*, the application program sections in *dark grey* and the system services in white.

The user partitions their program (dark grey) into two parts, the main program[1] (*Client Part*) and a sub-program module (*Server Part*). The client calls the sub-programs in the server via a remote procedure call (RPC).

When the client part calls routine xyz in the server part via an RPC, it is actually calling a routine in the *Client Stub* with the same name. This then sends a data packet with the input parameters and an identification for routine xyz, but before being sent, the data is converted into the transfer encoding, which sets the real number format.

The *Server Stub* receives the data packet, allocates memory for the parameter list into which it writes the parameters. It then calls the real xyz routine in the server part. Once the routine xyz has finished executing, the output parameters are sent back to the client stub in a similar fashion by the server stub, the client stub routine xyz then terminates after completion of the data transfer. The client part of the application program can then continue.

Transfer of the data packets is performed by the *Common Transport Interface*. This is a general interface to the different transport services which hides the differences between the various protocols and operating systems, and undertakes the crossover from Fortran to C.

Although the stubs are application specific, they are nonetheless automatically generated by the *Stub-Generator*. This requires an application interface definition which basically contains the parameter lists of the routines which are to be exported to the server part by the RPC. As well as the information already contained within the application program, for example the data type for each parameter, it is also necessary to add some RPC specific information, such as whether each entry is an input or output parameter as the RPC cannot make calls by reference, instead "call by in and out copy". As a result of these optional entries, a considerable increase in the application efficiency can be obtained.

The stubs also undertake number conversion when the participating computers have differing internal number representations. The user can choose the transfer encoding.

For minimal total compute time the transfer encoding format should usually be the same as the workstation number format (IEEE oder VAX), but for compute cost optimization, the number format of the computer server (e.g. CRAY) should be used.

The last module is the *Task start service* . This service enables the user to automatically start the server process(es) without the need to manually log into the compute server. The user can choose any available service, such as remote shell (rsh) or the network queuing service (nqs).

When several servers are used, any communication topology can be setup, for example, ring, grid, torus, cube, hypercube, binary tree or graph. See section 4.1-4.10 and 6.

For applications that are intrinsicly parallel, where the processors represent physical entities or components of a mathematical model, the asynchronous and parallel RPCs, datapipes and the central waiting point allow the RPC user to fully exploit the inherent parallel nature of the program.

The graphical monitor displays the activity of all participating processes, even with complex multi-server applications. See chapter 11.

A remote file access library has also been developed using the DFN-RPC; see chapter 10. Many of the possibilities that the DFN-RPC affords programmers are demonstrated in the examples discussed in chapter 9.

[1] including the sub-programs that are also part of the client.

4.1 Methods of Communication

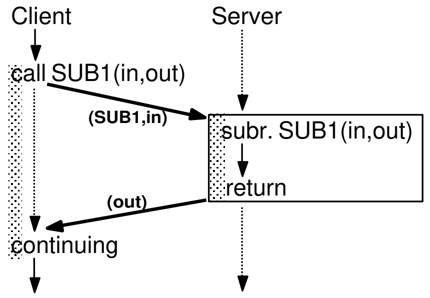


Abb. 4.2: The Synchronous RPC

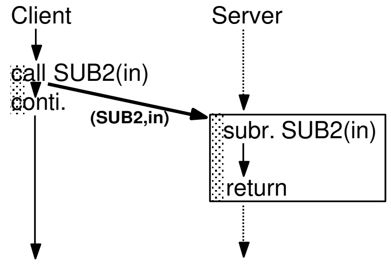


Abb. 4.3: The Asynchronous RPC

The normal, *synchronous* Remote Procedure Call (RPC) is shown in Fig. 4.2. The client shown on the left waits until the server routine (box on the right) is finished. The introductory example on page 19 contains a synchronous RPC.

With an *asynchronous* RPC (Fig. 4.3), the client program can continue as soon as the arguments are sent to the server. However, output arguments can not be returned.

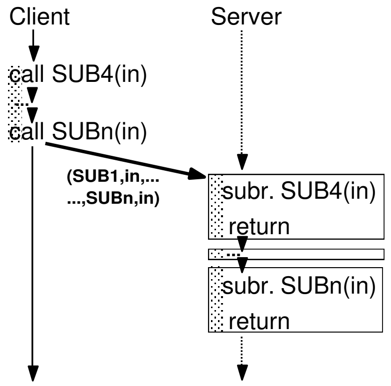


Abb. 4.4: The Buffered Asynchronous RPC

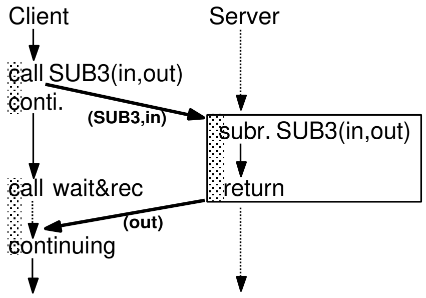


Abb. 4.5: The Parallel RPC

With *buffered asynchronous* RPCs (Fig. 4.4), data packets are sent from the client when the internal buffer is full or when a non-buffered RPC follows. Only after the server has received the data packets will the corresponding routines be executed in the order indicated by the client.

The *parallel* RPC (Fig. 4.5) is a generalization of synchronous and asynchronous RPC's. As soon as the client stub has sent the input arguments, control is returned to the client program. Using inquire and wait routines, the client program can determine when the server routine has finished, after which it can capture the output arguments using a receive routine that writes them directly into the output variables given in the call to the client stub.

With parallel RPCs, it is in principle possible to send the input parameters and receive the output parameters in a parallel thread. In this case, the input variables following the client stub call, and the output variables from the start of the receive procedure to the end of the receive routine are not to be accessed by the application program. Whether the input parameter is sent in a parallel thread or not is set in the interface definition, however, whether the output parameter is received with a parallel thread is set at run time by calling the appropriate routine. These interfaces currently exist, but their operation has not yet been implemented.

All four possibilities require that the RPC *dispatcher* is running in the server process, which receives the RPC's and calls the indicated server stubs, which in turn call the affiliated application server routines. The server routines that are called in a server process are therefore in a common context sequentially executed. Parallelization occurs only between the client and server processes.

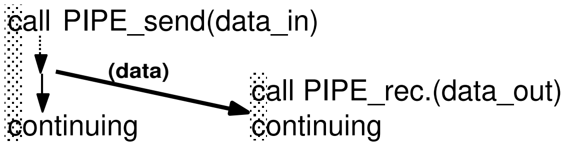


Abb. 4.6: Message Passing using Data Pipes

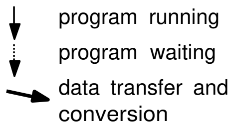


Abb. 4.7:

The message passing principle is realized along with the RPC principle. Using *data pipes*, two applications running in parallel can exchange data. Application programs run in parallel by starting several application processes. The same can be done with RPCs by using parallel or asynchronous RPCs. Datapipes can be used between multiple servers as well as between client and server.

As with RPC's, data type conversion takes place during the transfer if necessary due to different data representation in an heterogeneous environment.

For each data transfer there is an underlying end-to-end flow control. Because of this a sending process will be blocked if the receiver does not accept the data on time.

4.2 Calling Methods in Context

A client can send RPC's to several servers while a server is handling one client. A server can also be a client for other servers.

Regarding *one* client-server pair the following rules apply:

- Asynchronous buffering will be terminated by an asynchronous unbuffered, a synchronous or a parallel RPC.
- If a synchronous RPC or a parallel RPC is received, it must wait until all previous asynchronous server routine calls are processed.
- Receiving the output of a parallel RPC will occur automatically by a following RPC.
- All methods of RPC can also be callbacks.
- However a callback can only be started from a synchronous server routine.
- While using data pipes, obvious deadlocks are partially checked.

If data pipes are to be employed between client and server, then communication must occur between the asynchronously or parallelly called server routine and that part of the client application which follows the RPC call and which is processed before another RPC call or the receiving of output arguments from the parallel RPC to this server.

With data pipes between client and server, the same transport connection is used as for RPCs, but with data pipes between servers, a direct transport connection must be established.

With all forms of RPC synchronization it is guaranteed that the server routines will be called *exactly once* if no errors are reported by the RPC library, including the RPC-finish call at the end of the program, and *at most once* in the event of an error.

4.3 Parallel RPCs

The RPC interface for a subroutine can be defined as a *parallel RPC* in the interface definition. When such a server routine is called via a *parallel RPC*, the client stub passes the control directly back to the application program as soon as the input parameters have been passed to the server process. The client and server process then both execute further in parallel. Naturally, the output parameters of the parallel RPC are not yet available, however the client process can wait until the server has finished executing with the RPC library routine RPOREC, and then receive the output parameters. The values are stored in the same parameters that were passed to the parallel RPC call.

```
            CLIENTPROGRAM                    .
                   |                         .
Appl. is computing |                         . dispatcher
                   |                         . is waiting
              CALL XXX   Input arguments     .
             clientstub-->-->-->-->-->-->-->--serverstub
                return                         |
                   |                           SUBR. XXX
Appl. is computing |                               |
                   |                               |
             CALL RPOREC                           |
                   .                           RETURN
rporec is waiting  .     Output arguments      |
                storing--<--<--<--<--<--<--<--serverstub
                return   (Return package)    .
                   |                         .
```

RPOREC — waits for the results from the parallel RPC if they have not already arrived, and then writes the values in the same variables that were used for the RPC call.

Before the client process calls RPOREC, it can process the connection to the server (or servers) with other RPC library routines:

RPORCT — initiates the receiving of the results from a parallel RPC in a parallel thread[2]. This reception must be finished with a call to RPOREC, but between the calls to RPORCT and RPOREC the application must not use the output parameters. As opposed to RPOREC, RPORCT is non-blocking. RPORCT is implemented in the current release as a dummy routine.

RPOCUR — returns the object identifier of the current server i.e. the *current object*. Additionally for server processes, it returns the object identifier used for addressing the client.

RPOUSE — switches between the server processes, i.e. the object given for *current object* in the parameter list becomes the current server.

RPOCRY — waits until the current server sends either the return packet for a parallel RPC or a data packet (see under datapipes). The maximum wait time can be set within the range 0 and infinite milliseconds.

RPOLRY — is similar to RPOCRY, except that a list of server objects can be given.

[2] Not possible when the operating system doesn't support threads

RPORRY	similar to RPOLRY except that an additional list of file descriptors can be given to select when data is ready for a read operation. Using this routine, one can simultaneously wait until a parallel RPC is finished, a data packet has arrived, an input has taken place on stdin or an X-Window event has happened in a display. The Unix file descriptor must be known; 0 for stdin and the connection number (XOpenDisplay()) for the X-Window display.
RPOGRY	as RPOLRY, except that additional sockets can be given. The complete functionality of the Unix system call *select* is available; see section "'Select(2)" on page 150.
RPOGSD	determines the file descriptor of the DFN-RPC connection with the current server. This file descriptor is required when the application program uses other wait utilities or the system call *select*.
RPOISP	waits until a packet arrives, and determines whether it is a data packet, or a return packet from an RPC.
RPD..S/R	send/receive a data packet (see data pipes).

Also see example *para* on page 164.

4.3.1 Interface Definition for Parallel RPCs

For parallel RPCs, the synchronisation qualifier **PARALLEL** must be given with C*[RPI_BEG] in the RPI file (stub generator input) or in the ENTRY statement.

The entry **PARALLEL, INPUTTHREAD** causes the client stub to transfer the input parameters with an additional thread. The application must not use the input parameters until the routine RPOREC has finished. (The entry INPUTTHREAD is not currently evaluated.)

4.3.2 Further Details

Parallel RPCs have the following properties:

1. During a parallel RPC (i.e. after the call but before a successful RPOCRY), no further RPCs to the same server process (= object) are possible.
2. If the application attempts to start another RPC to the same server, then the client stub automatically calls RPOREC.
3. Callbacks are only possible from synchronous routines, i.e. server routines that are called via asynchronous, asynchronous buffered, and parallel RPCs cannot make callbacks (RPCs back to the client).
4. Limitations:

   Parallel RPCs cannot be used for functions as later utilization of the function value is not possible.

   Alternative returns are not possible with parallel RPCs.

   If externals are passed as parameters, care must be taken to ensure that these aren't realised as callbacks to the client.

   Parallel routines may have a maximum of 30 parameters only, whereby string parameters count as two parameters. (It is possible however, to make each parameter a multi-dimensional array of any size!)
5. The addresses of all parameters to a parallel RPC must still be valid when the respective RPOREC is called.

   As RPOREC writes the resulting values into variables that are not in its parameter list (they

are the variables from the parameter list of the send-stub), it must be so embedded that this side effect is possible. If care is not taken, the allocation of output variables in registers can lead to false results. There are several methods to prevent this:
a) All output variables are placed in a common block.
b) RPOREC is called explicitly (not implicitly through the stub of a following RPC) **and** the output parameters are placed at the end of the parameter list when calling RPOREC. The disadvantage here is that the compiler possibly warns that the RPOREC routine is called with a different parameter list.
c) On most workstations, the compiler allows one to declare the output variables as VOLATILE, however on one workstation, a test program showed that VOLATILE does not correctly work with every compiler optimization level.
d) The SUPPRESS compiler directive in the CRAY cf77 compiler should not be used, as it only works within connected execution paths, which is normally not the case.
e) Turn off all compiler optimization (not 100% certain).

Additionally, the output variables should be declared with SAVE if they otherwise would be deallocated between the client stub call and the RPOREC call.

4.4 An Example with Dynamic Load Balancing on Several Servers

The example shown below is to be parallelized on a RISC-cluster.

```
program simul
...
do while (¬ sim_fin)
  do i = 1,n_parts
    call simul_part(inp(i),out(i))
  end do
  sim_fin = end_criterion(out)
  if (¬ sim_fin) then
    inp = new_input(out)
  endif
end do
```

Fig. 4.8: The simulation computations in the inner loop are independent of each other and may therefore be parallelized. simul_part includes the compute-intensive simulation code of a simulation area. The outside loop implements the data exchange between the separate simulation parts using new_input.

The following program demonstrates how this application can be parallelized using DFN-RPC. The exchange of data between servers occurs here via the client process inside the routine new_input. A dynamic load balancing between the servers occurs if the number of servers is fewer than the number of parallel calls in the inner loop (n_parts).

```
C*[BLK_BEG] cb,     NAME='client block', CLIENT_SITE
C*[MOD_USE]         cm, LOCAL
C*[MOD_USE]         sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb,     NAME='server block', SERVER_SITE
C*[MOD_USE]         sm, SERVER
C*[BLK_END]
```

Here the stub generator is instructed to split up the source and to generate the necessary stubs. In 'client block' the client portion of the application and the client stub of the module sm are combined together. In 'server block' the routines themselves and their server stubs are positioned.

```
C*[MOD_BEG] cm,      NAME='client module - main'
      program simul
      ...
```

Client portion of the application.

```
      call rpquci(n_server, obj_id, iret)
```

Server initialization.

```
      do while (¬ sim_fin)
        do i_call=1, min(n_server,n_parts)
          call rpouse(obj_id(i_call), iret)
          call simul_part(inp(i_call),out(i_call))
        end do
```

In every server – selected by rpouse – the simulation routine simul_part is started using parallel RPC's.

```
        do i_rec = 1, n_parts
```

Result collection and further simul_part

```
        timeout = 60000 ! milliseconds                    calls: rpolry determines which
        call rpolry(n_server,obj_id(1:n_server),          parallel RPC is finished. With rpouse
             timeout,objrdy(1:n_server),obj1st, iret)     the corresponding server is selected
        call rpouse(obj_id(obj1st), iret)                 and with rporec the output arguments
        call rporec(iret)                                 are written into corresponding out(i).
        if (i_call < n_parts) then ; i_call=i_call+1      The server is immediately used for
          call simul_part(inp(i_call),out(i_call))        the next run of simul_part,
        endif                                             through which the automatic
      end do                                              load balancing is achieved.
      sim_fin = end_criterion(out)
      if (¬ sim_fin) then                                 Compute the new input data
        inp = new_input(out)                              and termination criteria, etc.
      endif
    end do
    call rpqucf(iret)                                     All servers are terminated.
    end
C*[MOD_END]
C*[RPI_NON]
C*[MOD_BEG] sm,      NAME='server routines mod.'          Server portion of the application.
    program exasrv                                        Special main-program for the
    character*100 client_addr                             server with rpqusp, conterpart
    call getarg(1,client_addr)                            of rpquci and rpqucf.
    call rpqusp(client_addr,iret)
    end
C*[RPI_BEG] PARALLEL                                      Parallel RPC Interface Definition,
    subroutine simul_part(inp,out)                        composed of the subroutine header,
    real inp [IN], out [OUT]                              extended by RPC-specific qualifiers
C*[RPI_NON]                                               in square brackets, e.g. [IN]
    out = simulation_code(inp)                            The subroutine body is without
    end                                                   meaning for the stub generation.
C*[RPI_END]
C*[MOD_END]
```

Fig. 4.9: Example simul with parallel RPC.

The stub generator separates the software in the client and server portions and generates the necessary stubs. The interface definition, the client and the server software can also be stored in different files. The dynamic load balancing demonstrated in the example requires that no data has to be exchanged between the servers.

Fig. 4.10 shows an example of a data base for server initialization using rpquci.

```
# Server database for program simul
rsh host_a -l my_username -n simul_sb DFNRPCADDR &
# The following host has 2 processors
rsh host_b -l my_username -n simul_sb DFNRPCADDR &
rsh host_b -l my_username -n simul_sb DFNRPCADDR &
# The following host does allow a telnet session only
echo "'Please start 'simul_sb DFNRPCADDR' on host_c by telnet"'
# n_server in the call of rpquci will be reduced to the number of servers above
-norepeat
```

Fig. 4.10: Example of server initialization using rpquci.

See also examples rshexa and crshexa on page 166.

4.5 Data Pipes

Data pipes can be used to exchange data between two objects that already have an existing transport connection. Between the client and server of a DFN-RPC application, the established RPC transport connection is used, as shown in figure 4.11. The RPC library routines RPOSSC and RPOC2S can be used to establish a transport connection between two servers, see figure 4.12.

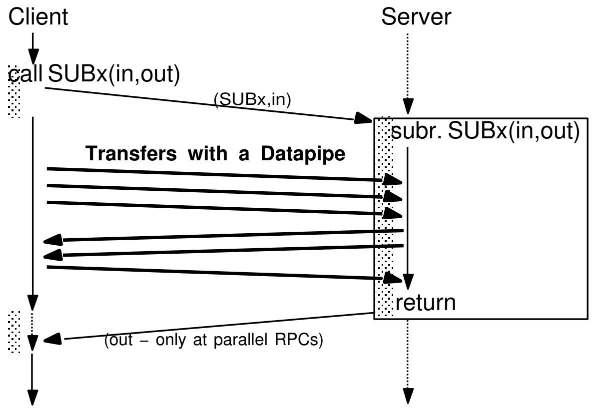


Abb. 4.11: Client-server data pipe

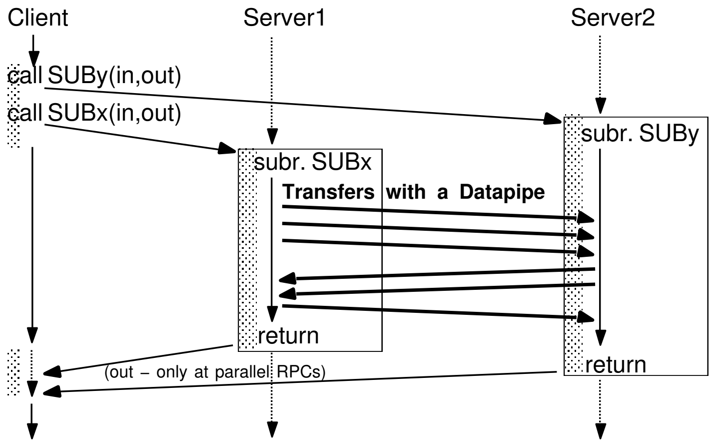


Abb. 4.12: Server-server data pipe

4.5.1 Data Transfer Using Library Routines

The following RPC library routines can be used to send and receive data packets:

RPDI0S	sends an integer array
RPDI0R	receives an integer array
RPDI2S/R	sends/receives an integer*2 array
RPDI1S/R	sends/receives a byte array
RPDL0S/R	sends/receives a logical array
RPDL2S/R	sends/receives a logical*2 array
RPDL1S/R	sends/receives a logical*1 array
RPDR0S/R	sends/receives a real array
RPDR8S/R	sends/receives a real*8 array
RPDD0S/R	sends/receives a doubleprecision array
RPDC0S/R	sends/receives a character string

The read routines RPD..R are blocking, i.e. they wait until a data packet arrives.

Using a flush parameter in the send routine, one can decide whether the message is to be buffered or not, or if the buffer should be sent at the end of this message.

The data pipe message transfer uses end to end flow control; the send routine waits if necessary until all data is buffered at least in the send side, or with a large amount of data, until part of the data has also been read by the appropriate receive routine on the receive side. The receive routine waits until all data has been successfully received. The application must consider that the two objects exchanging data packets are possibly synchronized, i.e. the sending object will be blocked until the receiving object is ready to receive the data packet. The application programmer must therefore ensure that no deadlock occurs.

There is no partitioning or re-assembly of the transmitted data; the RPD..R receives the array exactly as it was sent by the RPD..S routine.

With the parallel RPCs RPOCRY, RPOLRY, RPORRY and RPOGRY described above, the program can wait on one or more connections until a data packet has arrived. The routine RPOISP can then be used to determine the packet type.

Data pipes require two processes running in parallel, therefore it is not possible to test data pipes with a locally linked program (local block).

Data pipes can be used in cooperation with DFN-RPCs, whereby the parallelization is made possible through asynchronous (unbuffered!) or parallel RPCs. It is also possible to use data pipes independently from DFN-RPCs.

For further info. see section 4.6 on page 39 and examples *para* on page 164 and *pipe* on page 167.

4.5.2 Application Specific Message Stubs

The routines described above are so called *message stubs*, delivered as part of the DFN-RPC package.

It is possible however to use self-defined message stubs. In the interface definition, a message stub interface is defined which is then used to generate a send stub (all parameters are treated as input) and a receive stub (all parameters are output parameters). The stubs are given names by appending an "S" or an "R" on the end of the name given in the interface definition. There is no Fortran body in the interface definition, and no Fortran code is written to the .f files. The send stub places a header on the message, which is checked in the receive stub.

Message stubs are generated by defining a subroutine interface, and by using the keyword **PIPE** in the C*[RPI_BEG] statement. If the module is created as a client module in a block, then a message send stub is generated, and in the server module, the respective message receive stub (also see MOD_USE).

If the keyword **BACK** is given (with comma separator), then the message receive stub is placed in the client module, and the message send stub in the server module.

With the keyword **BOTH**, both stubs are created in the client and server modules.

Synchronization entries don't have any meaning in this context, so are not allowed. A header is always generated for the start of a message containing an operation code. This can be specifically given with the entry CODE=`<opcode>` . The Fortran subroutine header given in the interface definition is not written to any .f files.

Also see the example *rpcpipe* on page 165.

4.6 Datapipes and RPCs

Every communication connection established with the routines RPQ..., RPO1/2/3.., RPOPOP, RPOACC, RPOCON or RPOC2S can be used for RPCs as well as data pipes.

While a connection is being used by an RPC (i.e. a block dispatcher is running at one end), it can also be used by a data pipe as long as the RPC is asynchronous (unbuffered) or a parallel RPC.

Changing its use between RPCs and pipe routines is also possible. The routine RPOFBD (*finish block dispatcher*) can be used by the client to terminate the block dispatcher in the server, thereby allowing both processes to use the connection as a data pipe. Following a call to RPBD1 by the server, the client can send further RPCs.

The server process can also use RPOISP (*inspect package*) to obtain the type of the incoming packet, and then call the appropriate pipe routine or RPBD1P (*block dispatcher for one rpc*). (RPBD1A cannot be used for this purpose as it expects that the connection is used exclusively by RPCs. RPBD1A was developed for programming servers that serve several clients.)

See also example *rpcpipe* on page 165.

4.7 The FLUSH Argument

With asynchronous RPCs and data pipe send routines, the user can determine at runtime whether the call is to be buffered or whether it is flushed directly after the call.

This is set by either using a formal LOGICAL parameter with the [FLUSH] qualifier, or by giving **FLUSH** as a qualifier_expression.

4.8 Object Handling

4.8.1 Definition

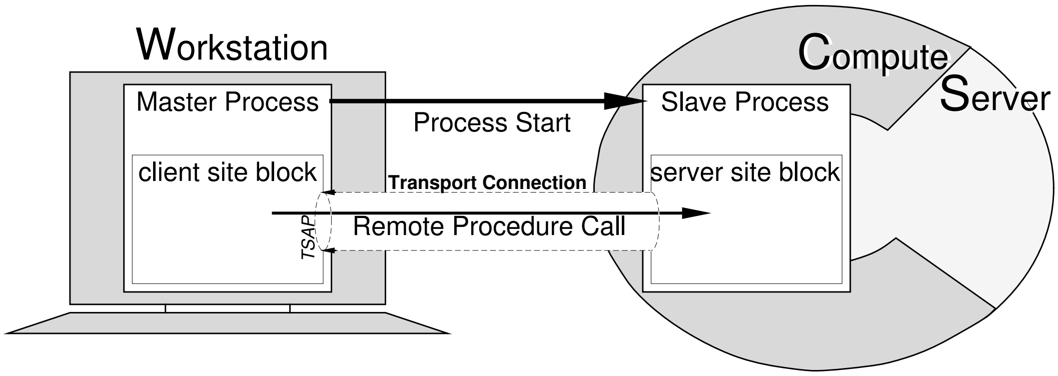


Abbildung 4.13: Workstation/Computeserver, Master/Slave and Client/Server - first scenario: Master contains the client_site block

Processes are considered as *objects* by the DFN-RPC. Object Handling is used to start the remote process. The process started by the user is called the *master process*, while the remote process started via object handling is called the *slave process*. Typically, the master process runs on a workstation and the slave process on a compute server.

The master process can contain either the client_site block or the server_site block, i.e. master/slave and client/slave are independent labels for the DFN-RPC. Thus the application program can be placed in the master process (workstation in figure 4.13) or in the slave process (compute server in figure 4.14).

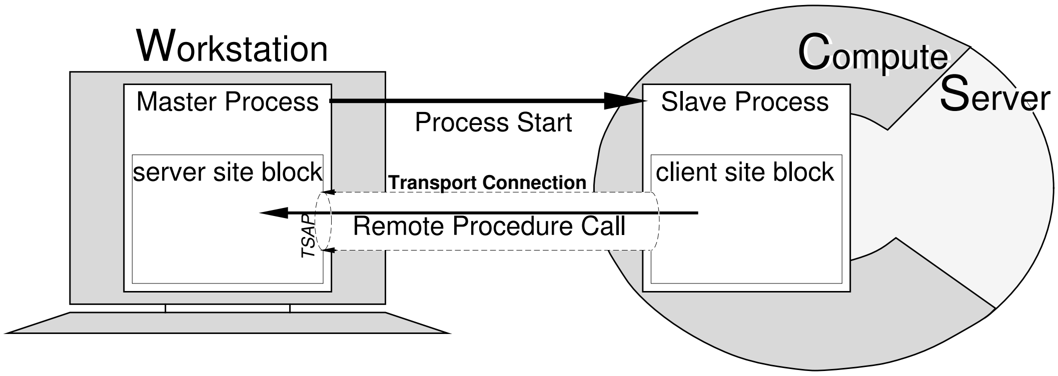


Abbildung 4.14: Second scenario: Master contains the server_site slock

To execute a remote procedure call, a so called transport connection is necessary[3]. This are setup by the object handling routines. This connection setup is directional, that is one side (normally the master) creates a transport service access point (TSAP) and the other side (slave) establishes the connection, i.e. executes the connect. A TSAP is a sort of virtual plug with a name. To make this connection, the slave must know the address of the computer (normally the hostname) on which the master is running, as well as the master's TSAP name. With an automatic process start using the object handling, this information is passed from the master to the slave.

4.9 Process Start and Connection Setup

4.9.1 The Quick Start Routines

With the quick start routines, the master contains the client_block as shown in figure 4.13, the one exception being when there is only one server.

To make the startup of parallel applications as simple as possible, the client routines **RPQUCI** and **RPQUCF** plus the server routines **RPQUSP** or **RPQUSI** and **RPQUSF** were written.
The user can choose to start the server process by hand or automatically. Passing the port address information required to setup the connection can also be done either automatically or manually. The client calls RPQUCI with the number or processes as one of its arguments. If the file **dfnrpc.servers** exists, then each line in the file is executed with the CALL SYSTEM command. Below is an example file:

```
# Servers startet with a remote shell:
rsh servername_1 -n server_program DFNRPCADDR &
rsh servername_2 -n server_program DFNRPCADDR &
rsh servername_3 -n server_program DFNRPCADDR &
rsh servername_4 -n server_program DFNRPCADDR &
```

[3] All information required for an RPC is passed via this transport connection, for client_site block to server_site block calls and also for callbacks. Callbacks are RPC calls made from the server_site block back to the client_site block. These are not shown in the above figures.

```
# The following server must be started with telnet:
echo start anywhere server_program DFNRPCADDR
# The following line prevents, that the server list is
# repeated to start the number of servers wanted by the client
-norepeat
```

Figure 4.15: Example dfnrpc.servers file for RPQUCI

The keyword DFNRPCADDR is replaced by the address of a newly opened port before the SYSTEM call. The server_program passes this string to RPQUSP which then establishes the connection with the client. If the keyword DFNRPCADDR is not included, or the the file dfnrpc.servers cannot be found in the current directory (i.e. RPQUSP passes an empty string), then the server process starts but passing the port address to the server must be done interactively. The *reverse connection setup* used in the quick start routines is described in [17] Figure 4.5, on page 24.

The examples *rshexa*, page 166 and *crshexa*, page 166 show the use of these routines in both C and Fortran programs. In the examples *ben* and *test*, their use can be chosen, and in the examples shown in figures 4.9 and 4.10 their use is also demonstrated.

If the servers process is not intended simply to process RPC calls, but also for example, to work independently and communicate via data pipes with the client, then RPQUSP cannot be used as it calls the block dispatcher RPBD1 once only. Instead, the routine RPQUSI is to be used for the server start, following which the server can directly call the block dispatcher or the data pipe routines. The connection to the client is terminated by a server call to RPQUSF. This mechanism is shown in the example *rpcpipe* (see page 165).

If only one server process has been started with RPQU..., then the client and server can swap their roles once the routines RPQUCI and RPQUSI are finished. For this to work, the client process (which is actually the master process) has to call the block dispatcher RPBD1.

4.9.2 Interconnection Network

Freely configurable interconnection networks between servers can be setup using the routines RPQSSC and RPQSSS (see section 6.4.27 on page 128). A list of server-server connections is passed to the client routine RPQSSC. The complimentary server routine RPQSSS must be called in every server process, which then sets up all connections required for each server and passes back a list of the application parts via its parameter list. In the example sscon (see section 9.8, on page 168) an interconnection network with nine servers is setup. Also see the example on page 130.

4.9.3 Explicit Connection Setup

The object start routines RPQU... combine the process start and connection setup. However, using the routines RPOPOP, RPOCON and RPOACC, it is possible to establish connections independently of the process start.

port open	**RPOPOP** opens a port (also called TSAP) in a process (*Master* in the following).
connect	**RPOCON** establishes a connection with this port from another process (*Slave* in the following).
accept	**RPOACC** is used by the master to accept the connection.

RPOPOP passes back the complete address of the opened port (*master_address*), which the application must then pass to the slave_process. This information is required for the call to RPOCON. The *master_address* can be given, for example, as a command line argument for the slave process using GETARG (3F). RPOCON and RPOACC verify the connection with a two-way information exchange, which also serves to synchronise the two processes. The connection can now be used just as if it had been setup with RPQUCI or RPQUSI.

RPOCCL and RPOPCL are used to terminate the connection.

close connection	**RPOCCL** closes the connection. It should be called from both master and slave, and may be called directly after the last data packet has been sent. The connection is not closed until RPOCCL has been called at both ends.
port close	**RPOPCL** must be called by the master to close the port. RPOPCL is the counterpart of RPOPOP.

When using data pipes, the following points should be noted:

1. A maximum of 512 object identifiers can be given in one process. These are given during the start of a server object or during the initialization of a data pipe.
2. A maximum of only 512 transport connections can be used (except with sunlink or sunnet OSI where the maximum is 10).
3. With a call to RPSINI, the sum of the required incoming and outgoing connections may not be greater than 512.
4. The object start routine and the pipe start routine RPOC2S then setup the connections. Therefore, the values given for the number of connections in RPSINI should not be too small.
5. When stopping a server, care should be taken when calling RPOUSE that on both the client and the server side the respective partner is set as the *current object*.
6. A dispatcher started by the server must be stopped by the client with a call to RPOFBD before the connection can be terminated with RPOCCL and RPOPCL.

See example *con* on page 165.

4.10 Addressing Different Server Objects

4.10.1 Object Identifiers

Objects are independent processes in the DFN-RPC concept. Connections can exist between objects, which for a TCP/IP connection requires the computer name and port address. When a connection exists between two objects, then it is given an object identifier by both ends. Because the connection is always made with another object, the object identifier of the remote object is used to identify the connection.

```
        Object A                                  Object B
+----------------------+                +----------------------+
|                      |                |                      |
|        +------------|                |------------+         |
|        | ObjId of B |    Transport-  | ObjId of A |         |
|        | valid in A |================| valid in B |         |
|        +------------|    Connection  |------------+         |
|                      |                |                      |
+----------------------+                +----------------------+
```

The object start routines and RPOC2S return the respective object identifier. The object identifier is an output parameter from the object start routines RPQUCI, RPQUSI and RPO1/2/3.., and the connection routines RPOPOP, RPOCON and RPOC2S.

Care must be taken with RPOC2S to ensure that the client knows the valid object identifier of any server-server data pipes. It can then pass the identifier to application routines within the server via an RPC, thereby allowing them to use the data pipe.

4.10.2 "Current Object and "Current Client Object"

RPC and pipe calls always refer to the so called *current object identifier*. The DFN-RPC library contains an internal global variable called the *current object identifier*, which is given a new value during an object start or the making of a connection. RPOUSE can be used at any time to change the variable's value. Its current value can be determined with RPOCUR.

For receiving RPCs there is another global variable called the *current client object identifier*. Following the start of the block dispatcher, the value of the *current object identifier* is stored in the *current client object identifier* and used until the client calls a finish routine RPQUCF, RPO..CF or RPOFBD. At the program end, it sets the *current object identifier* as the *current client object identifier* so that the following RPO..SF routine terminates the connection with the correct object.

4.11 Navigation Through the DFN-RPC Software

The DFN-RPC directory tree contains the following relevant directories and files:

```
Directory:                                            Files:

$DFNRPCROOT / dfnrpc / 1.0 /                        / dfnrpc_set
                                                      dfnrpc.ps

                             src / examples / ua1a / source_makefile
                                                   / ua1a.rpi
                                                   / ua1a_cb_stb.m4
                                                   / ua1a_sb_stb.m4

                                        ... / ben  / source_makefile
                                                   / ben.rpi
                                                   / ben_cb_stb.m4
                                                   / ben_sb_stb.m4

                       .../sys_debug  or
                       .../sys
                          .../sun4    or others
                              .../ bin             / dfnrpcgen
                              .../ lib             / libdfnrpc.a
                              .../ include         / dfnrpc.inc

                              .../ examples / ua1a / makefile
                                                   / ua1a.rpi
                                                   / ua1a_lb.f
                                                   / ua1a_lb
                                                   / ua1a_cb.f
```

```
                                           / ua1a_cb_stb.m4
                                           / ua1a_cb_stb.f
                                           / ua1a_cb
                                           / ua1a_sb.f
                                           / ua1a_sb_stb.m4
                                           / ua1a_sb_stb.f
                                           / ua1a_sb
                                           / dfnrpc.inc -> ...

                                ... / ben  / ben_cb
                                           / ben_sb
                                ... / test / test_cb
                                           / test_sb
```

If the subdirectory .../sys/examples...is empty, then the examples have not been installed.

The source files are delivered without internal documentation or comments. They are required for the compilation during the installation, but do not provide any basis for support of this software. The following shell script helps with navigation:

```
            Module description of the dfnrpc_set
            ====================================

Installation of dfnrpc_set:
---------------------------
   You should include in your .cshrc file the following lines:
       setenv DFNRPCROOT ~
                          (or where you have installed the dfnrpc.)
       setenv DFNRPCREL   1.0
       setenv DFNRPCTRANS ieee
       source $DFNRPCROOT/dfnrpc/$DFNRPCREL/dfnrpc_set
                          (Before calling the following configuration
                           you can change the defaults in
                           DFNRPCROOT, DFNRPCREL and DFNRPCTRANS.)
       SUN4               (or an other configuration, see below.)
   Available configurations with optimized executables:
    ALLI ALPHA APO CVX CRAY2 DEC FUJITSU HP IRIS3 IRIS4 PARAGON
    RS6 SUN3 SUN4 VAX
   and with the UltraNet Compatibility Library:
    CVXU CRAY2 IRIS4U SUN3U SUN4U
   and  with OSI:
    CRAY2 SUN3O SUN4O
   All configurations are also available compiled with debug option:
    SUN3D SUN4D CRAY2D ...
   CAUTION: The values of DFNRPCROOT, DFNRPCREL and DFNRPCTRANS must
            not be followed by any blanks or tabs.

Usage:
------
   After calling a configuration (like SUN4) there are available
   several aliases and setenv's:
   - general aliases                         and belonging setenv
       alias  dfnh  typing_this_information
       alias  dfns  cd general_source_directory       $DFNRPCSRC
```

```
    alias  ct4s  cd source_of_ct4
    alias  rpcs  cd source_of_rpc
    alias  gens  cd source_of_gen
    alias  ipsms cd source_of_ipsm
    alias  makes cd source_of_makefiles
    alias  exams cd general_examples_source_dir
    alias  dfnc  cd general_config_directory      $DFNRPCCONF
    alias  dfnb  cd configuration/bin             $DFNRPCBIN
    alias  dfnl  cd configuration/lib             $DFNRPCLIB
    alias  dfnld cd configuration/lib_debug
    alias  dfni  cd configuration/include         $DFNRPCINCLUDE
    alias  exab  cd general_examples_bin_dir      $DFNRPCEXAMPLES
- for each example XYZ
    alias  XYZs  cd source_of_example_XYZ         XYZs
    alias  XYZb  cd bin_of_example_XYZ            XYZb
  with XYZ = awi,ben,cav,con,crshexa,csum,cvttest,exa,moni,multicl,
             para,ph1,ph2,phi,pipe,rfa,rpcpipe,rshexa,rw,sscon,sum,
             test,ua1a,ua1b,ua1c,ua1d

Depending on your installation, $DFNRPCEXAMPLES and all example
binaries XYZb refer to your default transfer type.
With the aliases
    TRANS_ieee
    TRANS_dec
    TRANS_vax
    TRANS_cray
you can change to that transfer type, i.e. you change the
values of XYZb.
```

4.12 Details of Implementation

With DFN-RPC marshalling the data is copied from the application's memory into a transfer buffer without any further buffering in between. During the copy the data is converted if required. The content of the transfer buffer is written directly in the TCP BSD-Socket using *write*. With unmarshalling the data is read into the transfer buffer using *read* and copied (and reconverted if necessary) from there directly into the memory of the application. By selecting a suitable buffer size, the number of system calls is reduced to a minimum and a maximum data throughput is achieved. With DFN-RPC no further daemon or agent processes are required for the communication between the processes of the distributed application. Daemons are only required when the monitor is turned on. Chapter 12.4 clearly demonstrates that this design provides for a better transfer rate and less delay in comparison to DCE, ONC and PVM. DFN-RPC speed approaches nearly that of a pure TCP-IP socket connection. To achieve higher speeds is it necessary to use a more suitable transfer protocol and implementation, as for example in the Peregrine RPC System [5].

Within one process only **one** transfer buffer is normally set up for marshalling and unmarshalling purposes. Only when operations on several processes of the application are buffered is it necessary to set up further transfer buffers.

If an asynchronous RPC or a data pipe send is unbuffered, then a partially full transfer buffer is sent with *write* at the end of the client stub. Otherwise it is filled through further RPC or pipe calls.

Every *read* operation is blocking. The *write* operation can also block through end-to-end flow control.

Before receiving the output data of a parallel RPC or of a data pipe, using an unblocking inquiry one can query the status of one or more RPC and data pipe connections, or one can wait until the next packet has arrived or a timeout period given by the application has elapsed. Also, the list of transport connections can be completed by further Unix filedescriptors. A global waiting point can be set on which all communication paths of the application wait for the next incident. This is accomplished internally using *select*.

During implementation the number of system calls is held to a minimum in order to achieve the required functionality. Data conversion is programmed in vectorizeable loops.

A distinct improvement in performance can therefore be achieved simultaneously with more functionality in relation to the parallelization and distribution of the application programs in comparison to DCE-RPC, SUN-RPC and PVM.

4.13 Data Conversion Library

To enable data exchange between two computers with different number formats, the data must be converted during transfer.

For flexible conversion of different number formats and for problem free transfer the following points should be noted:

The local data on the sending computer is converted from its local format to a transfer format which the receiving computer may read directly. If it cannot read the transfer encoding format, then the data is converted on the target computer to its local data format.

The data to be sent is passed to the conversion routine RPVCSB together with the length of an array element in bytes. All currently used lengths of INTEGER and REAL numbers can be converted. To specify which array elements should be converted, two arguments containing the start and end array indexes of the required section are passed.

The conversion result is placed in a send buffer, which is actually a byte array to allow simple data transfer. To specify the length of the array elements (in bytes), an additional argument is used, valid values for which are INTEGER numbers with 1, 2, 4 or 8 bytes and REAL numbers with 4, 8 or 16 bytes in length. Options for the transform representation are INTEGER BIG or LITTLE ENDIAN, LOGICAL BIG or LITTLE ENDIAN and for REAL numbers, IEEE, CRAY, DEC or VAX formats.

The length of send buffer to be used (which must be in 16 byte sections) and the length of the send buffer (in byte steps) that has already been filled is passed to the conversion routine. Following a successful conversion, the latter parameter is passed back containing a new value. Additionally, a parameter is passed to the send routine which contains the last position (in byte steps) of the send buffer for which the contents have already been sent. This makes it possible for the conversion routine to shorten the send buffer by the amount that has already been sent, thereby allowing the send routine to transfer of sections of the send buffer.

Before the conversion of the data into the send buffer is started, it is shortened by the amount already sent, and is then aligned to the next whole value of the data type currently being dealt with. For example, if the current position is at byte 19 in the array, and the current data type is 8-byte real numbers, then the position is set to byte 24.

When using the conversion routine as described above, various problems can arise as discussed below:

- If the send buffer is too small to allow conversion of all data, then conversion continues until the buffer is full. Complete filling with enough data is always possible by choosing an appropriate send buffer length (in steps of 16 bytes).
- If an under- or overflow is detected during conversion (i.e. the value is too small or large to convert), then the conversion is stopped and the routine exited. The under- and overflow checks can be disabled by setting the appropriate flag, and it is also possible to enable an automatic underflow correction (which sets the value to zero).
- If a mistake is made when setting the argument values, e.g. the given number of bytes for a send buffer element is not possible, then the routine exits before starting the conversion, returning an appropriate error value.

There are two additional routines available on the send-side; a routine to initialize the send buffer (RPVISB) and a routine (RPVFSB) to align the send buffer on a 16 byte boundary, improving the buffer transfer.

On the receiving computer, the process described above is reversed, i.e. the data is converted from the transfer representation into the local format.

The received data must be passed to the de-conversion routine RPVCRB as a byte array receive buffer. The actual buffer length (in bytes) must be passed to the routine, also the number of bytes in the receive buffer that have already been processed. These two variables give the user the ability to start converting data that has arrived before the data transfer is complete.

If possible, the buffer is shortened by 16 byte sections following the successful conversion of a section, i.e. the processed data at the front of the array is overwritten by un-converted data further down the array. The parameters describing the buffer length and the current conversion status are appropriately updated.

The size (in bytes) and the type of the elements in the receive buffer must be passed to the de-conversion routine as described for the conversion routine.

Additionally, the data array in which the converted data is placed must also be declared as with the conversion routine, i.e. the following parameters must be passed:

- The required length (in bytes) of an array element.
- The starting index, from which the conversion starts.
- Lastly the end index, after which no more converted data may be written to the array.

As long as the start index does not reach the end index, and the receive buffer is not empty, the conversion continues (or starts). It may be necessary to make a new call to the de-conversion routine if a new set of data arrives.

The flags for control of the under/overflow checking and correction are also present in the de-conversion routine.

Two other routines are available on the receiving side; RPVFRB which aligns the receive buffer on 16-byte boundaries, and RPVIRB which shortens the receive buffer as is done for the send buffer.

The range for the normalised real numbers $(-1)^s * 2^e * m$ on different computer architectures is given by:

- Sign $s = 0$ oder 1
- Mantissa $1 \leq m \leq 2$

- Exponent e:

Computer architecture	REAL*4	REAL*8	REAL*16
ieee	-126..127	-1022..1023	-16382..16383
dec	-126..127	-1022..1023	-16382..16383
vax	-128..126	-128..126	-16384..16382
cray	-8190..8189	-8190..8189	-8190..8189
ibm	-260..251	-260..251	-260..251
convex (ieee)	-126..127	-1022..1023	..
convex (native)	-128..126	-1024..1022	-16384..16382

The routines are described in section 6.4.37 “RPV... – Conversion Routines” on page 137.

5 The RPI Definition Language

The *Remote Procedure Interface Definition File* (**RPI-File**) contains the input data for the stub generator, written in the Fortran RPI Definition Language (F-RPIDL). A single RPI file can contain all the information necessary for the stub generator to create the Fortran code and stubs for all objects (processes) in a distributed application (i.e. the single source file principle).

Alternatively, the RPI file may contain only a part of this information:

1. Only the RPI definition without the Fortran source code (see section 5.4 on page 80).
2. Information for one block only or one module only (see below), so that the stub can be generated a module at a time (see section 5.5 on page 80).

5.1 The Block and Module Concept

To enable separate compilation of RPC stubs and server routines, a four layered concept has been developed:

- Process
- Block
- Module
- Subroutine

If one wants to distribute a program (local block) that contains the modules CM, SM and CX, so that the module SM will be executed on a remote computer (as shown in figure 5.1 on page 50), then the following steps must be taken:

- Instead of having one process, the program must be divided into a master and slave process. The use of the terms *master* and *slave* implies that the former is responsible for starting the latter, which for this example simply means that the master asks the user to manually start the slave.
- A *block* is a program that is run as one of these processes. A block may contain one or more modules.

  The *local block* is the undistributed program, whereas the terms *client block* and *server block* are used to describe the distributed program. Remote Procedure Calls are issued from the client block (except with a callback - see below).

  Every block has a *block_name* and *block_longname*. The block_name is used internally to address the block, and is also used in the generation of filenames for block code, so it should be a short or a mnemonic name. The block_longname can be a more complete description.
- A *module* is the logical collection of a group of subroutines and functions. In the non-distributed case (i.e. in the "local block"), the module is simply a collection of subroutines. When a program is distributed, the situation is as follows:
  - The module CM in figure 5.1 contains only the routines known to the client block, the program's *MAIN* routine being perhaps the most important of these.
  - The client block module CP is required for distributed computing, containing routines for *starting the slave process* (see also *object handling* and *object start*).
  - The server block module SP is also required for the distributed computing. It is CP's counterpart, and its main function is to hold the server block main program. Application module SM runs in the server block, so the client block has a client module SM containing the *client stubs* for the subroutines in SM. The server module SM in the server block

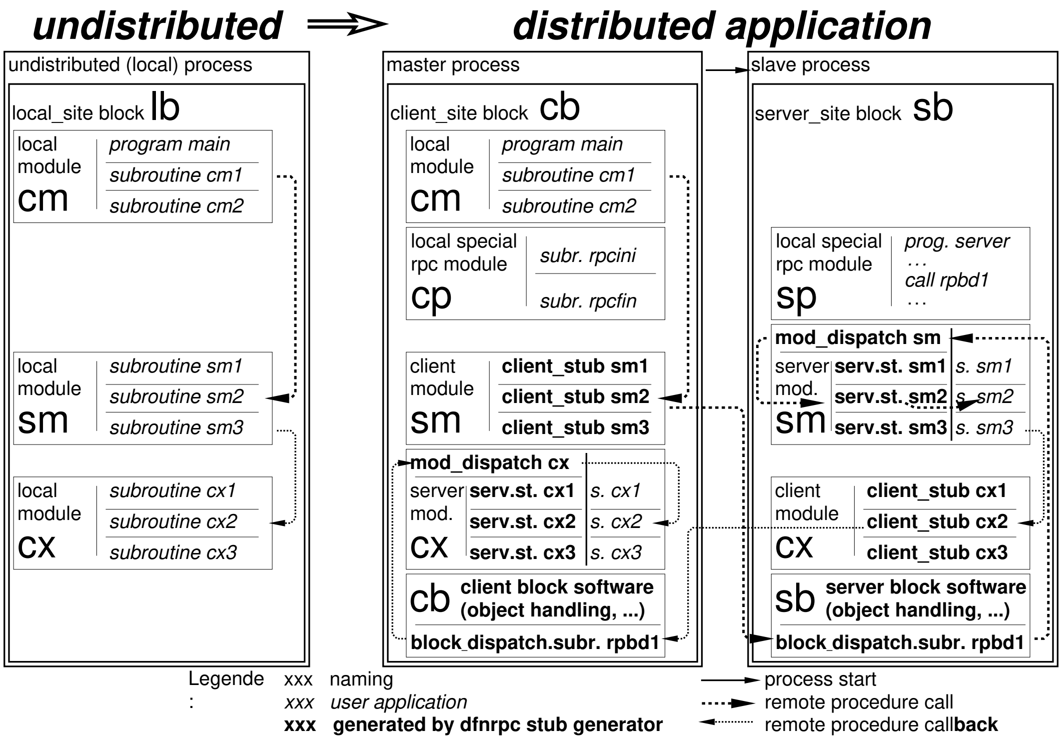


Abbildung 5.1: Four layered concept – process, block, module, subroutine

contains the *server stubs*, which, as a result of the incoming operation codes, call the respective server stub routines, and then the actual application server routines.

The server stubs can be generated by using the 'stub generator'.

– The module CX contains routines that are called by SM, but run in the client block. In the distributed case, this is a so called *callback*, therefore the CX client module is placed in the server block, and the CX server module (including the CX application subroutines) in the client block.

Every module has a *module_name* and *module_longname*. The module_name is used internally to address the module, and is also used in the generation of filenames, internal subroutine and common block names, and may only be two characters long; the first a letter, the second a number[1]. The module_longname may be a descriptive name.

- The application *subroutines* and functions are arranged into the separate modules, as described above.

5.2 Addressing

- To enable a subroutine to be called locally, the loader gives the subroutine a location address which is then used during a call to the subroutine. This enables a faster (indexed) access to the subroutine, as opposed to access via its name, which involves a search.

  With a remote procedure call, the call and location are on separate computers. Just as with the local call, we want to prevent a search when calling the remote procedure. As the binding

[1] Generated subroutine names are {*module_name*}1 / 2 / 3 / 4 / .../ D1 / D2 / D3 / .../ INI / NAM. A generated common block name is {module_name}_M. All DFN-RPC system subroutines start with RP or CT4.

and loading procedure on both the computers are independent, each subroutine is given an *operation code* by the stub generator. Therefore, when an application calls a subroutine from a client module, the corresponding client stub sends the appropriate operation code to the server module, and there, the *module dispatcher routine* calls the server stub, which in turn calls the wanted application subroutine.

- As different modules can be independently generated and compiled, it is not possible to ensure that each subroutine has a unique operation code, so each module within a block must have a unique *module code* for identification. For every client and server block, the stub generator creates a *block dispatcher*. In the client module the dispatcher adds the module code to every operation code to be sent, and in the server module, it passes the incoming call to the module dispatcher given by the module code. (See figure 5.2.)
- The addressing of slave processes and their constituent server blocks is explained in section 4.10 on page 42.

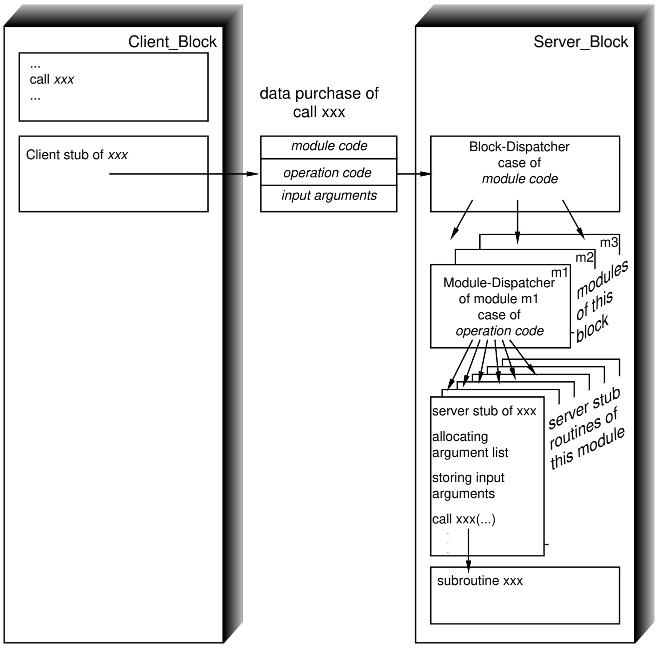


Abbildung 5.2: Dispatcher, module and operation code

5.3 The Fortran RPI Language

5.3.1 Input Format

RPI Definitions can be given in the following formats:

- Independent RPI statements of the form:

  `C*[function] parameter1 , parameter2 , ...`

  Possible *functions* are:

  ```
  BLK_BEG,  MOD_BEG,  RPI_BEG
  BLK_END,  MOD_END,  RPI_END
  BLK_OPT,  MOD_USE,  RPI_NON
  OPTION,   MOD_PAR,  RPI_CON
  ```

- Fortran statements:
  SUBROUTINE, FUNCTION, REAL, INTEGER, COMPLEX, LOGICAL, DOUBLE PRECISION, CHARACTER, BYTE, DIMENSION, PARAMETER, EXTERNAL, ENTRY
- Miscellaneous other Fortran statements.
- An RPI-**Qualifier** may be placed within a Fortran statement. There are two possibilities here:
  - Inline; the qualifier is placed in square brackets, e.g.

    ```
          REAL x(*) [IN, MAXINDEX=n]
    ```

  - On special comment lines, starting with C*RPC+, whereby the qualifier is placed in square brackets, e.g.

    ```
          REAL x(*)
    C*RPC+          [IN, MAXINDEX=n]
    ```

- Fortran statements that should only appear in the source code for DFN-RPC purposes should start with C*RPC (for the first line of a Fortran statement) or with C*RPC+ (for Fortran continuation lines, where any character except a blank can be used instead of +), e.g.

  ```
  C*RPC call rpcini ( para1, para2, para3,
  C*RPC+              para4, para5, para6 )
  ```

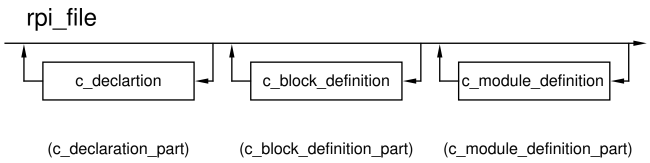
- Normal comment lines may start with C, c or *. *Inline comments* can be placed between a ! (exclamation mark) and the end of line (as with VAX Fortran).
- Key words can be written in upper or lower case letters.

5.3.2 General Structure

An RPI definition comprises the following parts:

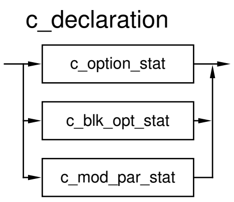
1. declaration part
2. block definition part
3. module definition part

The three parts must appear in this order, and in principle, each one may be left empty.



5.3.3 Declaration Part

There are three logical parts to the declaration, the order of which is not important.



1. General options

   C∗[OPTION] *keyword = value*

   In the following, the example values are the defaults.

   C∗[OPTION] SEND_BUFFER_LENGTH = 65528

   > All RPC routines in a block use a single send buffer to pass data to the remote computer. The size of the buffer (in bytes) can be set with this option (min 600, max 65528).
   > This option is currently unused.

   C∗[OPTION] RECEIVE_BUFFER_LENGTH = 65528

   > See description to SEND_BUFFER_LENGTH. The RECEIVE_BUFFER_LENGTH may be smaller than the SEND_BUFFER_LENGTH of the communicating partners. The underlying communications software automatically partitions the buffer. (Min size 600 bytes, max 65528). This option is currently unused.

   C∗[OPTION] WORD_BUFFER_LENGTH = 100000

   > All RPC routines in a given block make use of two buffers (word and character) in which the server stubs place all arguments. The word buffer is used for integer, real, double and complex arrays (single non-array arguments being stored in local variables by the server stubs). It is always initialized in the server block, also in the client block (even when it doesn't contain a server module), and is used by all server modules. The word buffer length (in bytes) is set with this option.
   > **WARNING**: With **separate generation** of *block* RPC software and *server* module RPC software, the *same value* must be given for all parts of a block. The values given for client and server blocks may be different, see also page 62 and page 92.

   C∗[OPTION] CHARACTER_BUFFER_LENGTH = 10000

   > See explanation of WORD_BUFFER_LENGTH above. Strings and string arrays are stored in the character buffer. (With MANU = ibm, max. length is 8000).

C*[OPTION] TRACE_DEPTH = 20

When an error occurs, the DFN-RPC writes a list of all currently called remote procedure calls and callbacks. TRACE_DEPTH sets what depth of call list should be maintained.

WARNING: See WORD_BUFFER_LENGTH.

This option is currently not used.

C*[OPTION] OPERATION_BITS = 10

There are in total 16 bits available for the module code and operation code (see section 5.1 on page 49). This parameter sets the number of bits to be used for the operation code. The module can then have a maximum of 2**OPERATION_BITS - 1 subroutines, functions and entries.

Consequently, only 2**(16-OPERATION_BITS) -1 modules can be used.

Valid range: 6 to 14 bits.

WARNING: OPERATION_BITS is evaluated during the generation of the block RPC software. **The same value must be used for the client and server blocks.**

C*[OPTION] OBJECTS_MAXIMUM = 512

Sets how many objects can be concurrently open during the object handling. See section 4.8 on page 39.

This option is currently not used.

C*[OPTION] WAIT_TIME_NEXT_CALL = 1000

Sets how long (in seconds) a server waits until the next RPC arrives, or how long the client waits until the RPC finishes.

C*[OPTION] WAIT_TIME_SEND = 300

Sets how many seconds a client stub waits until a send buffer, filled with arguments, is sent to the server stub, or how long the server stub waits until the send buffer, filled with output arguments, is sent to the client stub. This is important with asynchronous calls, or with full/slow networks.

C*[OPTION] WAIT_TIME_CONNECT = 100

Sets how many seconds a block has to establish a transport connection.

C*[OPTION] WAIT_TIME_ACCEPT = 2000

Sets how many seconds a block with an open port should wait until a connection on this port is established.

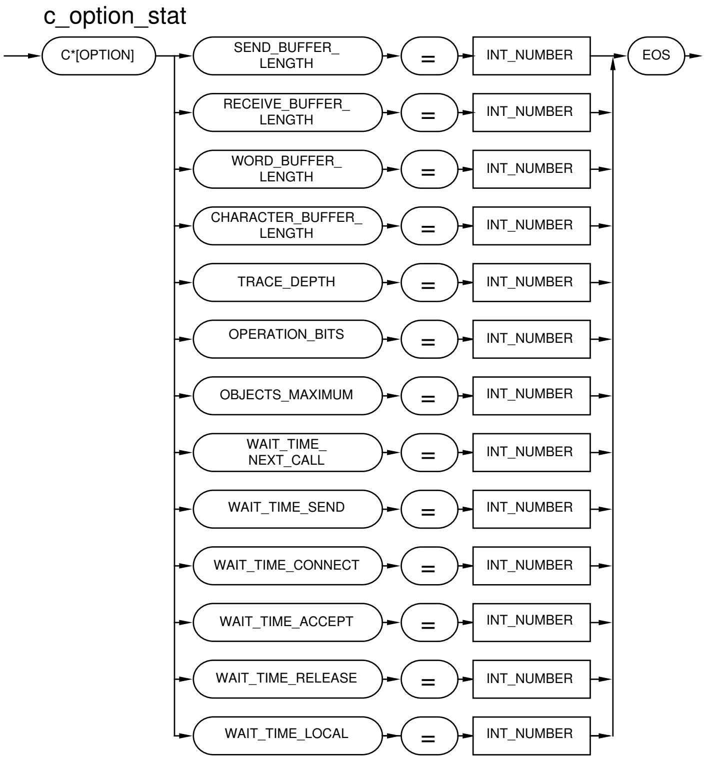
C*[OPTION] WAIT_TIME_RELEASE = 200

Sets the maximum time (in seconds) the setup of a connection may take.

C*[OPTION] WAIT_TIME_LOCAL = 100

Sets how many seconds a block has to open a transport service or port.

The WAIT_TIMEs are not implemented in release 1.0; **currently, the processes wait indefinitely.**



The following table shows where which constants are set, and how they are used during run time.

Constant	Written in the stub source file in the:	The constant is used:
WAIT_TIME_... OPERATION_BITS	Block software, generated through BLOCK_DEFINITION	Everywhere, i.e. in the block and module software, and in the stubs for each routine, both on the client and server side.
WORD_BUFFER_LENGTH CHARACTER_BUFFER_L.	Server stubs of each routine	In the server stubs
SEND_BUFFER_LENGTH RECEIVE_BUFFER_LENGTH TRACE_DEPTH OBJECTS_MAXIMUM	These options are currently not evaluated, but they are compiled into the DFN-RPC library during installation.	

2. Block Options

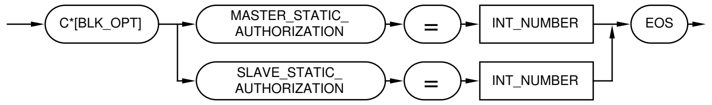
C∗[BLK_OPT] *keyword* = *value*

In the following, the value used in the example is the default.

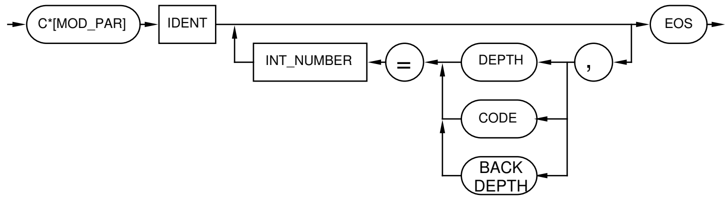
C∗[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 0
C∗[BLK_OPT] SLAVE_STATIC_AUTHORIZATION = 0

Any integer value between 0 and 999999999 (9 places) can be given. The processes use this value to check whether they are part of the same program.

c_blk_opt_stat



c_mod_par_stat



3. Module Parameters

C∗[MOD_PAR] *mod_name*,**DEPTH=***mod_depth*,**BACKDEPTH=***backdepth*,
CODE=*mod_code*

mod_name Indicates the module. See section 5.3.5 on page 62, and section 5.1on page 49.

DEPTH = mod_depth

This argument is optional.
See also figure 5.2 on page 51 and figure 5.5 on page 58.
When a module is used as a server module, (normally in a server block), then in addition to the stubs for every remotely called subroutine, the stub generator must also create the module dispatcher, whose job it is to interpret the header of each remote procedure call, and then call the appropriate server stub routine. The server stub routine then calls the application routine. When a second server makes a call via an RPC, or when a callback is made (i.e. a subroutine in the client block is called via an RPC), followed by another callback to the server block, then the block dispatcher and module dispatcher are called **recursively**.
Figure 5.2 on page 51 shows the operation of the block and module dispatcher during a simple remote procedure call. In the case of an embedded callback as shown in figure 5.3, the block dispatcher on the server side must be called at least twice, firstly as a result of branch to subroutine A according to the module and operation code in CALL A, and secondly the branch to subroutine C, as shown in figure 5.4. This means that figure 5.2 should really show an extra depth dimension. Also see figure 5.5 on page 58.
DEPTH=0 means that this type of recursive call is allowed[2].
DEPTH=mod_depth (>0) causes the stub generator to create mod_depth equivalent module dispatcher routines as an alternative to recursion. This value is also used by the block dispatcher, which calls the module dispatcher routine(s).
Default: DEPTH=0.

[2]Although standard Fortran 77 does not allow recursion, most Fortrans under posix can cope with it.

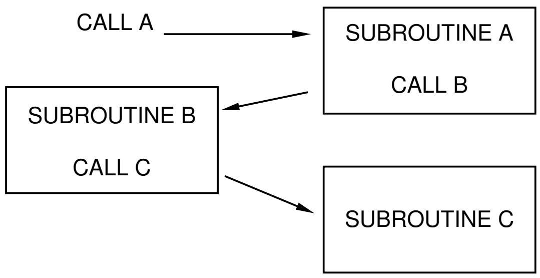


Abb. 5.3: Embedded Callback, 2-layered

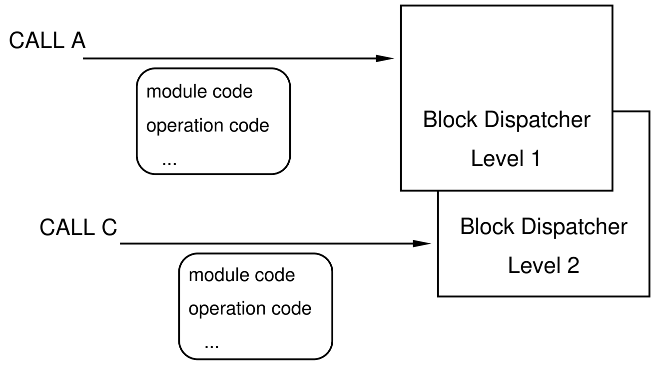


Abb. 5.4: Block dispatcher is called twice concurrently

BACKDEPTH = backdepth

This argument is optional.

The *backdepth* value is used in every client stub within the module, to generate the call for the appropriate block dispatcher routine for this callback depth (see DEPTH).

BACKDEPTH >= 0 - see DEPTH

BACKDEPTH = -1 means that no callbacks are expected, so no calls to a block dispatcher will be generated.

Default: BACKDEPTH=0

CODE = mod_code

This argument is optional.

Every module that is to be used as a client or server module within a block must be assigned a *module_code*. This can be automatically given by the stub generator, or manually set using a MOD_PAR statement. The code must be the same in every block in which the module is used. If the blocks are to be generated separately, then it is recommended that the MOD_PAR statement is used. The code must be within the range 1 to 2**(16-OPERATION_BITS)-1, which, for the default OPERATION_BITS value of 10 means in the range 1 to 63. See page 54 - OPERATION_BITS. Default: Automatic assignment of values happens when no value is given in a MOD_PAR statement, and the module is used for the first time in a MOD_USE statement as CLIENT or SERVER. The values 1, 2, ... are used (except where they have already been used in a MOD_PAR statement).

See section 5.2 on page 50.

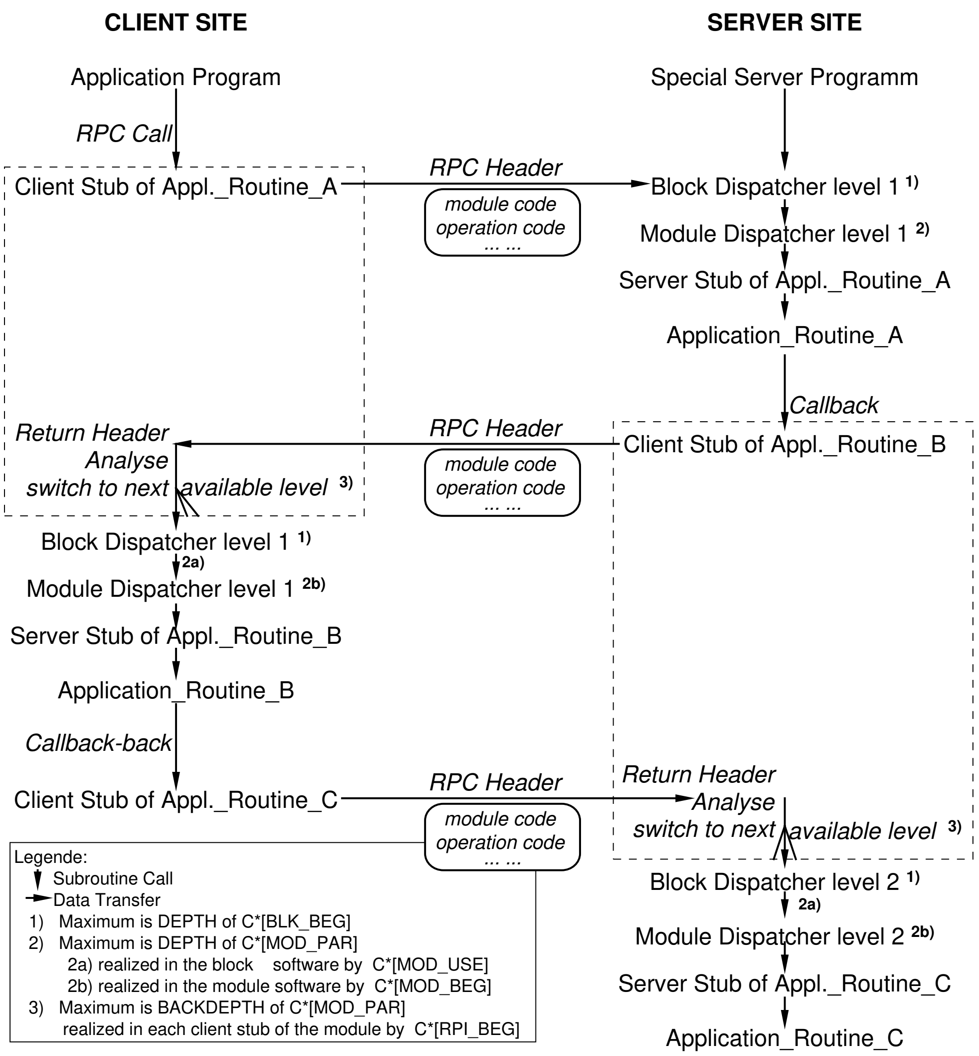


Abb. 5.5: Block, module and back depth

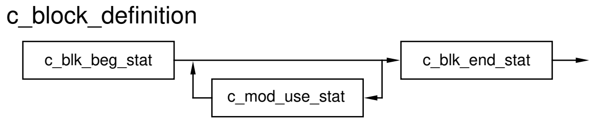
5.3.4 Block Definition Part

The block definition part comprises separate *Block Definitions*.

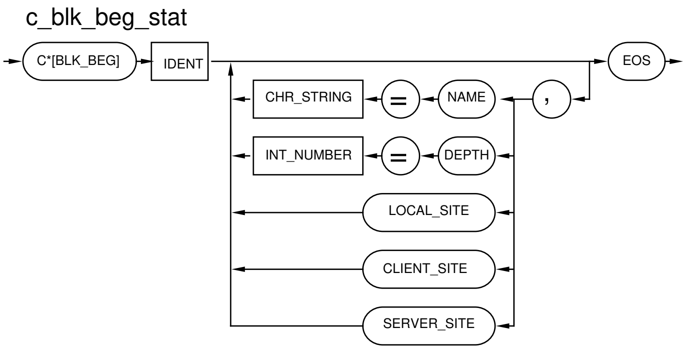
A single block definition has the following structure:

C∗[BLK_BEG] *block_name*,**NAME=**'*block_longname*',**DEPTH=***block_depth*, *site*
C∗[MOD_USE] *module_name*, *module_type*
...
C∗[BLK_END]

A block can contain one or more modules.



C∗[BLK_BEG] *block_name*, **NAME=**'*block_longname*',**DEPTH=***block_depth*, *site*



marks the start of a block definition.

block_name

Defines the block_name of this block. See section 5.1 on page 49.

NAME = block_longname

Defines the block_longname of this block. This is optional, and its default value is the block name.

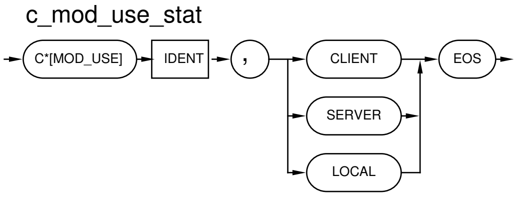
DEPTH = block_depth

This argumentis optional. See section 5.3.3, "Module Parameters" on page 56, and figure 5.5 on page 58.
DEPTH = 0 means that the block dispatcher may be called recursively[3].
DEPTH = block_depth (>0) causes the stub generator to create block_depth equivalent block dispatcher routines as an alternative to recursion.
DEPTH = -1 makes sense on the CLIENT_SITE block only, and means that no callbacks are expected, therefore no block dispatcher will be generated.
Default: DEPTH=0

[3]Although standard Fortran 77 does not allow recursion, most Fortrans under posix can cope with it.

site — Possible values for *site* are **CLIENT_SITE**, **SERVER_SITE** or **LOCAL_SITE**. One generates a client block by using CLIENT_SITE etc., see section 5.1 on page 49.

C*[MOD_USE] *module_name*, *module_type*



includes the given module in this block.

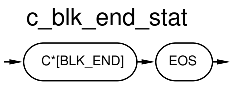
module_name

See section 5.1 on page 49.

module type

Possible values for *module_type* are **CLIENT**, **SERVER** or **LOCAL**.
A client module contains only the client stubs for this module, but a server module contains the server stubs and the respective application subroutine. A local module contains the application subroutines only.

C*[BLK_END] marks the end of a block definition.



A client block (cb in the following example) must contain a local module (cm) containing the main program, and at least one client module (sm). It may also contain a server module, the subroutines of which are called from a remote process using the so called 'callback' mechanism (not shown in this example). Most client blocks also contain a local module (cp) which has the subroutines for initializing and terminating the remote server block(s).

A server block (sb) must contain a local module (sp) with the main program which undertakes all operations necessary for starting the DFN-RPC server side process. It must also contain the server module (which must naturally has the same name as the client module given above, that is sm). If callbacks are used, that is RPCs from the server block back to the client block, then the server block must also contain a client module (which must correspond to the server module given above).

Lines starting with **C*RPC** or **C*RPC+** are activated in client and server blocks. See section 5.3.1 on page 52.

A local block (lb) may not make any RPCs, and therefore contains local modules only (cm and sm).

Example:

```
C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     cp, LOCAL
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sp, LOCAL
C*[MOD_USE]     sm, SERVER
C*[BLK_END]

C*[BLK_BEG] lb, NAME='local program', DEPTH=0, LOCAL_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     sm, LOCAL
C*[BLK_END]
```

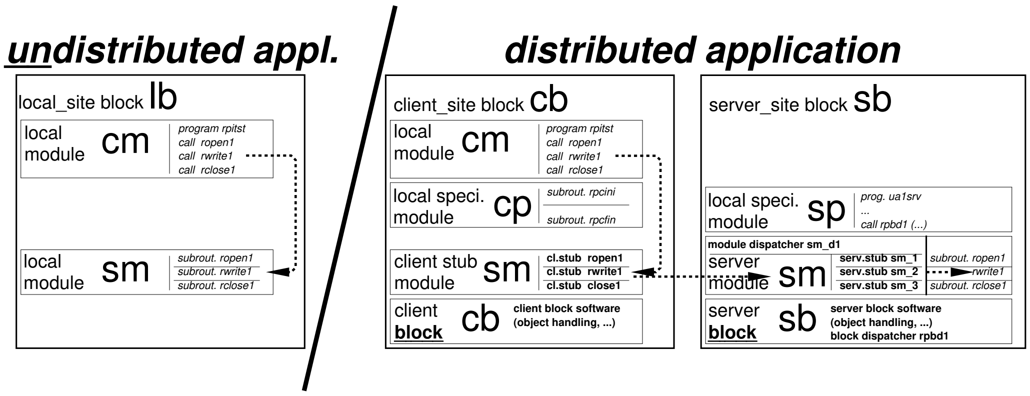


Abbildung 5.6: Simple Example

As a result of the block definition parts, the stub generator can create the application and stub source files for each block. If the above example, including the module definition parts, is a single source file (e,g, inp.rpi), then the stub generator will create the following files:

```
                                    inp.rpi
                                       |
             +-------------------------+------------------------+
For the      |                         |                        |
blocks ..... cb                        sb                       lb
following    |\                        |\                       |
files will   | \                       | \                      |
be generated: | inp_cb_stb.m4          | inp_sb_stb.m4          |
             | |                       | |                      |
       inp_cb.f inp_cb_stb.f    inp_sb.f inp_sb_stb.f     inp_lb.f

Stub-Presourcefile:     inp_cb_stb.m4  and  inp_sb_stb.m4
Stub-Sourcefile:        inp_cb_stb.f   and  inp_sb_stb.f
Application-Sourcefile: inp_cb.f       and  inp_sb.f  and inp_lb.f
```

If instead the block definition part (including the declarations) is contained in inp.rpi, but one would like to generate the modules separately, then the following files are created:

```
                                      inp.rpi
                                         |
               +------------------------+
 For the       |                        |
 blocks ..... cb                       sb
               |                        |
 following      \                        \
 files will      \                        \
 be generated:   inp_cb_stb.m4            inp_sb_stb.m4
                 |                        |
                 inp_cb_stb.f             inp_sb_stb.f
```

The stub files contain the block part of the stubs only, that is the block dispatcher and object start routines.

The block definition part can also be dropped, in which case the modules are separately generated.

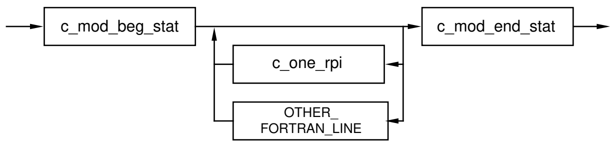
5.3.5 The Module Definition Part

The module definition part comprises separate *module definitions*.

The structure of a module definition has the following form:

C*[MOD_BEG] *module_name*,**NAME=**'*module_longname*'   **module_body** **C*[MOD_END]**

c_module_definition



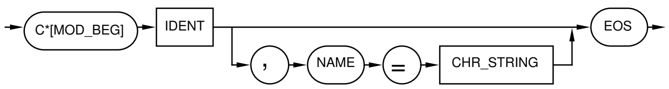
C*[MOD_BEG] *module_name*,**NAME=**'*module_longname*'

marks the start of a module definition.

module_name
: Defines the module_name of this module, see section 5.1 on page 49.

NAME = module_longname
: Defines the module_longname of this module.

c_mod_beg_stat



module_body

The module_body is made from the *application program source* including the *remote procedure interface* (**RPI**) *definition* of all subroutines, functions and entries in the module.

In principle, it is possible to place a **C∗RPC** or **C∗RPC+** line at any point within the body. See section 5.3.1 on page 52 and section 5.3.4 on page 59.

Formally, the *module body* is constructed from a series of:

- Local routines

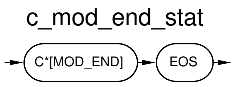
  That is, the local main program, local subroutines and local functions. These are not read by the stub generator, instead are simply written into the block Fortran file containing this module, with a *module_type* of SERVER or LOCAL (see C∗[MOD_USE]).

- Remote Procedure Interface (RPI)

  Remote Procedure Interfaces are handled by the stub generator in the following way (also see section 5.3.6 on page 65):

  1. The same as local routines, that is the Fortran source, and are therefore written to the same Fortran files as the other local routines.
  2. On the basis of the subroutine/function header, and also the various RPI qualifiers (see next section) placed in the file, the client and server stubs for this module and the module dispatcher routine are generated.

C∗[MOD_END] marks the module definition end.



Example (using the block definition example in the last section):

```
C*[MOD_BEG] cm, NAME='client module - main program'

      PROGRAM ...
      ...
C*RPC CALL rpcini            ! These lines will be duplicated and
      ...                    ! the second activated in the generated
C*RPC CALL rpcfin            ! source of block cb
      ...
      END
                             ! Source for the client module
      SUBROUTINE ...
      ...
      END

      FUNCTION ...
      ...
      END

C*[MOD_END]
```

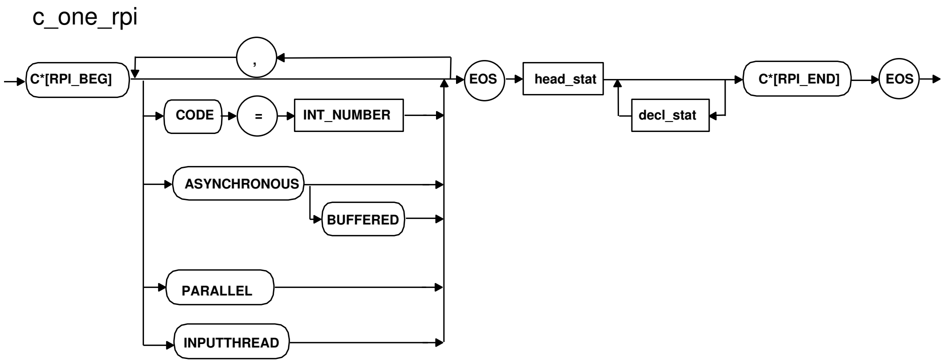
```
C*[MOD_BEG] cp, NAME='client rpc special routines '

      SUBROUTINE rpcini      ! Examples will be provided
      ...
      END

      SUBROUTINE rpcfin
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sp, NAME='server rpc special program  '

      PROGRAM rpcsrv         ! Special main program for the server block
      ...
      CALL BLKD1 (iret)      ! The block-dispatcher-routine
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sm, NAME='server module - server subroutines'

C*[RPI_BEG]
      SUBROUTINE sub1 (a,b,c) ! This is a RPI
      REAL a[IN], b[IN]
      REAL c
C*RPC+      [OUT]
      ...
C*[RPI_NON]
      a = b * b
      RETURN
      END
C*[RPI_END]

      SUBROUTINE ...         ! This is a local routine, i.e.
      ...                    ! not callable by remote blocks
      END

C*[MOD_END]
```

5.3.6 The Remote Procedure Interface (RPI) for a Routine

In a module that should be run remotely, all routines to be called by the client must be declared with a remote procedure interface (RPI). Routines that are not declared with an RPI are available on the server side (as help routines) only.

An RPI has the following structure:

C*[RPI_BEG] CODE=*operation_code*, **ASYNCHRONOUS** or **ASYNCHRONOUS BUFFERED** or **PARALLEL** or **INPUTTHREAD** **routine_header** **declaration_statements** ... **C*[RPI_END]**



C*[RPI_BEG] CODE=*operation_code*, **ASYNCHRONOUS** *or* **ASYNCHRONOUS BUFFERED** *or* **PARALLEL** *or* **INPUTTHREAD**

marks the start of a remote procedure interface.

CODE = operation_code

With every module, it is possible to decide whether the operation_code is selected by hand, or automatically by the stub generator. In the first case, the code must be explicitly given for every RPI, and in the second case none of the codes may be selected by hand. Explicitly setting the codes is of advantage only when creating libraries as when one later adds new server routines, the old codes will not be changed so old client programs will then be able to use the new server library.
Valid range for code values is 1 to 2**OPERATION_BITS - 1, that is, with the default of OPERATION_BITS = 10, valid OPERATION_CODEs are 1 to 1023. See also page 54; OPERATION_BITS.

None of the options ASYNCHRONOUS, ASYNCHRONOUS BUFFERED or PARALLEL

In this case, a **synchronous RPC** is used, meaning that the client stub sends the input arguments to the server stub, which calls the application routine while the client stub waits until the server routine is finished. The

output arguments are then sent from the server stub back to the client stub as shown in figure 5.7.
It should be noted that most networks and computers can manage a rate of 5 - 200 RPCs/sec (with an empty argument list). Therefore, one shouldn't attempt a rate of more than 10 synchronous RPCs/sec. [4]

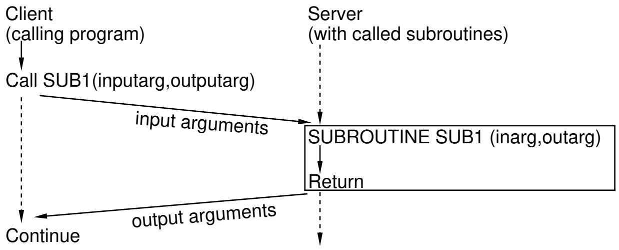


Abbildung 5.7: A Synchronous RPC

ASYNCHRONOUS

With an asynchronous RPC, the client stub sends the input arguments to the server stub, and then passes control directly back to the application program. On the server side, the RPC called subroutine runs in parallel on the remote machine, as shown in figure 5.8. The routine must have **no output arguments**, and it may not be function. If necessary, results can be written to a common block, and passed back to the client later via a synchronous call. Synchronization takes place again with the next synchronous RPC.
Asynchronous calls enable the parallelization of client and server. With a series of calls directly following each other, i.e. when the parallelization is not fully exploited, one can obtain up to three times the number of RPCs/sec (with an empty argument list) as compared to synchronous RPCs (it is recommended however that the following RPCs are used).[4]

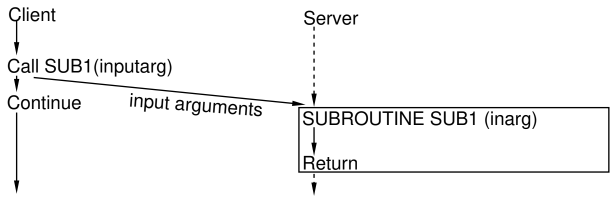


Abbildung 5.8: An asynchronous RPC

ASYNCHRONOUS BUFFERED

With asynchronous, buffered RPCs, the client stub waits until the send_buffer is full, or a following synchronous or unbuffered asynchronous RPC call is made before sending the buffer. This helps to reduce the number of system calls made, increasing the transmission rate. See figure 5.9. The routine may not return any arguments, and cannot be a

[4] Measurements on an ethernet with TCP between two Sun4s, every call with a 28 byte argument list (plus a 4 byte header): 3.0ms/call synchronous, 1.2ms/call asynchronous with 100 or more calls, 0.18ms/call asynchronous buffered with 85 calls (i.e. 2 full MTUs with 1500 bytes each were sent per system call).

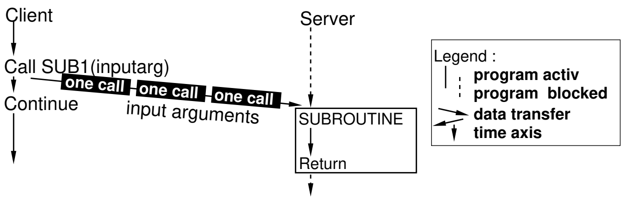


Abbildung 5.9: Asynchronous, buffered RPC

function. With small argument lists, the round-trip-time is significantly reduced through the buffering.[4]

PARALLEL

With parallel RPCs, the control is passed directly back to the client program as soon as the input parameters have been sent, as with asynchronous RPCs. On the server side, the RPC called server routine is then executed concurrently with client continuation, as shown in figure 5.10. Once

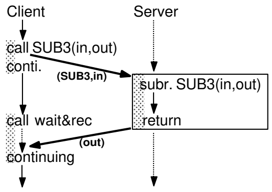


Abbildung 5.10: The parallel RPC

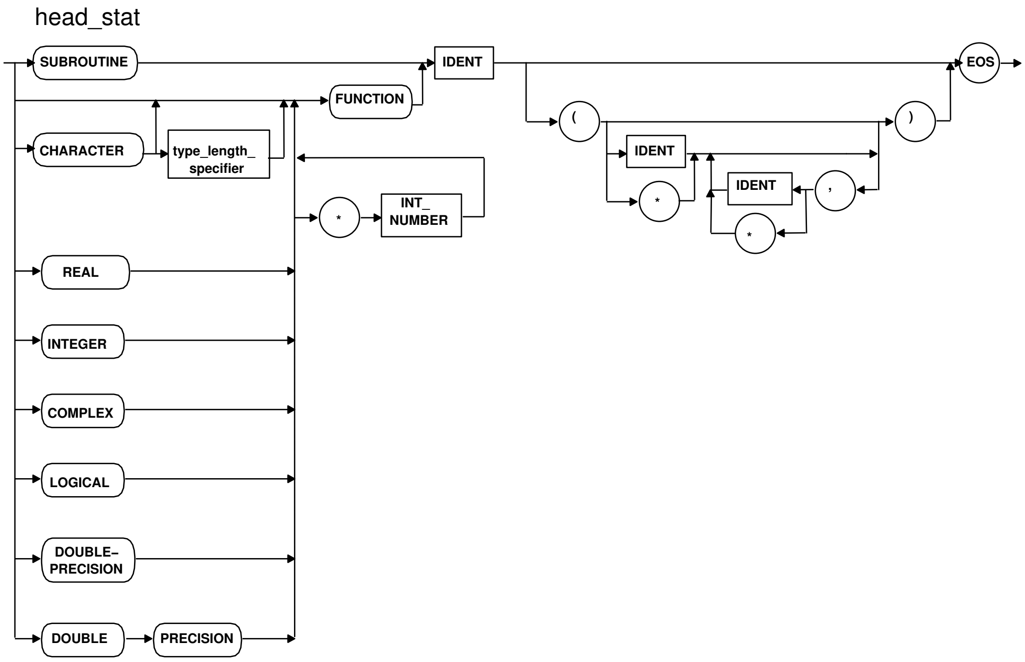
the server routine is finished, it sends the output parameters back, and is therefore blocked until the library routine RPOREC is called by the client. The routine RPOREC writes the output values that are given as output parameters in the RPC call.
Also see the library routines RPOISP for checking an arriving packet and RPOCRY, RPOLRY, RPORRY and RPOGRY to enable waiting for a return packet (and also RPORCT for a threaded receive - not yet implemented).

INPUTTHREAD

Sending an input parameter in a parallel thread.
This is optional, and is only possible with ASYNCHRONOUS or PARALLEL RPCs. INPUTTHREAD is not yet implemented for generated stubs.

Lastly, a **header_statement** must be given as a standard Fortran statement:

SUBROUTINE ... **... FUNCTION ...**



The following **Fortran declaration statements** can then be used:

```
CHARACTER ...
 INTEGER ...
 LOGICAL ....
 REAL ...
 DOUBLE PRECISION ...
 COMPLEX ...
 BYTE ...
 DIMENSION ...
 PARAMETER ...
 EXTERNAL ...
 ENTRY ...
```

Within the declarations, a **qualifier** can be given, with which all additional information required by the stub generator is specified. The qualifiers are described in the next section.

Additionally, **include statements** are also interpreted within an RPI definition.

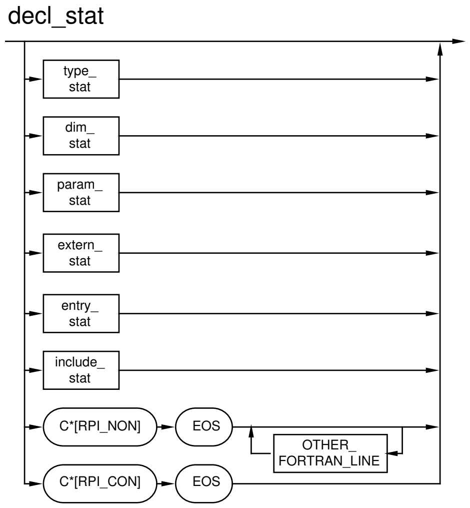
```
      INCLUDE    filename
or  INCLUDE (  filename  )
or  INCLUDE  '  filename '
or  INCLUDE ('  filename ')
```

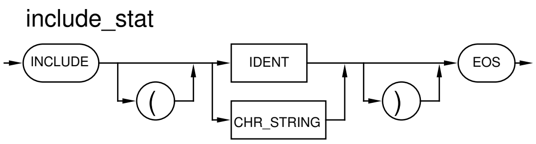
All other Fortran code and declarations that should not be interpreted by the stub generator must be placed in **nonparseable blocks**:

```
C*[RPI_NON]
      other Fortran lines etc.
C*[RPI_CON]
```

If there are no further declarations, i.e. a C*[RPI_END] follows, then the C*[RPI_CON] need not be used.

Nonparseable blocks are used to hide all information unnecessary for stub generation from the stub generator, such as equivalence and common block declarations, data statements and executable statements. All routines that are not declared as RPI statements are treated as nonparseable blocks.

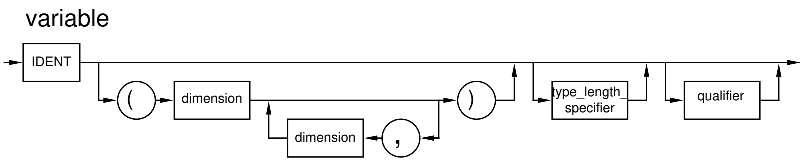




5.3.7 Numerical and String Arguments

Additional information for the stub generator can be added to the formal argument declarations by using a qualifier.

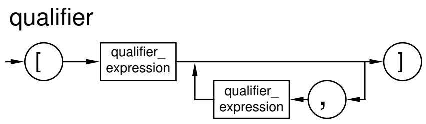
A qualifier can only be used with formal arguments, and is placed behind the dimension entry (...) or type length specification *1/2/4/8/16/32.



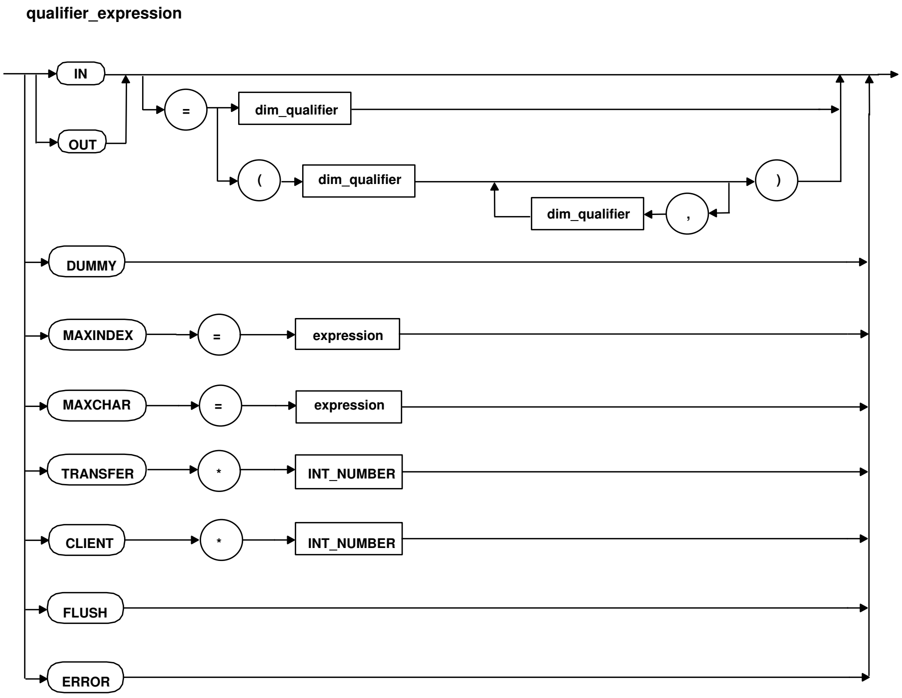
The general syntax for declaration statements is described in section 5.6 on page 81.

The **qualifier** is placed in **square brackets** , and is constructed from one or more **qualifier_expressions**.

type	*identifier*	[*qualifier_expression*]
oder		[*qualifier_expression1* , *qualifier_expression2* , **...**]
DIMENSION		[**...**] , *identifier2* , [**...**] , ...



A qualifier_expression starts with one of the following keywords: **IN**, **OUT**, **DUMMY**, **MAXINDEX**, **MAXCHAR**, **TRANSFER**, **CLIENT**, **FLUSH** or **ERROR**.



The following sections describe the meaning of the **qualifier_expressions**.

5.3.7.1 IN, OUT, DUMMY

IN
IN = *Dimension_ Qualifier*
IN = (*Dimension_ Qualifier1, Dimension_ Qualifier2* , ...)
OUT
OUT = *Dimension_ Qualifier*
OUT = (*Dimension_ Qualifier2, Dimension_ Qualifier2* , ...)
DUMMY

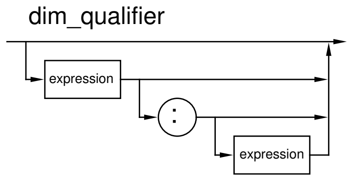
As opposed to normal procedure calls, it is not possible to use a call-by-reference argument in an RPC. Instead, a call-by-in-copy-and-call-by-out-copy is done, i.e. the value of the input argument is passed from client to server before the actual subroutine call. The value is automatically converted during transmission, on the client side from the client format to the transfer encoding, and on the server side from the transfer encoding to the server number format. The format used for transfer encoding with the DFN-RPC can be freely chosen; see the stub generator dfnrpcgen option -T on page 155. Normally, the transfer encoding should be one of the client or server number formats. Once the subroutine is finished, the output arguments are transmitted back to the client.

With call-by-reference, the direction set for the arguments (in or out) is not important for the compiler, and so does not need to be given in the Fortran code. With the RPC however, it is necessary to specify if the argument is an output or input argument so that the stubs know whether to make an in-copy or an out-copy.

Example:

```
C*[RPI_BEG]
C     [IN], [OUT], [IN,OUT] and [DUMMY] Arguments
C     -------------------------------------------

      SUBROUTINE sub1 (a,b,c,d,e,x)
      REAL    a[OUT],  b[IN],  c[IN],  d, e[IN,OUT], x[DUMMY]

C     the default [in,out] is used for d
C     x is dummy, i.e. a workspace argument, i.e. it will not be
C     transferred, neither for input nor for output

C*[RPI_NON]
      a = b * c
      x = d
      d = e
      e = x
      RETURN
      END
C*[RPI_END]
```

The dimension_qualifier is used to specify the section to be transmitted. If the dimension_qualifier given is smaller than the number of array dimensions, then the dimension_qualifier is used for the first of the array dimensions. The additional dimensions are passed completely.



A dimension_qualifier can have one of the following three formats:

empty	i.e. this array dimension is completely transmitted.
expression	i.e. this array dimension from the start to the index given will be transmitted.
expr1:expr2	i.e. for this array dimension, only the section given by the two indexes will be passed.

By using **IN**, one can specify which array section will be used as an input argument, if any.

With **OUT**, one can specify the array section to be used as an output argument. Several IN and OUT sections can be given.

When **DUMMY** is used, no actual values are transmitted, i.e. on the server side, memory is allocated for the argument only.

If none of the qualifiers IN, OUT or DUMMY are given, then the default is [IN, OUT], i.e. the argument is transmitted in both directions.

WARNING: If RPC routines are called with **constants** as arguments, then the formal argument should be declared as [IN], otherwise the constant will be overwritten during the subroutine return. On some computers, this can cause a program crash.

With asynchronous RPCs, it is only possible to use input arguments, and for safety, they must be specifically declared as IN.

5.3.7.2 The MAXINDEX and MAXCHAR of an Array

MAXINDEX	= *expression*
MAXCHAR	= *expression*

MAXINDEX must be given when the last array dimension is a $*$, i.e. variable. The maximum index is specified with MAXINDEX (not the array length). MAXINDEX sets how much memory should be allocated in the server stub for this argument. It is also possible to allocate a different amount of memory for an array of set length as is given in the array's declaration. MAXINDEX also sets the maximum possible section size that can be transmitted.

MAXCHAR overwrites the length of a string argument. With string arrays, MAXCHAR applies to the string length i.e. MAXCHAR sets the length that can be queried in the server routine with LEN.

All arrays are placed in word or character buffers on the server side. The sum of the memory requirements[5] for all arguments must not be larger than WORD_BUFFER_LENGTH and CHARACTER_BUFFER_LENGTH, see page 53.

[5] Care must be taken with a callback to a callback, two routines are active on the server side at the same time, i.e. there must be enough memory for both argument lists.

Example:

```
C*[RPI_BEG]
C     Arrays and MAXINDEX
C     -------------------

      SUBROUTINE sub2 (n,a,b,c)
      INTEGER n    [IN]
      REAL    a(n) [OUT, MAXINDEX=n/2]
      REAL    b(n) [IN]
      REAL    c(n) [IN]

C     MAXINDEX overwrites the upper bound of the (last) dimension of
C     an array.
C     MAXINDEX must be given, if the (last) dimension is an asterisk.
C     MAXINDEX is used for memory allocation for the actual argument
C     in the server stub. MAXINDEX is also used as default and maximum
C     for the [IN] and [OUT] range (see next example).

C*[RPI_NON]
      DO 10  i=1,n/2
         a(i) = b(i)*c(i) + b(i+n/2)*c(i+n/2)
10    CONTINUE
      RETURN
      END
C*[RPI_END]


C*[RPI_BEG]
C     Arrays with specified IN/OUT Range
C     ----------------------------------

      SUBROUTINE sub3 (n,a)
      INTEGER n(2) [IN]
      REAL    a(*) [MAXINDEX=n(1)*n(2), IN=1:n(1), OUT=n(2)+1:n(2)+n(1)]
C*[RPI_NON]
      DO 10 i=1,n(1)
         a(n(2)+i) = a(i) * a(i)
10    CONTINUE
      RETURN
      END
C*[RPI_END]


C*[RPI_BEG]
C     Multidimensional Arrays
C     -----------------------

      SUBROUTINE sub4 (a,n,nbeg,nend,m,mbeg,mend)
      INTEGER n[IN], nbeg[IN], nend[IN]
      INTEGER m[IN], mbeg[IN], mend[IN]
      REAL    a(n,*) [MAXINDEX = MAX(m,mend)           ,
     +                IN       = (nbeg:nend, mbeg:mend),
     +                OUT      = (nbeg:nend, m:m)      ]
```

```
C*[RPI_NON]
      DO 20 i = nbeg, nend
         sum = 0
         DO 10 j = mbeg, mend
            sum = sum + a(i,j) * a(i,j)
10       CONTINUE
         a(i,m) = sum
20    CONTINUE
      RETURN
      END
C*[RPI_END]


C*[RPI_BEG]
C     Character String Arguments
C     --------------------------

      SUBROUTINE sub5 (s1, s2, s3, s4, s5)

      CHARACTER*(*)  s1 [IN]
      CHARACTER*(*)  s2 [OUT=3]
      CHARACTER*(*)  s3 [IN, OUT, MAXCHAR=90]
      CHARACTER*(*)  s4 [DUMMY]
      CHARACTER*(*)  s5 [DUMMY,   MAXCHAR=500]

C*[RPI_NON]

C     If sub5 is called with the following actual argument list

C          CALL sub5 (a1(1:2), a2(11:15), a3(1:7), a4(41:100), ' ')

C     then s1 is declared as char.*2   and has got the value of a1(1:2)
C          s2 is declared as char.*5   and has got no value
C          s3 is declared as char.*90  and has got the value of a1(1:7)
C          s4 is declared as char.*60  and has got no value
C          s5 is declared as char.*500 and has got no value

C     After return  a1 will be unchanged
C                   a2(11:13) gets the content of s2(1:3)
C                   a3(1:7)   gets the content of s3(1:7)
C                   a4 will be unchanged
C                   a5 will be unchanged

      RETURN
      END
C*[RPI_END]

C Remarks:
C     The length of a string argument is always an invisible input
C     argument. The length of the allocated memory in the server is the
C     length of the actual argument and can be overwritten by the
C     MAXCHAR parameter.
C     The default of the [IN] and [OUT] range is the minimum of the
C     length of the actual argument and the MAXCHAR parameter.
```

```
C*[RPI_BEG]
C     Arrays of Strings
C     -----------------

      SUBROUTINE sub6 (sarr,n)

      CHARACTER*(*)  sarr(*) [MAXINDEX=n, IN = (   ,1:n/2)  ,
     +                                    OUT= (1:3,n/2+1:n)]

C     In the IN and OUT ranges, the first dimension is the substring.
C     The omitted dimension in IN=(,...) is the same as 1:*, where *
C     is the string length, which can be overwritten by MAXCHAR.

C*[RPI_NON]
      DO 10  i=1,n/2
         sarr(n/2+i)(1:3) = func6 (sarr(n/2+i))
10    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.3.7.3 TRANSFER - Reduced Transfer Accuracy

TRANSFER	$*$ **1**
	$*$ **2**
	$*$ **4**
	$*$ **8**
	$*$ **16**
	$*$ **32**

The number of bytes used to transfer a numerical value is set with **TRANSFER**. If no value is given, then the following table shows the default values used:

Type	Def. transfer length	allowed
INTEGER	4	1,2,4,8
LOGICAL	4	1,2,4,8
REAL	4	4,8,16
DOUBLE PRECISION	8	4,8,16
COMPLEX	8	8,16,32
BYTE	1	1,2,4,8
INTEGER * 1/2/4/8	1/2/4/8	1,2,4,8 resp.
LOGICAL * 1/2/4/8	1/2/4/8	1,2,4,8 resp.
REAL * 4/8/16	4/8/16	4,8,16 resp
COMPLEX * 8/16/32	8/16/32	8,16,32 resp

Example:

```
C*[RPI_BEG]
C     Reduced Transfer Precision and Check Modus
```

```
C     -----------------------------------------

      SUBROUTINE sub7 (i,j,x,y)

      INTEGER           i [IN,  TRANSFER *2]
      INTEGER           j [IN,  TRANSFER *1]
      DOUBLEPRECISION   x [OUT, TRANSFER *4]
      DOUBLE PRECISION  y [OUT, TRANSFER *16]
C     To obtain full precision between two hosts with 64 bit
C     numerical units, one must qualify reals with TRANSFER*8
C     and doubles with TRANSFER*16, e.g. see argument y.
C*[RPI_NON]
      x = float(i)**j
      y = float(i)**j
      RETURN
      END
C*[RPI_END]
```

5.3.7.4 CLIENT - Different Data Type Lengths in Client and Server

CLIENT	$*$ **1**
	$*$ **2**
	$*$ **4**
	$*$ **8**
	$*$ **16**
	$*$ **32**

CLIENT is used to set how many bytes the argument in the client stub is declared as, that is, which units will be used in the client for the argument when calling the remote subroutine.

```
C Client:
C -------
      REAL*8  x
      REAL*4  y
      CALL sub8 (x,y)

C Server:
C -------
C*[RPI_BEG]
C     Differing Declarations in the Client
C     ------------------------------------

      SUBROUTINE sub8 (x,y)
      REAL*4  x [CLIENT *8]  ! in server routine and server stub real*4
                             ! in the client stub real*8
      REAL*8  y [CLIENT *4]  ! in server routine and server stub real*8
                             ! in the client stub real*4
C     For choosing the number of bytes which will be transferred the
C     minimum of both declarations will be relevant.
C*[RPI_NON]
```

```
      d = x
      x = y
      y = d
      RETURN
      END
C*[RPI_END]
```

For **INTEGER**, **LOGICAL** and **BYTE**, the values 1, 2, 4 and 8 are allowed, for **REAL** the values 4, 8, 16 and for **DOUBLE PRECISION** and **COMPLEX** the values 8, 16 and 32.

5.3.7.5 FLUSH - Buffered or Unbuffered Call

FLUSH

FLUSH can be used to qualify a LOGICAL parameter in an ASYNCHRONOUS RPC.

The value of FLUSH during the client call sets whether the asynchronous RPC is buffered (.FALSE.) or sent directly (.TRUE.).

5.3.7.6 ERROR

ERROR

ERROR can be used to qualify an INTEGER parameter.

If an error occurs during an RPC, then the application is not stopped, instead the error code is written into the client's ERROR parameter. A return value of 0 means that no error was found.

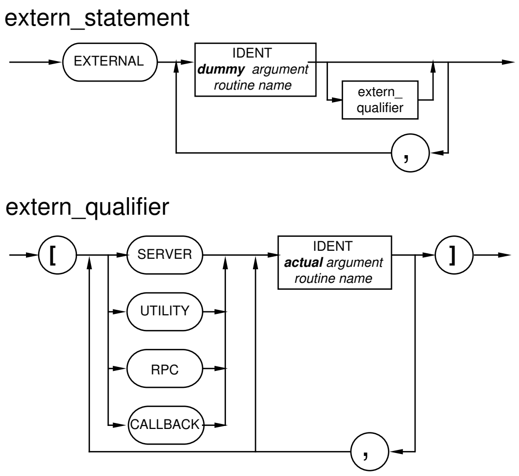
This qualifier is currently not fully implemented.

5.3.8 Externals as Arguments

Externals can also be passed as call-by-reference arguments to RPCs, which causes two main problems:

- Is the routine that is passed as an argument actually found on the server or the client side?
- What is the reference called on the other side?

If no relevant routine is found on the client side, then a dummy routine is created so that the appropriate reference can be created and converted to an argument.



The various possibilities are shown in the following example:

Example:

```
C*[RPI_BEG]
C     Externals as dummy arguments
C     ----------------------------

      SUBROUTINE sub12 (e)

      EXTERNAL  e [SERVER   s1, s2, s3,
     +             UTILITY  u1, u2, u3, u4,
     +             RPI      r1,
     +             CALLBACK b1 ]

C*[RPI_NON]
      RETURN
      CALL e
      END
C*[RPI_END]

C Remarks:

C     Subroutine or function dummy arguments must be declared as
C     external in the subroutine or function, where it is in the
C     dummy argument list. (Normally it must be declared only in the
C     calling routine, i.e. if a routine name is used as an actual
C     argument.)
C     The whole range of possible actual values must be specified.
C     Also it must be specified, where those routines are located.

C     'mmm' should be the module in which sub12 is declared.
C     'sss' should be the block where sub12 is located as routine,
C     i.e. where mmm is declared (C*[MOD_USE]) as SERVER.
C     'ccc' should be the block from where sub12 is called,
C     i.e. where mmm is declared (C*[MOD_USE]) as CLIENT.
C     Then the following table illustrates the location of
C     the routines used as actual arguments for e
```

```
C     and on which host the actual arguments are used

C     declared as | acc                 |                  sss | called on
C     ------------+---------------------+---------------------+----------
C     SERVER      |                     |            s1 s2 s3 |   sss
C     UTILITY     | u1 u2 u3 u4         |         u1 u2 u3 u4 |   sss
C     RPI         | Stub_with_entry_r1 --------> stub      r1 |   sss
C     CALLBACK    | b1   stub <--------- stub_with_entry_b1 |   ccc

C     In the case of SERVER, the stub generator will generate in the
C     client stub of mmm (here located in ccc) the corresponding
C     dummy routines, here s1, s2 and s3.

C     CAUTION: In all blocks where mmm is used as client stub or server
C              the EXTERNAL declaration must be valid, i.e. the role of
C              possible actual arguments (s1,s2,s3 / u1,u2,u3,u4 / r1 /
C              b1) must be se same (SERVER / UTILITY / RPI / CALLBACK).
```

If, within a given block, a subroutine used in several modules for which client stubs are generated and as a SERVER routine in each of these modules, then it must only be declared once as a SERVER routine, and should be set as a utility routine in other modules, otherwise several dummy routines will be created within the block.

5.3.9 Alternative Returns

Alternative returns are possible.

Example:

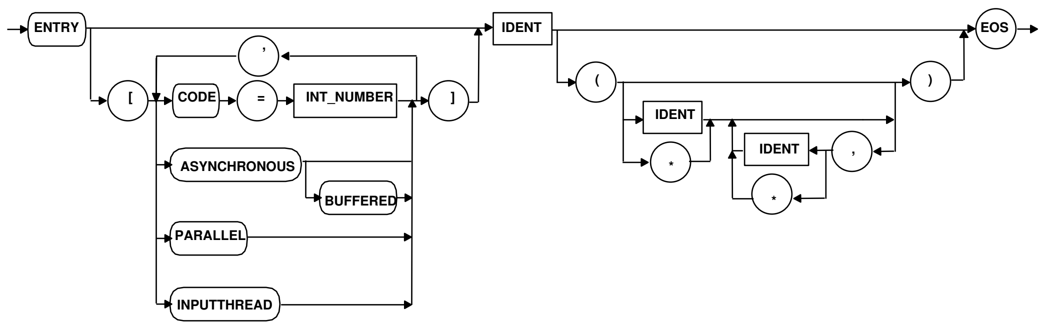
```
C*[RPI_BEG]
C     Alternative Return Jumps
C     ------------------------

      SUBROUTINE sub13 (n,*,*,*)
      INTEGER  n [IN]

C*[RPI_NON]
      i = max (1, min( -2+n/3, 3)
      RETURN i
      END
C*[RPI_END]
```

5.3.10 The Fortran ENTRY Statement

ENTRY [**CODE=***operation_code*, **ASYNCHRONOUS**] *identifier (...)* or **ASYNCHRONOUS BUFFERED** or **PARALLEL** or **INPUTTHREAD**

The qualifier contains the same elements as the C*[RPI_BEG] statement, see page 65f.

entry_stat



Example:

```
C*[RPI_BEG] CODE = 14
C     Entry Statements
C     ----------------

      SUBROUTINE sub14
      INTEGER  n [OUT]

C*[RPI_NON]
      ...
      RETURN

C*[RPI_CON]
      ENTRY [CODE = 15] sub15 (n)
C*[RPI_NON]
      n = 171717
      RETURN
      END
C*[RPI_END]
```

5.4 RPI Without the Fortran Source

It is basically the nonparseable blocks that are not included. These contain Fortran code and various other declarations which are not relevant for the stub generator. See example C in section 5.7.3 on page 91.

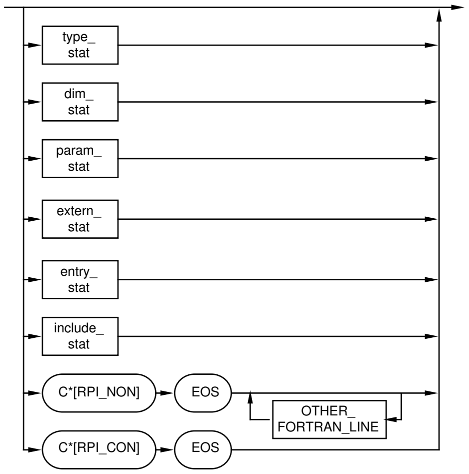
The stub generator must then be run with the **-n** option. See page 155.

5.5 Separate Block and Module Generation

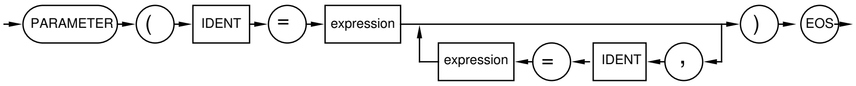
See example D in section 5.7.4 on page 92.

5.6 General Syntax of the Declaration Statements

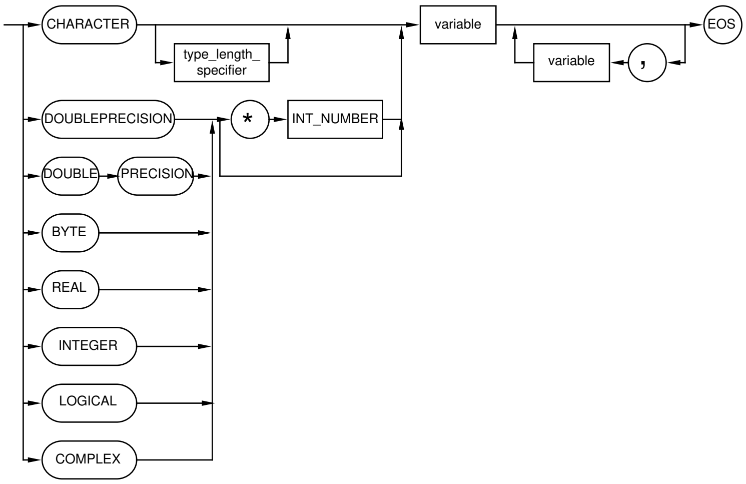
decl_stat



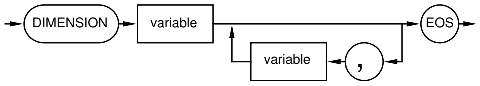
param_stat



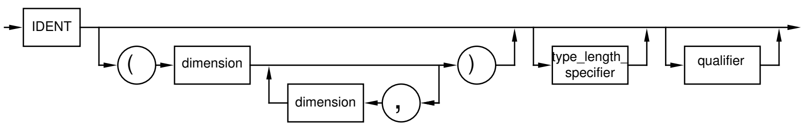
type_stat



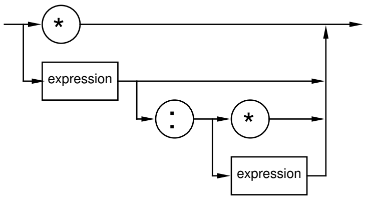
dim_stat



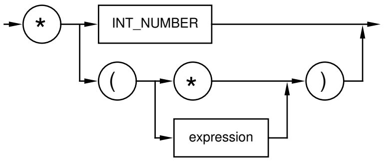
variable



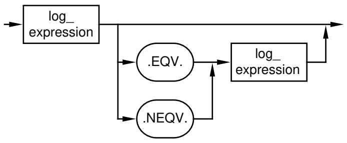
dimension



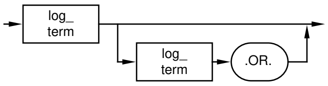
type_length_specifier



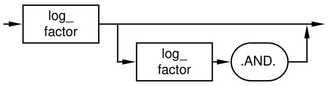
expression



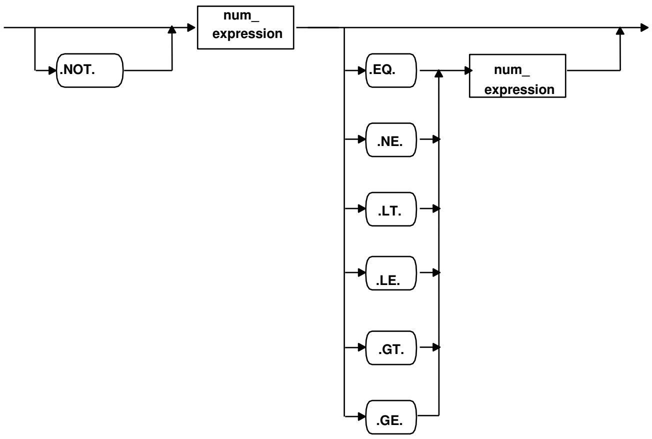
log_expression



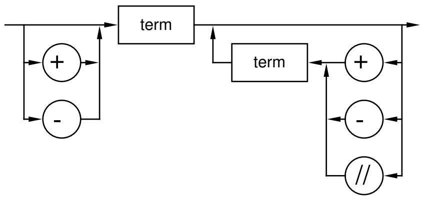
log_term



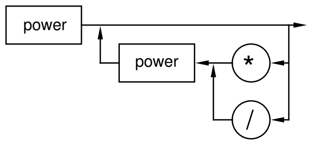
log_factor



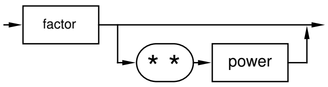
num_expression



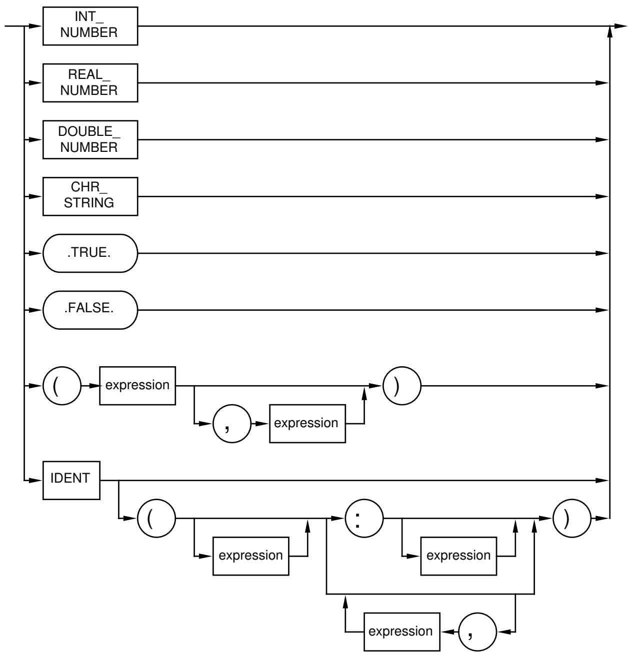
term



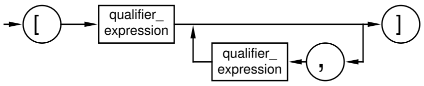
power



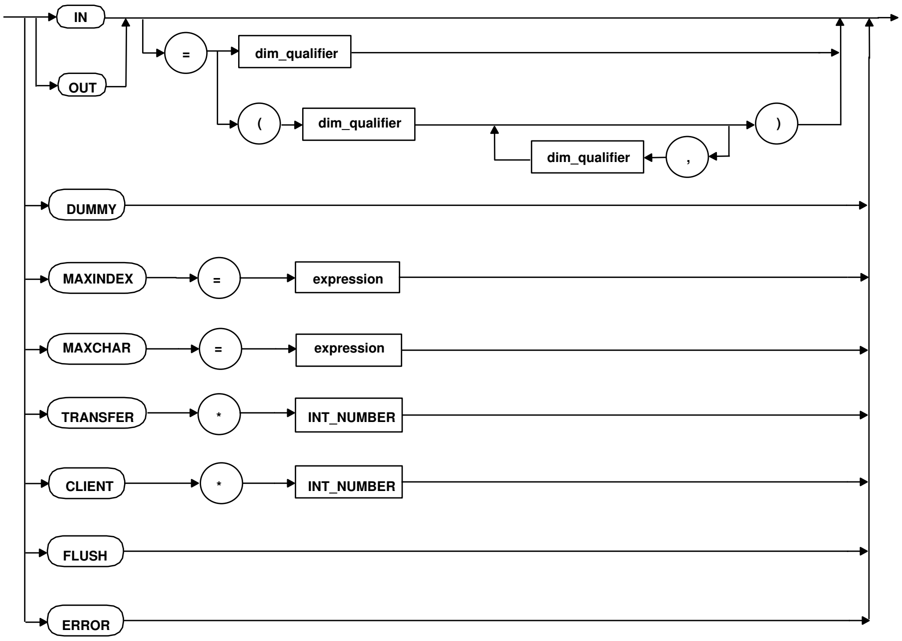
factor



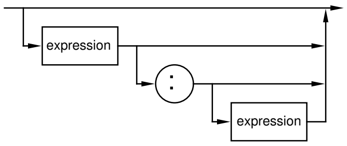
qualifier



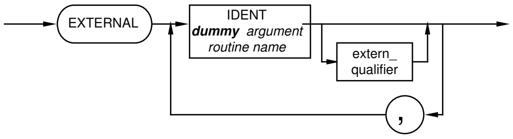
qualifier_expression



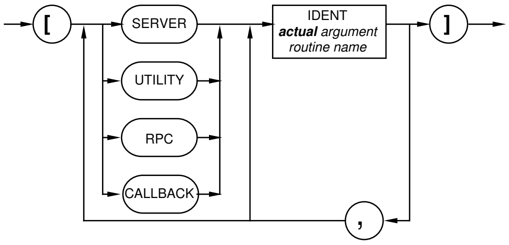
dim_qualifier



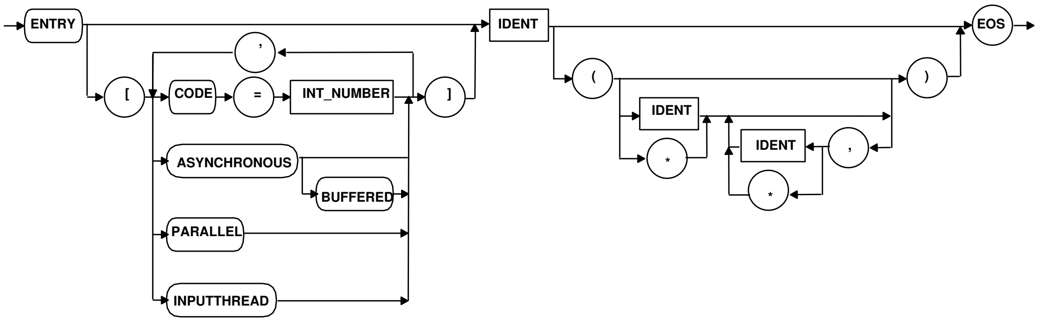
extern_statement



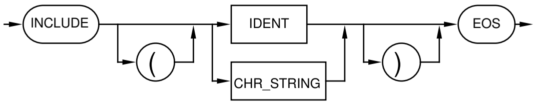
extern_qualifier



entry_stat



include_stat



5.7 RPI Examples

The following RPI definition files contain all relevant RPC scenarios, and how these can be used by the application programmer.

```
C     +-------------------------------------------------------+
C     |                  E X A M P L E S                      |
C     |                      F O R                            |
C     | R E M O T E   P R O C E D U R E   I N T E R F A C E |
C     |              D E F I N I T I O N S                    |
C     +-------------------------------------------------------+
```

5.7.1 Global Structure of the RPI Definition

```
C  A) The Global Structure of the Remote Interface Definition (RPI)
C     -------------------------------------------------------------

C     A RPI definition file consist of:
C
C       declaration part        with   options
C                                      block options
C                                      module parameters
C
C       block definition part   with   several block definitions
C
C       module definition part  with   several module definitions
C
C     These three parts must be written in this order.
C     Inside the parts any order is possible.
```

5.7.2 RPI Definition Embedded in Fortran Source Code

```
C  B) Example of a Remote Procedure Interface (RPI) Definition, embedded
C     into Fortran Source Code with a Client and a Server block
C     ------------------------------------------------------------------

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C     The following options must not be declared if one wants to use the
C     defaults. (The values used in this example are the defaults.)

C     The length values are given in bytes, the time values in seconds.


C*[OPTION] WORD_BUFFER_LENGTH      = 100000
C*[OPTION] CHARACTER_BUFFER_LENGTH =  10000
C*[OPTION] TRACE_DEPTH             =     20
C*[OPTION] OPERATION_BITS          =     10     ! Range 6..14
```

```
C*[OPTION] OBJECTS_MAXIMUM        =    512
C*[OPTION] WAIT_TIME_NEXT_CALL    =   1000
C*[OPTION] WAIT_TIME_SEND         =    300
C*[OPTION] WAIT_TIME_CONNECT      =    100
C*[OPTION] WAIT_TIME_ACCEPT       =   2000
C*[OPTION] WAIT_TIME_RELEASE      =    200
C*[OPTION] WAIT_TIME_LOCAL        =    100


C     The following block options are used by the authorization
C     mechanism. The default is zero. Please use an integer
C     random value between 0 and 999999999=10**9-1

C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321


C     With the following line one can define some options for the later
C     used and declared modules.

C*[MOD_PAR] sm, DEPTH=0, BACKDEPTH=0, CODE=1

C     DEPTH: If a module is used as a server in a block, then the stub
C            generator must generate not only the server stub for each
C            routine, it also must generate a module dispatcher routine
C            which interprets the header of each incoming remote
C            procedure call and calls the apropriate server stub routine.
C            If the called routine calls a second server or calls back a
C            routine in the client block and this one calls again the
C            server block then the block and module dispatcher routines
C            are called recursively, although no application routine is
C            called recursively.
C            DEPTH=0 means that the module dispatcher routine may be
C            called recursively.
C            DEPTH=d (>0) means that the stub generator must generate
C            d equivalent module dispatcher routines in this module
C            (if it is a SERVER module) to prevent the recursion.
C            This module DEPTH value will be used in the block dispatcher
C            routine, where the module dispatcher routine(s) must be
C            called.
C            Default: DEPTH=0.

C     BACKDEPTH:
C            This backdepth value is used in each subroutine client stub
C            of this module to generate the calls to the apropriate
C            number of block dispatcher routines. These calls are used
C            to handle callbacks (see also DEPTH).

C     CODE:  Each module which is used as CLIENT or SERVER in a block
C            must have a module code. It can be assigned in a MOD_PAR
C            statement or it can be automatically assigned by the stub
C            generator. This code must be the same in each block.
C            If one uses one file to generate all blocks then one can
C            use the automatic assignment. If one wants to generate
```

```
C            the blocks separately then the explicit assignment in the
C            MOD_PAR statement is recommended. The code must be in the
C            range 1 .. 2**(16-OPERATION_BITS)-1  or with the default of
C            10 operation bits, the module code range is 1..63.
C            Default: The automatic assignment is done when it is first
C            used in a MOD_USE statement as CLIENT or SERVER, see below.


C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C     The following block definitions are necessary.
C     They are using the OPTION, BLK_OPT and MOD_PAR
C     (only DEPTH and CODE) information.

C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     cp, LOCAL
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sp, LOCAL
C*[MOD_USE]     sm, SERVER
C*[BLK_END]

C*[BLK_BEG] lb, NAME='local program', DEPTH=0, LOCAL_SITE
C*[MOD_USE]     cm, LOCAL    ! only LOCAL is possible at LOCAL_SITE
C*[MOD_USE]     sm, LOCAL
C*[BLK_END]

C     CLIENT means that for each RPI routine in the module (here sm)
C            a client stub will be generated and embedded into this
C            block (here cb)
C     SERVER means first that the module itself is included into the
C            block and
C            second that for each RPI routine in the module (here sm)
C            a server stub will be generated and
C            third the generation of the module dispatcher routine and
C            fourth that this dispatcher routine will be called by the
C            block dispatcher routine.
C     LOCAL  means that only the module is embedded into the block.

C     CLIENT_SITE means that the block contains the application main
C            program.
C     SERVER_SITE means that the block is a server block, built by
C            a special main program which calls the server main loop
C            which is generated by the stub generator.
C     LOCAL_SITE means that remote procedure calls are not used.
```

```
C     ---------->>>>>>>> module definition part <<<<<<<<---------------

C     The following module definitions are necessary.
C     They are using the OPTION and MOD_PAR (only NAME and DEPTH)
C     information.

C*[MOD_BEG] cm, NAME='client module - main program'

      PROGRAM ...
      ...
C*RPC CALL rpcini            ! Those lines will be duplicated and
      ...                    ! the second activated in the generated
C*RPC CALL rpcfin            ! source of CLIENT_SITE and SERVER_SITE
      ...                    ! blocks.
      END
                             ! Source for the client module
      SUBROUTINE ...
      ...
      END

      FUNCTION ...
      ...
      END

C*[MOD_END]


C*[MOD_BEG] cp, NAME='client rpc special routines '

      SUBROUTINE rpcini      ! Examples will be provided
      ...
      END

      SUBROUTINE rpcfin
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sp, NAME='server rpc special program  '

      PROGRAM rpcsrv         ! Special main program for the server block
      ...
      CALL RPCLOP            ! The server main loop of this rpc service
      ...
      END

C*[MOD_END]


C*[MOD_BEG] sm, NAME='server module - server subroutine'

C*[RPI_BEG] CODE = 8         ! Two possibilities for all rpi routines
```

```
                            ! of a module: The routine codes are
                            ! explicitely defined always or never.
      SUBROUTINE sub1 (a,b,c)! RPI-Definition for this subroutine
      REAL a[IN], b[IN]      ! for generating client and server stub
      REAL c                 ! and source for this server module
C*RPC+      [OUT]            ! (only declarations, especially for
      ...                    ! the dummy arguments)
C*[RPI_NON]
      a = b * b              ! further source for this server module
      RETURN                 ! (further declarations and
      END                    ! executable statements)
C*[RPI_END]                  ! RPI_END is the signal for stub generation

      SUBROUTINE ...
      ...                    ! Further source for this module, but
      ...                    ! without stub generation, i.e.
      END                    ! not callable by remote blocks
      .
      .
C*[MOD_END]


C If this is on the input file inp.rpi then the stub generator will
C generate the following files:
C
C                                    inp.rpi
C                                       |
C               +-----------------------+-----------------------+
C For the       |                       |                       |
C blocks ..... cb                      sb                      lb
C               |                       |                       |
C following     |\                      |\                      |
C files will    | \                     | \                     |
C be generated: | inp_cb_stb.m4         | inp_sb_stb.m4         |
C               | |                     | |                     |
C        inp_cb.f inp_cb_stb.f    inp_sb.f inp_sb_stb.f    inp_lb.f
C
C
C Stub-Presourcefile:     inp_cb_stb.m4  and  inp_sb_stb.m4
C Stub-Sourcefile:        inp_cb_stb.f   and  inp_sb_stb.f
C Application-Sourcefile: inp_cb.f       and  inp_sb.f  and inp_lb.f
C
C Only in the case of CLIENT_SITE or SERVER_SITE (files inp_cb.f
C and inp_sb.f) all statements beginning with C*RPC will be duplicated
C and in the second line C*RPC will be substituted by five blanks.
C
C Then (also in the case C*[BLK_BEG] ... LOCAL)
C all qualifiers [...] between C*[RPI_BEG] / C*[RPI_CON] and
C C*[RPI_NON] / C*[RPI_END] will be substituted by blanks,
C here in the source files inp_cb.f, inp_sb.f and inp_lb.f.
C
```

5.7.3 RPI Definition Not Embedded in Fortran Source Code

```
C  C) Example of a Remote Procedure Interface Definition, not embedded
C     into Fortran Source Code
C     -----------------------------------------------------------------

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321

C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sm, SERVER
C*[BLK_END]

C     ---------->>>>>>>> module definition part <<<<<<<<----------------

C*[MOD_BEG] sm, NAME='server module - server subroutine'

C*[RPI_BEG]                  ! Here automatic operation CODE assignment
      SUBROUTINE sub1 (a,b,c)! is used.
      REAL a[IN], b[IN], c[OUT]
C     ...
C*[RPI_END]

C*[RPI_BEG]
      SUBROUTINE sub2
C     ...
C*[RPI_END]

C*[MOD_END]

C If this is on the input file inp.rpi then the stub generator will
C generate the following files (using -n flag)
C
C                                      inp.rpi
C                                         |
C               +------------------------+
C For the       |                         |
C blocks ..... cb                        sb
C               |                         |
C following      \                         \
C files will      \                         \
C be generated:   inp_cb_stb.m4             inp_sb_stb.m4
C                 |                         |
C                 inp_cb_stb.f              inp_sb_stb.f
```

5.7.4 RPI Definition in a Separate File

```
C  D) Example of a Remote Procedure Interface Definition, not embedded
C     into Fortran Source Code, with separated block and module gener.
C     ----------------------------------------------------------------

C File cb.rpi:
C ============

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...

C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321

C*[MOD_PAR] sm, DEPTH=0,CODE=1

C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] cb, NAME='client block', DEPTH=0, CLIENT_SITE
C*[MOD_USE]     sm, CLIENT
C*[BLK_END]



C File sb.rpi:
C ============

C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...

C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321

C*[MOD_PAR] sm, DEPTH=0,CODE=1

C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] sb, NAME='server block', DEPTH=0, SERVER_SITE
C*[MOD_USE]     sm, SERVER
C*[BLK_END]
```

```
C File sm.rpi:
C ============

C     ---------->>>>>>>> declaration part <<<<<<<<---------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...

C*[MOD_PAR] sm, DEPTH=0

C     ---------->>>>>>>> module definition part <<<<<<<<---------------

C*[MOD_BEG] sm, NAME='server module - server subroutine'

C*[RPI_BEG]
      SUBROUTINE sub1 (a,b,c)
      REAL a[IN], b[IN], c[OUT]
C     ...
C*[RPI_END]

C*[RPI_BEG]
      SUBROUTINE sub2
C     ...
C*[RPI_END]

C*[MOD_END]



C              cb.rpi            sb.rpi                  sm.rpi
C               |                 |                       |
C               |                 |                    +--+--+
C For the       |                 |                    |     |
C blocks ..... cb                sb                  (cb)   (sb)
C               |                 |                    |     |
C following     |                 |                    |     |
C files will    |                 |                    |     |
C be generated:cb_cb_stb.m4    sb_sb_stb.m4   sm_sm_clt.m4  sm_sm_srv.m4
C               |                 |                    |     |
C              cb_cb_stb.f     sb_sb_stb.f    sm_sm_clt.f   sm_sm_srv.f
C                              /   \   \ |                  /   \
C                         ____/     \   \|___             /     \
C The filenames          /           \       \           /       \
C originate from  input_filename block_name fixed  input_file module_name
C                                =====                        ======
```

5.7.5 Examples of Remote Procedure Interface Definitions

```
C  E) Examples of Remote Procedure Interface Definitions
C     ---------------------------------------------------
```

5.7.5.1 [IN], [OUT], [IN,OUT] und [DUMMY] Argumente

```
C*[RPI_BEG]
C     [IN], [OUT], [IN,OUT] and [DUMMY] Arguments
C     ------------------------------------------

      SUBROUTINE sub1 (a,b,c,d,e,x)
      REAL    a[OUT],  b[IN],  c[IN],  d, e[IN,OUT], x[DUMMY]

C     the default [in,out] is used for d
C     x is dummy, i.e. a workspace argument, i.e. it will not be
C     transferred, neither for input nor for output
C*[RPI_NON]
      a = b * c
      x = d
      d = e
      e = x
      RETURN
      END
C*[RPI_END]
```

5.7.5.2 Arrays and MAXINDEX

```
C*[RPI_BEG]
C     Arrays and MAXINDEX
C     -------------------

      SUBROUTINE sub2 (n,a,b,c)
      INTEGER n[IN]
      REAL    a(n) [OUT, MAXINDEX=n/2]
      REAL    b(n) [IN]
      REAL    c(n) [IN]

C     MAXINDEX overwrites the upper bound of the (last) dimension of
C     an array.
C     MAXINDEX must be given, if the (last) dimension is an asterisk.
C     MAXINDEX is used for memory allocation for the actual argument
C     in the server stub. MAXINDEX is also used as default and maximum
C     for the [IN] and [OUT] range (see next example).
C*[RPI_NON]
      DO 10  i=1,n/2
10       a(i) = b(i)*c(i) + b(i+n/2)*c(i+n/2)
      RETURN
      END
C*[RPI_END]
```

5.7.5.3 Arrays with specified IN/OUT Range

```
C*[RPI_BEG]
C     Arrays with specified IN/OUT Range
C     ----------------------------------

      SUBROUTINE sub3 (n,a)
      INTEGER n(2) [IN]
      REAL    a(*) [MAXINDEX=n(1)*n(2), IN=1:n(1), OUT=n(2)+1:n(2)+n(1)]

C*[RPI_NON]
      DO 10 i=1,n(1)
         a(n(2)+i) = a(i) * a(i)
10    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.7.5.4 Multidimensional Arrays

```
C*[RPI_BEG]
C     Multidimensional Arrays
C     -----------------------

      SUBROUTINE sub4 (a,n,nbeg,nend,m,mbeg,mend)
      INTEGER n[IN], nbeg[IN], nend[IN]
      INTEGER m[IN], mbeg[IN], mend[IN]
      REAL    a(n,*) [MAXINDEX = MAX(m,mend)           ,
     +                IN       = (nbeg:nend, mbeg:mend),
     +                OUT      = (nbeg:nend, m:m)      ]

C*[RPI_NON]
      DO 20 i = nbeg, nend
         sum = 0
         DO 10 j = mbeg, mend
            sum = sum + a(i,j) * a(i,j)
10       CONTINUE
         a(i,m) = sum
20    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.7.5.5 Character String Arguments

```
C*[RPI_BEG]
C     Character String Arguments
C     --------------------------

      SUBROUTINE sub5 (s1, s2, s3, s4, s5)

      CHARACTER*(*)  s1 [IN]
      CHARACTER*(*)  s2 [OUT=3]
      CHARACTER*(*)  s3 [IN, OUT, MAXCHAR=90]
      CHARACTER*(*)  s4 [DUMMY]
      CHARACTER*(*)  s5 [DUMMY,   MAXCHAR=500]

C*[RPI_NON]

C     If sub5 is called with the following actual argument list

C          CALL sub5 (a1(1:2), a2(11:15), a3(1:7), a4(41:100), ' ')

C     then s1 is declared as char.*2   and has got the value of a1(1:2)
C          s2 is declared as char.*5   and has got no value
C          s3 is declared as char.*90  and has got the value of a1(1:7)
C          s4 is declared as char.*60  and has got no value
C          s5 is declared as char.*500 and has got no value

C     After return  a1 will be unchanged
C                   a2(11:13) gets the content of s2(1:3)
C                   a3(1:7)   gets the content of s3(1:7)
C                   a4 will be unchanged
C                   a5 will be unchanged

      RETURN
      END
C*[RPI_END]

C Remarks:

C     The length of a string argument is always an invisible input
C     argument. The length of the allocated memory in the server is the
C     length of the actual argument and can be overwritten by the
C     MAXCHAR parameter.
C     The default of the [IN] and [OUT] range is the minimum of the
C     length of the actual argument and the MAXCHAR parameter.
```

5.7.5.6 Arrays of Strings

```
C*[RPI_BEG]
C     Arrays of Strings
C     -----------------

      SUBROUTINE sub6 (sarr,n)

      CHARACTER*(*)  sarr(*) [MAXINDEX=n, IN = (   ,1:n/2)  ,
     +                                    OUT= (1:3,n/2+1:n)]

C     In the IN and OUT ranges, the first dimension is the substring.
C     The omitted dimension in IN=(,...) is the same as 1:*, where *
C     is the string length, which can be overwritten by MAXCHAR.
C*[RPI_NON]
      DO 10  i=1,n/2
         sarr(n/2+i)(1:3) = func6 (sarr(n/2+i))
10    CONTINUE
      RETURN
      END
C*[RPI_END]
```

5.7.5.7 Reduced Transfer Precision

```
C*[RPI_BEG]
C     Reduced Transfer Precision and Check Modus
C     ------------------------------------------

      SUBROUTINE sub7 (i,j,x,y)

      INTEGER          i [IN,  TRANSFER *2]
      INTEGER          j [IN,  TRANSFER *1]
      DOUBLEPRECISION  x [OUT, TRANSFER *4]
      DOUBLE PRECISION y [OUT, TRANSFER *16]

C     TRANSFER *length   specifies the number of bytes that will be
C                        transferred for each value.
C                        Integer   : *1,*2,*4,*8
C                        Real      : *4,*8,*16
C                        Double    : *4,*8,*16
C                        Complex   : *8,*16,*32
C                        Logical   : *1,*2,*4,*8
C     To obtain full precision between two hosts with 64 bit
C     numerical units, one must qualify reals with TRANSFER*8
C     and doubles with TRANSFER*16, e.g. see argument y.
C*[RPI_NON]
      x = float(i)**j
      y = float(i)**j
      RETURN
      END
C*[RPI_END]
```

5.7.5.8 Differing Data Type Lengths Between Client and Server

```
C Client:
C -------
      REAL*8  x
      REAL*4  y
      CALL sub8 (x,y)

C Server:
C -------
C*[RPI_BEG]
C     Differring Declarations in the Client
C     -------------------------------------

      SUBROUTINE sub8 (x,y)

      REAL*4  x [CLIENT *8]  ! in server routine and server stub real*4
                             ! in the client stub real*8
      REAL*8  y [CLIENT *4]  ! in server routine and server stub real*8
                             ! in the client stub real*4

C     For choosing the number of bytes which will be transferred the
C     minimum of both declarations will be relevant.

C*[RPI_NON]
      d = x
      x = y
      y = d
      RETURN
      END
C*[RPI_END]
```

5.7.5.9 Summary of Numerical Data Types

```
C*[RPI_BEG]
C     Summary of numeric types
C     ------------------------

      SUBROUTINE sub11 (b, l,l1,l2,l4,l8, i,i1,i2,i4,i8,
     +                  r,r4,r8,r16, d, c,c8,c16,c32)

C            internal bytes on    SUN VAX CRAY2  | transfer
C            with compiler        f77 for cft77  | default
      BYTE            b     !      1   1    -    |   1
      LOGICAL         l     !      4   4    8    |   4
      LOGICAL*1       l1    !      1   1    8 w  |   1
      LOGICAL*2       l2    !      2   2    8 w  |   2
```

```
      LOGICAL*4        l4    !     4   4    8 w  |   4
      LOGICAL*8        l8    !     -   -    8 w  |   8
      INTEGER          i     !     4   4    8    |   4
      INTEGER*1        i1    !     -   -    -    |   1
      INTEGER*2        i2    !     2   2    8 w  |   2
      INTEGER*4        i4    !     4   4    8 w  |   4
      INTEGER*8        i8    !     -   -    8 w  |   8
      REAL             r     !     4   4    8    |   4
      REAL   *4        r4    !     4   4    8 w  |   4
      REAL   *8        r8    !     8   8    8 w  |   8
      REAL   *16       r16   !     -  16   16 w  |  16
      DOUBLE PRECISION d     !     8   8   16    |   8
      COMPLEX          c     !   4+4 4+4  8+8 w  | 4+4
      COMPLEX*8        c8    !   4+4 4+4  8+8 w  | 4+4
      COMPLEX*16       c16   !   8+8 8+8  8+8    | 8+8
      COMPLEX*32       c32   !     -   -    -    |  --

C                            w = the compiler issues a warning
C*[RPI_NON]
      RETURN
      END
C*[RPI_END]

C Remarks:

C     One can write DOUBLE PRECISION also without a blank:
C     DOUBLEPRECISION
C     The stub generator accepts non standard fortran data types
C     as listed above.
C     The number of transferred bytes depends only on the declaration
C     which can be changed by [TRANSFER *length].
C     It doesn't depend on the involved hosts (ieee, dec, vax, convex
C     or cray) nor on the chosen transfer coding (ieee, dec, vax or
C     cray).
```

5.7.5.10 Subroutines and Functions as Arguments (EXTERNALS)

```
C*[RPI_BEG]
C     Externals as dummy arguments
C     ----------------------------

      SUBROUTINE sub12 (e)

      EXTERNAL  e [SERVER   s1, s2, s3,
     +             UTILITY  u1, u2, u3, u4,
     +             RPI      r1,
     +             CALLBACK b1 ]

C*[RPI_NON]
      RETURN
      CALL e
      END
C*[RPI_END]
```

```
C Remarks:

C     Subroutine or function dummy arguments must be declared as
C     external in the subroutine or function, where it is in the
C     dummy argument list. (Normally it must be declared only in the
C     calling routine, i.e. if a routine name is used as an actual
C     argument.)
C     The whole range of possible actual values must be specified.
C     Also it must be specified, where those routines are located.

C     'mmm' should be the module in which sub12 is declared.
C     'sss' should be the block where sub12 is located as routine,
C     i.e. where mmm is declared (C*[MOD_USE]) as SERVER.
C     'ccc' should be the block from where sub12 is called,
C     i.e. where mmm is declared (C*[MOD_USE]) as CLIENT.
C     Then the following table illustrates the location of
C     the routines used as actual arguments for e
C     and on which host the actual arguments are used

C     declared as | ccc                |                sss | called on
C     ------------+--------------------+--------------------+----------
C     SERVER      |                    |           s1 s2 s3 |   sss
C     UTILITY     | u1 u2 u3 u4        |        u1 u2 u3 u4 |   sss
C     RPI         | Stub_with_entry_r1 --------> stub    r1 |   sss
C     CALLBACK    | b1   stub <--------- stub_with_entry_b1 |   ccc

C     In the case of SERVER, the stub generator will generate in the
C     client stub of mmm (here located in ccc) the corresponding
C     dummy routines, here s1, s2 and s3.

C     CAUTION: In all blocks where mmm is used as client stub or server
C              the EXTERNAL declaration must be valid, i.e. the role of
C              possible actual arguments (s1,s2,s3 / u1,u2,u3,u4 / r1 /
C              b1) must be se same (SERVER / UTILITY / RPI / CALLBACK).
```

5.7.5.11 Alternative Return Jumps

```
C*[RPI_BEG]
C     Alternative Return Jumps
C     ------------------------

      SUBROUTINE sub13 (n,*,*,*)
      INTEGER  n [IN]

C*[RPI_NON]
      i = max (1, min( -2+n/3, 3)
      RETURN i
      END
C*[RPI_END]
```

5.7.5.12 The Fortran ENTRY Statement

```
C*[RPI_BEG] CODE = 14
C     Entry Statements
C     ----------------

      SUBROUTINE sub14
      INTEGER  n [OUT]

C*[RPI_NON]
      ...
      RETURN

C*[RPI_CON]
      ENTRY [CODE = 15] sub15 (n)
C*[RPI_NON]
      n = 171717
      RETURN
      END
C*[RPI_END]
```

5.7.5.13 Asynchronous RPCs

```
C*[RPI_BEG] CODE = 16, ASYNCHRONOUS BUFFERED
C
C     Asynchronous Routines and Entries
C     ---------------------------------

      SUBROUTINE sub16
C*[RPI_NON]
      ...
      RETURN

C*[RPI_CON]
      ENTRY [CODE = 17, ASYNCHRONOUS] sub17
C*[RPI_NON]
      ...
      RETURN
      END
C*[RPI_END]
```

5.7.6 RPI Definition with Several Servers

```
C  F) Example of a Remote Procedure Interface (RPI) Definition
C     with Several Servers, embedded into one source file
C     --------------------------------------------------------
C
C     Block      b1                b2                b3                b4
C
C     Modules    P                 S                 S                 S
C                R
C                +->a_clt  a_srv->A
C                B<-b_srv  b_clt<-+
C
C                +->c_clt  c_srv->C         c_srv->C
C                                  D                 D
C                                  E                 +->e_clt  e_srv->E


C     The same picture, but with the possible connections at
C     execution time:
C
C     Block      b1                b2                b3                b4
C
C     Modules    P                 S                 S                 S
C                R
C                +---------------> A
C                B <---------------+
C
C                +---------------> C
C                +--------------------------------> C
C                                  D                 D
C                                  E                 +---------------> E


C     ---------->>>>>>>> declaration part <<<<<<<<----------------------

C*[OPTION] SEND_BUFFER_LENGTH      =   2400
C ...
C*[BLK_OPT] MASTER_STATIC_AUTHORIZATION = 123456789
C*[BLK_OPT] SLAVE_STATIC_AUTHORIZATION  = 987654321



C     ---------->>>>>>>> block definition part <<<<<<<<-----------------

C*[BLK_BEG] b1, NAME='client block b1', CLIENT_SITE
C*[MOD_USE]     p, LOCAL
C*[MOD_USE]     r, LOCAL
C*[MOD_USE]     a, CLIENT
C*[MOD_USE]     b, SERVER
C*[MOD_USE]     c, CLIENT
C*[BLK_END]
```

```
C*[BLK_BEG] b2, NAME='server block b2', SERVER_SITE
C*[MOD_USE]     s, LOCAL
C*[MOD_USE]     a, SERVER
C*[MOD_USE]     b, CLIENT
C*[MOD_USE]     c, SERVER
C*[MOD_USE]     d, LOCAL
C*[MOD_USE]     e, LOCAL
C*[BLK_END]

C*[BLK_BEG] b3, NAME='server block b3', SERVER_SITE
C*[MOD_USE]     s, LOCAL
C*[MOD_USE]     c, SERVER
C*[MOD_USE]     d, LOCAL
C*[MOD_USE]     e, CLIENT
C*[BLK_END]

C*[BLK_BEG] b4, NAME='server block b4', SERVER_SITE
C*[MOD_USE]     s, LOCAL
C*[MOD_USE]     e, SERVER
C*[BLK_END]

C*[BLK_BEG] loc, NAME='local program', LOCAL_SITE
C*[MOD_USE]     p, LOCAL
C*[MOD_USE]     a, LOCAL
C*[MOD_USE]     b, LOCAL
C*[MOD_USE]     c, LOCAL
C*[MOD_USE]     d, LOCAL
C*[MOD_USE]     e, LOCAL
C*[BLK_END]


C     ---------->>>>>>>> module definition part <<<<<<<<---------------

C*[MOD_BEG] p, NAME='main program module'
      PROGRAM ...
      ...
C*RPC CALL rpcini
      ...
C*RPC CALL rpcfin
      ...
      END

      SUBROUTINE ...
      ...
      END
      FUNCTION ...
      ...
      END
C*[MOD_END]

C*[MOD_BEG] r, NAME='special module - rpcini/rpcfin'
      SUBROUTINE rpcini
      ...
      END
```

```
      SUBROUTINE rpcfin
      ...
      END
C*[MOD_END]

C*[MOD_BEG] s, NAME='special module - server main'
      PROGRAM rpcsrv
      ...
      CALL RPBD1
      ...
      END
C*[MOD_END]


C*[MOD_BEG] a, NAME='module A'
C*[RPI_BEG] CODE = 1
      SUBROUTINE aaa1 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
      b = a * a
      WRITE (*,*) 'aaa1 - b=', b
      RETURN
      END
C*[RPI_END]
C
C*[RPI_BEG] CODE = 2
      SUBROUTINE aaa2 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
      b = a + a
      CALL bbb1 (b)
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] b, NAME='module B'
C*[RPI_BEG] CODE = 1
      SUBROUTINE bbb1 (b)
      REAL b[IN]
C*[RPI_NON]
      WRITE (*,*) 'bbb1 - b=', b
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] c, NAME='module C'
C*[RPI_BEG]
      SUBROUTINE ccc1 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
```

```
      b = a * a * a
      CALL ddd1 (b)
      RETURN
      END
C*[RPI_END]
C
C*[RPI_BEG]
      SUBROUTINE ccc2 (a,b)
      REAL a[IN], b[OUT]
C*[RPI_NON]
      b = a + a + a
      CALL eee1 (b)
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] d, NAME='module D'
C*[RPI_BEG]
      SUBROUTINE ddd1 (b)
      REAL b[IN]
C*[RPI_NON]
      WRITE (*,*) 'ddd1 - b=', b
      RETURN
      END
C*[RPI_END]
C*[MOD_END]


C*[MOD_BEG] e, NAME='module E'
C*[RPI_BEG]
      SUBROUTINE eee1 (b)
      REAL b[IN]
C*[RPI_NON]
      WRITE (*,*) 'eee1 - b=', b
      RETURN
      END
C*[RPI_END]
C*[MOD_END]
```

6 Library Routines

6.1 Overview – Basis Routines

```
block dispatcher                   - RPBD1  (IRET)
block dispatcher for one async.seq. - RPBD1A (IRET)
block dispatcher for one procedure  - RPBD1P (IRET)

data pipe, character string receive - RPDCOR (FLUSH, N, STR, IRET)
data pipe, character string send    - RPDCOS (FLUSH, N, STR, IRET)
data pipe, double precision receive - RPDDOR (FLUSH, N, ARR, IRET)
data pipe, double precision send    - RPDDOS (FLUSH, N, ARR, IRET)
data pipe, integer          receive - RPDIOR (FLUSH, N, ARR, IRET)
data pipe, integer          send    - RPDIOS (FLUSH, N, ARR, IRET)
data pipe, integer*1        receive - RPDI1R (FLUSH, N, ARR, IRET)
data pipe, integer*1        send    - RPDI1S (FLUSH, N, ARR, IRET)
data pipe, integer*2        receive - RPDI2R (FLUSH, N, ARR, IRET)
data pipe, integer*2        send    - RPDI2S (FLUSH, N, ARR, IRET)
data pipe, logical          receive - RPDLOR (FLUSH, N, ARR, IRET)
data pipe, logical          send    - RPDLOS (FLUSH, N, ARR, IRET)
data pipe, logical*1        receive - RPDL1R (FLUSH, N, ARR, IRET)
data pipe, logical*1        send    - RPDL1S (FLUSH, N, ARR, IRET)
data pipe, logical*2        receive - RPDL2R (FLUSH, N, ARR, IRET)
data pipe, logical*2        send    - RPDL2S (FLUSH, N, ARR, IRET)
data pipe, real             receive - RPDROR (FLUSH, N, ARR, IRET)
data pipe, real             send    - RPDROS (FLUSH, N, ARR, IRET)
data pipe, real*8           receive - RPDR8R (FLUSH, N, ARR, IRET)
data pipe, real*8           send    - RPDR8S (FLUSH, N, ARR, IRET)

monitor, delete user node           - RPMDUN (NODE, IRET)
monitor, set user node              - RPMSUN (NODE, VALUE, IRET)
object, wait u. current object ready- RPOCRY (TIMEMS, OBJRDY, IRET)
object, get current & client obj.id - RPOCUR (CUROBJ, CLTOBJ)
object, finish block dispatcher     - RPOFBD (IRET)
object, flush send buffer           - RPOFLS ()
object, wait u. general list ready  - RPOGRY (N, OBJID, TIMEMS, OBJRDY, OBJ1ST,
                                              nfds, readfds,writefds,exceptfds, rfds,IRET)
object, get socket descriptor       - RPOGSD (FD, IRET)
object, inspect return/message pack.- RPOISP (PACTYP, MODCOD, OPCODE, IRET)
object, wait u. object list ready   - RPOLRY (N, OBJID, TIMEMS, OBJRDY, OBJ1ST, IRET)
object, receive ret. in a thread    - RPORCT (IRET)
object, receive return package      - RPOREC (IRET)
object, wait u. o.& read-list ready - RPORRY (N, OBJID, TIMEMS, OBJRDY, OBJ1ST,
                                              NFD, RFD, RFDRDY, RFD1ST, IRET)
object, empty synchronous RPC       - RPOSYN (IRET)
object, flush & switch to other obj.- RPOUSE (OBJID, IRET)

quick start server-ser.netw.at clnt.- RPQSSC (NSERVER, SOBJID, NCONN, CONNE1,CONNE2, IRET)
quick start server-ser.netw.at serv.- RPQSSS (NSERVER, SNUM, NCONN, COBJID, SOBJID, IRET)
quick process start, finish at clnt.- RPQUCF (IRET)
quick process start, initi. at clnt.- RPQUCI (NSERVER, SOBJID, IRET)
quick process start, finish at serv.- RPQUSF (IRET)
quick process start, init   at serv.- RPQUSI (Master_Address, OBJID, IRET)
quick process start, program a.serv.- RPQUSP (Master_Address, IRET)
```

6.2 Overview – Advanced Routines

```
block, finish                      - RPBFIN (IRET)
block, initialization              - RPBINI (IRET)

object, accept  a transp.connection - RPOACC (OBJID, Slave_Address, Slave_Addr_Lng,
                                              Slave_Host_Lng, Slave_Port_Lng, IRET)
object, connect two servers        - RPOC2S (OBJID1, OBJID2, NHOST2,
                                              PIPID1, PIPID2, IRET)
object, close   a transp.connection - RPOCCL (IRET)
object, connect a transp.connection - RPOCON (Start_FLG, Master_Address, OBJID,
                                              Slave_Address, Slave_Addr_Lng,
                                              Slave_Host_Lng, Slave_Port_Lng, IRET)
object, close a port of a tr.conn.  - RPOPCL (IRET)
object, open  a port of a tr.conn.  - RPOPOP (Start_FLG, OBJID, Master_Address,
                                              Master_Addr_Lng, Master_Host_Lng,
                                              Master_Port_Lng, Master_Pass_Lng, IRET)
object, set local  trace flags      - RPOSLF (SOBJFL, COBJFL, SCT4FL, CCT4FL, IRET)
object, set remote trace flags      - RPOSRF (SOBJFL, COBJFL, SCT4FL, CCT4FL, IRET)

transport service, finish           - RPSFIN (IRET)
transport service, init             - RPSINI (Type_of_Tp_SerVice,
                                              Type_of_Network_Service,
                                              Type_of_Tp_SoftWare, Max_Outgoing_Conn,
                                              Max_Incoming_Conn, SERID, IRET)
transport service, list all interfa.- RPSLTI ()
transport service, use as current...- RPSUSE (SERID, IRET)

convert, receive buffer conversion  - RPVCRB (Darray, Dbytes, starti, stopi,
                                              Tbytes, Ttype,
                                              Rbuffer, RBread, RBused, Flags, IRET, *)
convert, send    buffer conversion  - RPVCSB (Darray, Dbytes, starti, stopi,
                                              Tbytes, Ttype,
                                              Sbuffer, SBused, SBsent, SBlength,
                                              Flags, IRET, *)
convert, receive buffer alignment   - RPVFRB (Rbuffer, RBread, RBused, DONE)
convert, send    buffer alignment   - RPVFSB (SBused)
convert, receive buffer initializ.  - RPVIRB (RBread, RBused)
convert, send    buffer initializ.  - RPVISB (SBused, SBsent)
```

6.3 Overview – Obsolete Routines

```
object, client=master normal finish - RPO1CF (IRET)
object, client=master normal start  - RPO1CS (Start_Flags, Master_Host_String,
                                              Object_Start_String, OBJID, IRET)
object, server=slave  normal finish - RPO1SF (IRET)
object, server=slave  normal start  - RPO1SS (OBJID, IRET)

object, client=master finish/restart- RPO2CF (Restart_Flag, IRET)
object, client=master initialization- RPO2CI (Start_Flags, Master_Host_String,
                                              Slave_Host_String, Object_Start_String,IRET)
object, client=master start/restart - RPO2CS (OBJID, IRET)
object, server=slave  finish/restart- RPO2SF (Restart_Flag, IRET)
object, server=slave  initialization- RPO2SI (OBJID, IRET)
object, server=slave  start/restart - RPO2SS (IRET)
```

```
object, client=slave  finish/restart- RPO3CF (Restart_Flag, IRET)
object, client=slave  start/restart - RPO3CS (OBJID, IRET)
object, server=master finish/restart- RPO3SF (Restart_Flag, IRET)
object, server=master initialization- RPO3SI (Start_Flags, Master_Host_String,
                                              Object_Start_String, OBJID, IRET)
object, server=master start/restart - RPO3SS (IRET)

object, get send & receive buf.len. - RPOGSR (LL_SND, LL_REC, LL_USE,
                                              LR_SND, LR_REC, LR_USE, IRET)
object, set send & receive buf.len. - RPOSSR (LL_SND, LL_REC, LR_SND, LR_REC, IRET)
```

6.4 Basis and Advanced Routines – alphabetical

6.4.1 RPBD1,RPBD1A,RPBD1P – Block Dispatcher

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPBD1  --- block dispatcher                    |
| Procedure      RPBD1A --- block dispatcher for one async.seq.|
| Procedure      RPBD1P --- block dispatcher for one procedure |
|                                                               |
| Synopsis       CALL RPBD1 (IRET)                              |
|                CALL RPBD1A(IRET)                              |
|                CALL RPBD1P(IRET)                              |
|                                                               |
| Task           Block Dispatcher:                              |
|                RPBD1 dispatches all incoming calls until a    |
|                finish package is received.                    |
|                RPBD1P dispatches only one remote Procedure    |
|                call.                                          |
|                RPBD1A dispatches an arbitrary number of       |
|                Asynchronous rpcs and at most once one         |
|                synchronous or parallel rpc. After dispatching|
|                the first rpc, RPBD1A continues with           |
|                dispatching further rpcs while the input       |
|                buffer is not empty at the beginning of a      |
|                new dispatching cycle.                         |
|                                                               |
|                At the beginning of RPBD1, RPBD1P or RPBD1A    |
|                the value of the 'current object identifier'   |
|                is stored as 'current client object ident'     |
|                that is used for receiving rpcs until the end |
|                of RPBD1, RPBD1P or RPBD1A.                    |
|                At the end of RPBD1, RPBD1P and RPBD1A         |
|                automatically RPOUSE('current client obj.id') |
|                is called, i.e. the 'current object ident'     |
|                is reset to the value it has had at the        |
|                beginning of RPBD1, RPBD1P or RPBD1A.          |
|                                                               |
| Parameter       Type            I/O Usage                     |
| IRET            integer         Out Return Code               |
|                                                               |
-----------------------------------------------------------------
```

6.4.2 RPBFIN – Block Finish

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPBFIN                                         |
|                                                               |
| Synopsis       CALL RPBFIN(IRET)                              |
|                                                               |
| Task           Block Finish                                   |
|                                                               |
| Parameter       Type            I/O Usage                     |
| IRET            integer         Out return code               |
|                                                               |
-----------------------------------------------------------------
```

6.4.3 RPBINI – Block Init

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPBINI                                         |
|                                                               |
| Synopsis       CALL RPBINI(IRET)                              |
|                                                               |
| Task           Block Init                                     |
|                                                               |
| Parameter       Type            I/O Usage                     |
| IRET            integer         Out return code               |
|                                                               |
-----------------------------------------------------------------
```

6.4.4 RPD... – Data Pipe Routines

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPDC0S/R  data message character     send/rec.|
| Procedure      RPDD0S/R  data message double precis.send/rec.|
| Procedure      RPDI0S/R  data message integer       send/rec.|
| Procedure      RPDI2S/R  data message integer*2     send/rec.|
| Procedure      RPDI1S/R  data message integer*1     send/rec.|
| Procedure      RPDL0S/R  data message logical       send/rec.|
| Procedure      RPDL2S/R  data message logical*2     send/rec.|
| Procedure      RPDL1S/R  data message logical*1     send/rec.|
| Procedure      RPDR0S/R  data message real          send/rec.|
| Procedure      RPDR8S/R  data message real*8        send/rec.|
|                                                               |
| Synopsis       CALL RPDC0S(FLUSH,N,STRING,IRET)               |
|                CALL RPDC0R(FLUSH,N,STRING,IRET)               |
|                                                               |
|                CALL RPDD0S(FLUSH,N,ARR,IRET)                  |
|                CALL RPDD0R(FLUSH,N,ARR,IRET)                  |
|                CALL RPDI0S(FLUSH,N,ARR,IRET)                  |
|                ...                                            |
```

```
|                                                                 |
| Task           RPD..S sends the data of the parameterlist      |
|                into a datapipe and RPD..R reads this data      |
|                from the pipe. For each call of RPD<xi>S        |
|                there must be a corresponding call to RPD<xi>   |
|                with the same type <xi> and in the same order.  |
|                                                                 |
|                RPD..S blocks if it should send more data than  |
|                the pipe can buffer. RPD..S blocks until the    |
|                data is received on the end of the pipe with    |
|                RPD..R or until the data is stored in the       |
|                buffers of the pipe (with flush=.false. the     |
|                DFN-RPC buffer is used, with flush=.true. the   |
|                network buffer is filled).                      |
|                                                                 |
|                RPD..R blocks until all data is received.       |
|                Therefore it is not necessary to wait with      |
|                RPOCRY, RPOLRY or RPOGRY until any data is      |
|                receiveable.                                    |
|                                                                 |
|                      RPD.._ In/Out                              |
| Parameter      Type         _S  _R  Usage                      |
| FLUSH          logical      In  Out .true. =flush send buffer  |
|                                     .false.=noflush=buffered   |
|                                     used in RPD..S             |
| N              integer      In  Out number of items in array  |
|                                     or characters in the       |
|                                     string to be transferred   |
| ARR            data_type(N) In  Out array                     |
|        RPDI0_: INTEGER(*)                                        |
|        RPDI2_: INTEGER*2(*)  if known, otherwise INTEGER(*)     |
|        RPDI1_: INTEGER*1(*)                                      |
|                or BYTE(*)    if known, otherwise INTEGER(*)     |
|        RPDI0_: LOGICAL(*)                                        |
|        RPDI2_: LOGICAL*2(*)  if known, otherwise LOGICAL(*)     |
|        RPDI1_: LOGICAL*1(*)  if known, otherwise LOGICAL(*)     |
|        RPDR0_: REAL(*)                                           |
|        RPDR8_: REAL*8(*)                                         |
|        RPDD0_: DOUBLE PRECISION(*)                               |
|        RPDC0_: CHARACTER*(*)   a string, not an array of char   |
|                                                                 |
| IRET           integer      Out Out return code               |
|   0 = okay                                                      |
| <>0 = error occurred                                            |
|   3 = invalid return/message  package counter tag              |
|   4 = invalid return/message  package operation tag            |
|  19 = object not opened or invalid current_object_handle       |
|                                                                 |
-----------------------------------------------------------------
```

6.4.5 RPMDUN – Monitor Delete User Node

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPMDUN --- RPc_Monitor_Delete_User_Node      |
|                                                             |
| Synopsis       CALL RPMDUN(NODE,IRET)                       |
|                                                             |
| Task           RPMDUN deletes a node in the MIB data base.  |
|                Can be used by an application to manage      |
|                a subtree in the monitor                     |
|                                                             |
| Argument       Type            I/O Usage                    |
| NODE           string          In  User node to be deleted  |
| IRET           integer         Out return code              |
|                                                             |
---------------------------------------------------------------
```

6.4.6 RPMSUN – Monitor Set User Node

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPMSUN --- RPc_Monitor_Set_User_Node         |
|                                                             |
| Synopsis       CALL RPMSUN(NODE,VALUE,IRET)                 |
|                                                             |
| Task           RPMSUN generates a new node in the MIB data  |
|                base.                                        |
|                Can be used by an application to manage      |
|                a subtree in the monitor                     |
|                                                             |
| Argument       Type            I/O Usage                    |
| NODE           string          In  User node to be generated |
| VALUE          string          In  Value for the node       |
| IRET           integer         Out return code              |
|                                                             |
---------------------------------------------------------------
```

6.4.7 RPOACC – Accept a Connection

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPOACC  --- RPc_Object_ACCept                |
|                                                             |
| Synopsis       CALL RPOACC(OBJID,Slave_Address,             |
|               +            Slave_Addr_Lng,Slave_Host_Lng,   |
|               +            Slave_Port_Lng,IRET)             |
|                                                             |
| Task           Accept Connection                            |
|                After the application has called RPOPOP and  |
|                has transferred the Master_Address to the    |
|                slave (where RPOCON must be called),         |
```

```
|                the master application must call RPOACC to    |
|                accept the connection.                        |
|                It does not matter which of RPOACC or RPOCON  |
|                is called first.                              |
|                Prior to the call of RPOACC, the object id.   |
|                given by RPOPOP must be set to the current    |
|                object identifier (this is done automatically |
|                by RPOPOP and must be done explicitely with   |
|                RPOUSE only if the current object id. was     |
|                changed in the meantime).                     |
|                If one is accepting more than one connection  |
|                to one port then the OBJID returned by RPOACC |
|                differs from that of RPOPOP.                  |
|                RPOACC sets the current object id. to the new |
|                one.                                          |
|                                                              |
| Remarks        See remarks of RPOPOP.                        |
|                                                              |
| Parameter       Type            I/O Usage                    |
|                                                              |
| OBJID           integer         Out object identifier        |
| Slave_Address   string          Out slave address (remote)   |
| Slave_Addr_Lng  integer         Out length of slave address  |
| Slave_Host_Lng  integer         Out length hostname portion  |
| Slave_Port_Lng  integer         Out length portname portion  |
| IRET            integer         Out return code              |
|                                                              |
----------------------------------------------------------------
```

6.4.8 RPOC2S – Connect Two Servers

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOC2S - RP_Object_Connect_2_Server           |
|                                                              |
| Synopsis       CALL RPOC2S(OBJID1,OBJID2,HOST2,              |
|               +            PIPID1,PIPID2,IRET)               |
|                                                              |
| Task           Called by the client, RPOC2S establishes a    |
|                data pipe between two server objects.         |
|                The pipe will be tested.                      |
|                The current object identifier is not changed  |
|                in the client nor in the servers.             |
|                RPOC2S blocks until both server objects are   |
|                able to execute a new remote procedure call.  |
|                In the server objects the current transport   |
|                service is used.                              |
| Caution: The number of incoming and outgoing connections in  |
|          the argument list of RPSINI must be large enough.   |
|          In the server with OBJID1 an outgoing connection    |
|          will be established, in the server with OBJID2 an   |
|          incoming one.                                       |
|                                                              |
```

```
| Parameter      Type           I/O Usage                    |
| OBJID1         integer        In  object identifier server 1|
| OBJID2         integer        In  object identifier server 2|
| HOST2          character*(*)  In  host address of server 2  |
|  ' '              = the same address as used for the client  |
|                     server connection  will be used          |
|                     (not yet implemented)                    |
|  'other contents' = this string is used as network address   |
| PIPID1         integer        Out object id of this pipe .. |
| PIPID2         integer        Out .. in server 1 and serv.2 |
| IRET           integer        Out return code               |
|   0 = okay                                                    |
| <>0 = error occurred, e.g.                                    |
|  17 = cannot create a new object handle (in object 1 or 2)   |
|  41 = invalid OBJID1                                          |
|  42 = invalid OBJID2                                          |
|  43 = invalid current service in object 1                     |
|  44 = invalid current service in object 2                     |
|  45 = transport layer error, e.g. connection not established |
|       due unknown host or port address                        |
|  50 = error in received test package                          |
----------------------------------------------------------------
```

6.4.9 RPOCCL – Close Connection

```
----------------------------------------------------------------
|                                                               |
| Procedure      RPOCCL  ---  RPc_Object_Connection_CLose       |
|                                                               |
| Synopsis       CALL RPOCCL (IRET)                             |
|                                                               |
| Task           Close a connection that is connected with      |
|                RPOACC or RPOCON.                              |
|                The current object identifier must point to    |
|                that connection (see RPOUSE).                  |
|                If this object id. is derived from a call to   |
|                RPOPOP then the current object id. is not      |
|                cleared and the port can be used for further   |
|                connections or can be closed with RPOPCL.      |
|                If it is derived from RPOACC or RPOCON then    |
|                the current object identifier is cleared.      |
|                                                               |
| Parameter       Type           I/O Usage                      |
| IRET            integer        Out return code                |
----------------------------------------------------------------
```

6.4.10 RPOCON – Connect

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOCON  --- RPc_Object_CONnect                |
|                                                              |
| Synopsis       CALL RPOCON(Start_Flags,Master_Address,       |
|               +            OBJID,Slave_Address,              |
|               +            Slave_Addr_Lng,Slave_Host_Lng,    |
|               +            Slave_Port_Lng,IRET)              |
|                                                              |
| Task           Connect to the master                         |
|                After a process (called master) has opened a  |
|                port with RPOPOP, an other process (called    |
|                slave) can connect to this port by calling    |
|                RPOCON. The connection must be accepted by    |
|                the master with RPOACC.                       |
|                It does not matter which of RPOACC or RPOCON  |
|                is called first.                              |
|                RPOCON sets the current object identifier     |
|                to the new one returned in OBJID.             |
|                                                              |
| Remarks        See also remarks of RPOPOP.                   |
|                                                              |
| Parameter       Type            I/O Usage                    |
| Start_Flags     integer         In  sum of following flags   |
|     - make object start and finish control output to stdout |
|                 0                = make output (normal)      |
|                 SFLG_Trace_Obj   = additional trace output   |
|     - make rpc and pipe call control output to stdout        |
|                 0                = make output (normal)      |
|                 SFLG_Trace_Call  = additional trace output   |
|                                                              |
| Master_Address  string          In  master address (remote)  |
| OBJID           integer         Out object identifier        |
| Slave_Address   string          Out slave address (local)    |
| Slave_Addr_Lng  integer         Out length of slave address  |
| Slave_Host_Lng  integer         Out length hostname portion  |
| Slave_Port_Lng  integer         Out length portname portion  |
| IRET            integer         Out return code              |
|                                                              |
----------------------------------------------------------------
```

6.4.11 RPOCRY – Wait 'Current' - Ready

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOCRY  ---  RPc_Object_Current_ReadY         |
|                                                              |
| Synopsis       CALL RPOCRY(TIMEMS,OBJRDY,IRET)               |
|                                                              |
| Task           Wait until a parallel RPC is                  |
|                ready for transmitting the output arguments   |
|                or a message can be received on a pipe        |
|                from the current object.                      |
|                                                              |
|                If a datapipe message or the return package of|
|                a parallel rpc is already read into the local |
|                buffer, then the system call "select" is not  |
|                done and OBJRDY=1 is returned.                |
|                This case can occur after                     |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an |
|                inspection with RPOISP.                       |
|                                                              |
| Remark:        It is not necessary to call RPOCRY before a   |
|                call to RPOISP, RPORCT, RPOREC or RPD..R to   |
|                wait until data can be received because they  |
|                (or the RPOREC in the thread of RPORCT)       |
|                are waiting internally until data is          |
|                receiveable.                                  |
|                                                              |
| After RPOCRY:  Upon successful waiting the application can   |
|                use RPOISP (not yet implemented) to           |
|                inspect whether it is a return package of a   |
|                parallel RPC or a message package and then    |
|                RPOREC to receive the return package or the   |
|                datapipe receive routine for message packages.|
|                                                              |
| Parameter      Type           I/O Usage                      |
| TIMEMS         integer        In  Timeout in millisec.       |
|  -1 = waiting infinitely                                     |
|  >= 0 waiting at most this amount of milliseconds            |
| OBJRDY         integer        Out object ready state         |
|  -2 = error occurred, details stored in IRET                 |
|  -1 = no parallel rpc active for the current object          |
|   0 = timeout reached                                        |
|   1 = parallel RPC output or message package can be received |
|       for the current object                                 |
| IRET           integer        Out return code                |
|   0 = okay                                                   |
| <>0 = error occurred                                         |
|  19 = object not opened or invalid current_object_handle     |
|  53 = error occurred in system call "select"                 |
----------------------------------------------------------------
```

6.4.12 RPOCUR – Get current Object Id

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOCUR  --- RPc_Object_inquire_CURrent_obj_id |
|                                                              |
| Synopsis       CALL RPOCUR(CUROBJ,CLTOBJ)                    |
|                                                              |
| Task           RPOCUR inquires the 'current object           |
|                identifier' (CUROBJ) used for calling rpc and |
|                all pipe operations.                          |
|                In a server process (i.e. RPBD1, RPBD1P or    |
|                RPBD1A is running)  the object identifier of  |
|                its client (CLTOBJ) is also returned.         |
|                                                              |
| Argument       Type            I/O Usage                     |
| CUROBJ         integer         Out current object identifier |
|                                    or -1 if it is not set    |
| CLTOBJ         integer         Out current client object id. |
|                                    or -1 if it is not set    |
----------------------------------------------------------------
```

6.4.13 RPOFBD – Finish Block Dispatcher

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOFBD --- RPc_Object_Finish_Block_Dispatcher |
|                                                              |
| Synopsis       CALL RPOFBD(IRET)                             |
|                                                              |
| Task           Finish Remote Block Dispatcher                |
|                RPOFBD called at client site finishs RPBD1    |
|                at server site.                               |
|                                                              |
| Parameter       Type            I/O Usage                    |
| IRET            integer         Out return code              |
----------------------------------------------------------------
```

6.4.14 RPOFLS – Flush Send Buffer

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOFLS --- Flush Send Buffer                  |
|                                                              |
| Synopsis       CALL RPOFLS()                                 |
|                                                              |
| Task           RPC FLuSh Call                                |
|                                                              |
|                Only after a RPSUSE or an explicit RPOFLS or  |
|                an implicit RPOFLS (done by synchronous rpc)  |
|                one can start a new object with RPO1..        |
----------------------------------------------------------------
```

6.4.15 RPOGRY – Wait 'Generalized' - Ready

```
---------------------------------------------------------------
|                                                             |
| Procedure      RPOGRY  ---  RPc_Object_Generalized_list_ReadY|
|                                                             |
| Synopsis       CALL RPOGRY(N,OBJID,TIMEMS,OBJRDY,OBJ1ST,    |
|               +            nfds,readfds,writefds,exceptfds, |
|               +            rfds,IRET)                       |
| Task           Wait until a parallel RPC is                 |
|                ready for transmitting the output arguments  |
|                or a message can be received on a pipe       |
|                from an object of the given list or          |
|                one other descriptor is ready.               |
|                If datapipe messages or return packages of   |
|                parallel rpcs are already read into local    |
|                buffers, then the system call "select" is not |
|                done and the returned arguments indicate     |
|                only those objects and rfds set to zero      |
|                and readfds, writefds and exceptfds remain   |
|                unchanged.  This case can occur after        |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an |
|                inspection with RPOISP.                      |
|                See also the Remark in the descrip. of RPOCRY.|
| After RPOGRY:  Upon successful waiting the application can  |
|                use RPOUSE to select a ready object, RPOISP to|
|                inspect whether it is a return package of a  |
|                parallel RPC or a message package and then   |
|                RPOREC to receive the return package or the  |
|                datapipe receive routine for message packages.|
|                                                             |
| Parameter      Type            I/O Usage                    |
| N              integer         In  Length of object list    |
| OBJID          integer(N)      In  Array of object identifier|
| TIMEMS         integer         In  Timeout in millisec.     |
|  -1 = waiting infinitely                                    |
|  >= 0 waiting at most this amount of milliseconds           |
| OBJRDY         integer(N)      Out object ready state       |
|  -2 = error occurred, for this object, e.g. invalid object id|
|  -1 = no parallel rpc active for this object of the list    |
|       and timeout reached for a possibly outstanding message |
|   0 = parallel rpc active for this object of the list       |
|       and timeout reached for parallel RPC output or message |
|   1 = parallel RPC output or message package can be received |
|       for this object of the list                           |
| OBJ1ST         integer         Out first ready object       |
|  -2 = error occurred, details stored in IRET, or N=0        |
|  -1 = no parallel rpc active for any object in OBJID        |
|       and timeout reached or a descriptor was first ready   |
|   0 = at least one parallel rpc is active and timeout reached|
|       or a descriptor (see below) was first ready           |
| 1..N= index of the first ready object in the list,          |
```

```
|      OBJID(OBJ1ST) is the object identifier of this object  |
|                                                             |
| nfds           integer         In  Number of used descriptors|
| readfds        integer(*)      I/O Read  file descriptor set |
| writefds       integer(*)      I/O Write file descriptor set |
| exceptfds      integer(*)      I/O Excep.file descriptor set |
| rfds           integer         Out Number of ready descrip-  |
|                                    tors or 0 if timed out    |
|                                    or an object was first    |
|                                    ready, or -1 if an error  |
|                                    is occurred in the select |
|                                    call                      |
|                                                             |
| IRET           integer         Out return code               |
|   0 = okay                                                  |
| <>0 = error occurred                                        |
|  53 = error occurred in system call "select"                |
|                                                             |
| See "man select" for description of nfds, readfds, writefds, |
| exceptfds and its return value, here stored in rfds.        |
| Timout is reached if rfds = 0 and OBJ1ST <= 0.              |
| RPOGRY merges the own socket descriptors and readfds before |
| calling "select" and upon calling they are removed from the |
| result (readfds, rfds).                                     |
|                                                             |
----------------------------------------------------------------
```

6.4.16 RPOGSD – Get Socket Descriptor

```
----------------------------------------------------------------
|                                                             |
| Procedure      RPOGSD  ---  RPc_Object_Get_Socket_Descriptor |
|                                                             |
| Synopsis       CALL RPOGSD(FD,IRET)                         |
|                                                             |
| Task           Get the I/O descriptor of the transport socket|
|                of the transport connection to the current   |
|                object.                                      |
|                                                             |
| Parameter      Type            I/O Usage                     |
| FD             integer         Out File descriptor or 0 if   |
|                                    there is no current object|
| IRET           integer         Out return code               |
|   0 = okay                                                  |
| <>0 = error occurred                                        |
|  19 = object not opened or invalid current_object_handle    |
----------------------------------------------------------------
```

6.4.17 RPOISP – Inspect next Package

```
--------------------------------------------------------------
| Procedure      RPOISP  ---  RPc_Object_InSpect_Package      |
|                                                             |
| Synopsis       CALL RPOISP(PACTYP,MODCOD,OPCODE,IRET)       |
|                                                             |
| Task           RPOISP waits (if necessary) until a packet has|
|                arrived and then RPORCT reads the header of  |
|                this package (and probably following ones)   |
|                into the internal buffer to inspect the      |
|                content of the package. Until the package is |
|                read with RPOREC or RPD..R a call to RPOISP  |
|                will always return the same header.          |
|                RPOISP must not be called while an RPORCT    |
|                is active on the same object.                |
|                                                             |
| Parameter      Type           I/O Usage                     |
| PACTYP         integer        Out package type              |
|                                     0 = synchronous call    |
|                                     1 = asynchronous call   |
|                                     2 = async. buffered call |
|                                     3 = parallel call       |
|                                     4 = return of sync. call |
|                                     5 = return of paral. call|
|                                     6 = datapipe package    |
|                                     7 = error package       |
| MODCOD         integer        Out module code               |
|                                     0 for dfnrpclib routines |
|                                    >0 application module    |
| OPCODE         integer        Out operation code            |
|                                     50 = RPDI0S/R           |
|                                     51 = RPDI2S/R           |
|                                     52 = RPDI1S/R           |
|                                     53 = RPDL0S/R           |
|                                     54 = RPDL2S/R           |
|                                     55 = RPDL1S/R           |
|                                     56 = RPDR0S/R           |
|                                     57 = RPDR8S/R           |
|                                     58 = RPDD0S/R           |
|                                     59 = RPDC0S/R           |
| IRET           integer        Out return code               |
|   0 = okay                                                  |
| <>0 = error occurred                                        |
|  19 = object not opened or invalid current_object_handle    |
|  58 = call not allowed while receiving output argu. w. RPORCT|
|                                                             |
--------------------------------------------------------------
```

6.4.18 RPOLRY – Wait 'List' - Ready

```
-----------------------------------------------------------------
| Procedure      RPOLRY  ---  RPc_Object_List_ReadY              |
|                                                                |
| Synopsis       CALL RPOLRY(N,OBJID,TIMEMS,OBJRDY,OBJ1ST,IRET)|
|                                                                |
| Task           Wait until a parallel RPC is                    |
|                ready for transmitting the output arguments     |
|                or a message can be received on a pipe          |
|                from an object of the given list.               |
|                If datapipe messages or return packages of      |
|                parallel rpcs are already read into local       |
|                buffers, then the system call "select" is not  |
|                done and the returned arguments are indicating|
|                only those objects. This case can occur after  |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an  |
|                inspection with RPOISP.                         |
|                See also the Remark in the descrip. of RPOCRY.|
| After RPOLRY:  Upon successful waiting the application can    |
|                use RPOUSE to select a ready object, RPOISP to|
|                inspect whether it is a return package of a    |
|                parallel RPC or a message package and then     |
|                RPOREC to receive the return package or the    |
|                datapipe receive routine for message packages.|
|                                                                |
| Parameter      Type            I/O Usage                       |
| N              integer         In  Length of object list       |
| OBJID          integer(N)      In  Array of object identifier|
| TIMEMS         integer         In  Timeout in millisec.        |
|  -1 = waiting infinitely                                       |
|  >= 0 waiting at most this amount of milliseconds              |
| OBJRDY         integer(N)      Out object ready state          |
|  -2 = error occurred, for this object, e.g. invalid object id|
|  -1 = no parallel rpc active for this object of the list       |
|       and timeout reached for a possibly outstanding message  |
|   0 = parallel rpc active for this object of the list          |
|       and timeout reached for parallel RPC output or message  |
|   1 = parallel RPC output or message package can be received  |
|       for this object of the list                              |
| OBJ1ST         integer         Out first ready object          |
|  -2 = error occurred, details stored in IRET, or N=0           |
|  -1 = no parallel rpc active for any object in OBJID           |
|       and timeout reached                                      |
|   0 = at least one parallel rpc is active and timeout reached|
| 1..N= index of the first ready object in the list,             |
|       OBJID(OBJ1ST) is the object identifier of this object   |
| IRET           integer         Out return code                 |
|   0 = okay                                                     |
| <>0 = error occurred                                           |
|  53 = error occurred in system call "select"                   |
-----------------------------------------------------------------
```

6.4.19 RPOPCL – Close Port

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOPCL  ---  RPc_Object_Port_Close             |
|                                                               |
| Synopsis       CALL RPOPCL (IRET)                             |
|                                                               |
| Task           Close a port that is opened with RPOPOP.       |
|                The current object identifier must point to    |
|                that port, see RPOUSE and RPOPOP.              |
|                Prior to the call of RPOPCL all connections    |
|                to this port must be closed with RPOCCL.       |
|                At return the current object id. is cleared.   |
|                                                               |
| Parameter       Type            I/O Usage                     |
| IRET            integer         Out return code               |
-----------------------------------------------------------------
```

6.4.20 RPOPOP – Open Port einer Verbindung

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOPOP  --- RPc_Object_Port_OPen               |
|                                                               |
| Synopsis       CALL RPOPOP(Start_Flags,OBJID,Master_Address,  |
|               +            Master_Addr_Lng,Master_Host_Lng,   |
|               +            Master_Port_Lng,Master_Pass_Lng,   |
|               +            IRET)                              |
|                                                               |
| Task           Port Open                                      |
|                It returns the address of an opened port.      |
|                The address is a string with the format        |
|                hostname:portname:password:                    |
|                The length of each portion is inclusive ':'.   |
|                The hostname is the official internet name     |
|                as stored in /etc/hosts or the numeric addr..  |
|                The password is a 20 byte alphanum. string.    |
|                The process that is calling RPOPOP is called |
|                the 'master' of that connection.               |
|                The current object id. is set to the returned |
|                OBJID.                                         |
|                                                               |
| Remarks        Establishing and releasing the connection is   |
|                done by the participating processes            |
|                'master' and 'slave'                           |
|                by calling ...          with the task ...      |
|                                                               |
|                RPOPOP                  opening the port       |
|                not part of DFN-RPC:    starting the slave and|
|                                        transmitting master's |
|                                        address to the slave   |
|                             RPOCON     connecting to master   |
```

```
|               RPOACC                  acceptig the connect. |
|                                                             |
|               Now the application can use the connection    |
|               for rpcs and as a data pipe.                  |
|                                                             |
|               RPOCCL       RPOCCL     closing the connection|
|               RPOPCL          -       closing the port      |
|                                                             |
|               It is all the same which of RPOACC and RPOCON |
|               and which of RPO1CF a. RPO1SF is called first.|
|                                                             |
| Parameter       Type            I/O Usage                   |
|                                                             |
| Start_Flags     integer         In  sum of following flags  |
|      - make object start and finish control output to stdout |
|                0                  = make output (normal)    |
|                SFLG_Trace_Obj   = additional trace output   |
|      - make rpc and pipe call control output to stdout      |
|                0                  = make output (normal)    |
|                SFLG_Trace_Call  = additional trace output   |
|                                                             |
| OBJID           integer         Out object identifier       |
| Master_Address  string          Out master address (local)  |
| Master_Addr_Lng integer         Out length of master address |
| Master_Host_Lng integer         Out length hostname portion |
| Master_Port_Lng integer         Out length portname portion |
| Master_Pass_Lng integer         Out length password portion |
| IRET            integer         Out return code             |
---------------------------------------------------------------
```

6.4.21 RPORCT – Receive Via a Thread

```
---------------------------------------------------------------
| Procedure      RPORCT  ---  RPc_Object_ReCeive_parallelreturn|
|                             in_a_second_Thread               |
| Synopsis       CALL RPORCT(IRET)                             |
|                                                              |
| Task           RPORCT can start a second thread which is     |
|                executing there a call to RPOREC.             |
|                The start of a second thread is only done     |
|                if threads are supported by the operating     |
|                system.                                       |
|                (In the moment the start of a thread is not   |
|                implemented.)                                 |
|                                                              |
| Parameter      Type           I/O Usage                      |
| IRET           integer        Out return code                |
|   0 = okay                                                   |
| <>0 = error occurred                                         |
|  19 = object not opened or invalid current_object_handle     |
|  55 = any parallel rpc is not active in the current object   |
|  58 = call not allowed while receiving output argu. w. RPORCT|
---------------------------------------------------------------
```

6.4.22 RPOREC – Receive Parallel Return

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPOREC  ---  RPc_Object_RECeive_parallelreturn|
|                                                               |
| Synopsis       CALL RPOREC(IRET)                              |
|                                                               |
| Task           RPOREC waits until a parallel RPC in the       |
|                current object is ready for transmitting the  |
|                output arguments.                              |
|                The output arguments are received and stored  |
|                into the original argument list.               |
|                RPOREC blocks until the output arguments can  |
|                be received. Therefore it is not necessary to |
|                wait with RPOCRY, RPOLRY or RPOGRY until data |
|                is receiveable.                                |
|                                                               |
|                If called by the application and after the    |
|                application has issued a RPORCT and RPORCT    |
|                has started a second thread, then this        |
|                routine waits only until the RPORCT thread    |
|                has finished.                                  |
|                                                               |
| Parameter      Type            I/O Usage                     |
| IRET           integer         Out return code               |
|   0 = okay                                                    |
| <>0 = error occurred                                          |
|  19 = object not opened or invalid current_object_handle     |
|  55 = any parallel rpc is not active in the current object   |
|  58 = call not allowed while receiving output argu. w. RPORCT|
|                                                               |
-----------------------------------------------------------------
```

6.4.23 RPORRY – Wait 'Read-List' - Ready

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPORRY  ---  RPc_Object_Read_list_ReadY       |
|                                                               |
| Synopsis       CALL RPORRY(N,OBJID,TIMEMS,OBJRDY,OBJ1ST,     |
|               +            NFD,RFD,RFDRDY,RFD1ST,IRET)        |
|                                                               |
| Task           Wait until a parallel RPC is                   |
|                ready for transmitting the output arguments   |
|                or a message can be received on a pipe        |
|                from an object of the given list or           |
|                one other descriptor is ready for read.       |
|                                                               |
|                If datapipe messages or return packages of    |
|                parallel rpcs are already read into local     |
|                buffers, then the system call "select" is not |
|                done and the returned arguments indicate      |
```

```
|                only those objects. This case can occur after |
|                the receipt of a (buffered) data message while|
|                the sender has sent a further one or after an |
|                inspection with RPOISP.                        |
|                See also the Remark in the descrip. of RPOCRY.|
|                                                               |
| After RPORRY:  Upon successful waiting the application can   |
|                use RPOUSE to select a ready object, RPOISP to|
|                inspect whether it is a return package of a   |
|                parallel RPC or a message package and then    |
|                RPOREC to receive the return package or the   |
|                datapipe receive routine for message packages.|
|                                                               |
| Parameter      Type            I/O Usage                     |
| N              integer         In  Length of object list     |
| OBJID          integer(N)      In  Array of object identifier|
| TIMEMS         integer         In  Timeout in millisec.      |
|  -1 = waiting infinitely                                     |
|  >= 0 waiting at most this amount of milliseconds            |
| OBJRDY         integer(N)      Out object ready state        |
|  -2 = error occurred, for this object, e.g. invalid object id|
|  -1 = no parallel rpc active for this object of the list     |
|       and timeout reached for a possibly outstanding message |
|   0 = parallel rpc active for this object of the list        |
|       and timeout reached for parallel RPC output or message |
|   1 = parallel RPC output or message package can be received |
|       for this object of the list                            |
| OBJ1ST         integer         Out first ready object        |
|  -2 = error occurred, details stored in IRET, or N=0         |
|  -1 = no parallel rpc active for any object in OBJID         |
|       and timeout reached or a descriptor was first ready    |
|   0 = at least one parallel rpc is active and timeout reached|
|       or a descriptor (see below) was first ready            |
| 1..N= index of the first ready object in the list,           |
|       OBJID(OBJ1ST) is the object identifier of this object  |
|                                                               |
| NFD            integer         In  Length of descriptor list |
| RFD            integer(NFD)    In  Array of read descriptors |
| RFDRDY         integer(NFD)    Out descriptor ready state    |
|   0 = no data seen at this filedescriptor                    |
|   1 = data can be received at this filedescriptor            |
| RFD1ST         integer         Out first ready descriptor    |
|  -2 = error occurred, details stored in IRET, or NFD=0       |
|   0 = timeout reached or an object (see above) is first ready|
| 1..N= index of the first ready descriptor in the list,       |
|       RFD(RFD1ST) is its file descriptor                     |
|                                                               |
| IRET           integer         Out return code               |
|   0 = okay                                                   |
| <>0 = error occurred                                         |
|  53 = error occurred in system call "select"                 |
|                                                               |
| Timout is reached if OBJ1ST <= 0 and RFD1ST <= 0.            |
----------------------------------------------------------------
```

6.4.24 RPOSLF,RPOSRF – Set Local/Remote Trace Flags

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPOSLF  ---  RP_Object_Set_Local_trace_Flags  |
|                RPOSRF  ---  RP_Object_Set_Remote_trace_Flags |
|                                                              |
| Synopsis       CALL RPOSLF (SOBJFL, COBJFL, SCT4FL,          |
|               +             CCT4FL, IRET)                    |
|                CALL RPOSRF (SOBJFL, COBJFL, SCT4FL,          |
|               +             CCT4FL, IRET)                    |
|                                                              |
| Task           Set the trace, monitor and transport flags.   |
|                                                              |
|                For each connection trace and monitor flags   |
|                are set with the Start_Flags argument while   |
|                starting the object with the routines         |
|                RPOPOP, RPOCON, RPO1CS, RPO2CI or RPO3SI      |
|                and the transport service flags are set       |
|                in the Type_of_Tp_SerVice argument of RPSINI. |
|                                                              |
|                RPOSLF:  The change is done for the local site|
|                         of the connection of the current     |
|                         object.                              |
|                                                              |
|                RPOSRF:  It can be called only on the client  |
|                         site while on the server site a      |
|                         block dispatcher is running.         |
|                         The changes are done on the server.  |
|                                                              |
|                With RPOSLF and RPOSRF the following flags    |
|                can be changed for the connection of the      |
|                current object at the site where RPOSLF is    |
|                called.                                       |
|                With SOBJFL and COBJFL one can set and clear  |
|                each trace or monitor flag:                   |
|                  OBJSTO - Trace Object                       |
|                  OBJSTC - Trace Calls                        |
|                  OBJSMO - Monitoring Off                     |
|                  OBJSM2 - Monitoring with sleep 2 sec in each|
|                           monitor step                       |
|                                                              |
|                With SCT4FL and CCT4FL one can set and clear  |
|                each transport service flags:                 |
|                  CT4PRT - print ct4 calls with argument list |
|                           (without buffers of CT4DIN, CT4DRQ)|
|                  CT4BUF - print buffers of CT4DIN and CT4DRQ |
|                  CT4PAU - pause at begin and end of ct4 calls|
|                                                              |
|                In each of these arguments any sum of the     |
|                mentioned flags can be used. The flags are    |
|                defined in the include file dfnrpc.inc .      |
|                The value 0 makes no change.                  |
```

```
|                The set operation is done first.                |
|                                                               |
| Parameter       Type           I/O Usage                      |
| SOBJFL          integer        In  Set   object flags         |
| COBJFL          integer        In  Clear object flags         |
| SCT4FL          integer        In  Set   ct4    flags         |
| CCT4FL          integer        In  Clear ct4    flags         |
| IRET            integer        Out return code                |
|   0 = okay                                                    |
| <>0 = error occurred                                          |
|  19 = object not opened or invalid current_object_handle      |
----------------------------------------------------------------
```

6.4.25 RPOSYN – Empty Synchronous RPC

```
----------------------------------------------------------------
|                                                               |
| Procedure      RPOSYN                                         |
|                                                               |
| Synopsis       CALL RPOSYN(IRET)                              |
|                                                               |
| Task           An empty synchronization call                  |
|                                                               |
| Parameter       Type           I/O Usage                      |
| IRET            integer        Out return code                |
----------------------------------------------------------------
```

6.4.26 RPOUSE – Select Current Object

```
----------------------------------------------------------------
|                                                               |
| Procedure      RPOUSE  --- RPc_Object_USE                     |
|                                                               |
| Synopsis       CALL RPOUSE(OBJID,IRET)                        |
|                                                               |
| Task           USE an alternate object for following          |
|                rpc and pipe operations.                       |
|                The parameter OBJID is stored as 'current      |
|                object identifier'.                            |
|                The 'current object identifier' is used for    |
|                calling rpcs and pipe operations.              |
|                At the beginning of RPBD1, RPBD1P or RPBD1A    |
|                the value of the 'current object identifier'   |
|                is stored as 'current client object ident.'    |
|                that is used for receiving rpcs until the end  |
|                of RPBD1, RPBD1P or RPBD1A.                    |
|                                                               |
|                If there is any information buffered for the   |
|                connection of the previous 'current object'    |
|                then and only then a new buffer must be used.  |
|                As default, the first buffer has a length      |
|                64 kByte, the next 8 buffers have each 4 kByte|
```

```
|               and the last 256 buffers have each 256 bytes. |
| Argument      Type           I/O Usage                      |
| OBJID         integer        In  object identifier          |
|                                  see RPO..I/S               |
| IRET          integer        Out return code                |
|  19 = object not opened or invalid current_object_handle    |
|                                                             |
---------------------------------------------------------------
```

(Changed functionality in comparison with DFN-RPC, Rel. 1.0, p. 99)

6.4.27 RPQSSC/S – Server-Server-Network

```
---------------------------------------------------------------
|                                                             |
| Procedure     RPQSSC    ---  RP_Quick_Server_Server_        |
|                                _interconnection_network     |
|                                _init_at_Client              |
|                                                             |
| Synopsis      CALL RPQSSC(NSERVER,SOBJID,NCONN,CONNE1,      |
|              +            CONNE2,IRET)                      |
|                                                             |
| Task          Initialization of a network of communication  |
|               links between multiple servers.               |
|                                                             |
|               RPQSSC must be called by the client.          |
|               In each Server RPQSSS must be called.         |
|                                                             |
|               The current object id. is not changed.        |
|               Invalid input in CONNE1 and CONNE2 will be    |
|               ignored.                                      |
|                                                             |
| Examples      See below and examples/sscon.                 |
|                                                             |
| Parameter     Type             I/O Usage                    |
| NSERVER       integer          In  number of servers        |
| SOBJID        integer(NSERVER) In  object id of each server|
| NCONN         integer          In  number of connctions     |
|                                    to be established        |
|                                    between the servers      |
| CONNE1        integer(NCONN)   In  server numbers of ...    |
| CONNE2        integer(NCONN)   In  ...the both endpoints    |
| IRET          integer          Out return code              |
|   0= okay                                                   |
| <>0= error occurred                                         |
|  63= error in a server while opening the port in RPQSSS     |
|  64= error in a server while connecting in RPQSSS           |
|  65= error in a server - no respoce for 60 sec -- fatal     |
|  and error numbers from RPOUSE, RPDIOS/R, RPOLRY, RPDCOS/R  |
|                                                             |
|  After a return with iret=63 or iret=64 some of the communi- |
|  cation links are established and the application can use   |
|  them.                                                      |
---------------------------------------------------------------
```

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPQSSS    ---  RP_Quick_Server_Server_        |
|                                 _interconnection_network      |
|                                 _init_at_Server               |
|                                                              |
| Synopsis       CALL RPQSSS(NSERVER,SNUM,NCONN,COBJID,        |
|               +            SOBJID,IRET)                      |
|                                                              |
| Task           Initialization of a network of communication  |
|                links between multiple servers.               |
|                                                              |
|                RPQSSS must be called in each server while    |
|                in the client RPQSSC is running.              |
|                                                              |
|                The current object id. must be the client     |
|                object id and it is not changed.              |
|                                                              |
| Examples       See below and examples/sscon.                 |
|                                                              |
| Parameter      Type             I/O Usage                    |
| NSERVER        integer          I/O number of servers        |
|                                     In:  declared length of  |
|                                          SOBJID              |
|                                     Out: its value in RPQSSC|
| SNUM           integer          Out number of this server    |
| NCONN          integer          Out number of connections    |
|                                     between this server and  |
|                                     the other servers        |
| COBJID         integer          Out object id of the client  |
| SOBJID         integer(NSERVER)  Out object id of the connec-|
|                                     tions to the other       |
|                                     servers or -1            |
| IRET           integer          Out return code              |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  66= did not get correct protocol unit from client''s RPQSSC |
|  67= NSERVER is to small, see also RPQSSC and dfnrpc.servers |
|  68= timeout (30s) reached while waiting for a connection    |
|  69= errors occurred in some other servers in RPQSSS         |
|  and error numbers from RPOUSE, RPDIOS/R, RPOPOP, RPOGSD,    |
|     RPDCOS/R, RPOCON, RPORRY, RPOACC                         |
|                                                              |
|  After a return with iret=11,12,68 or 69 some of the communi-|
|  cation links are established and the application can use    |
|  them.                                                       |
|                                                              |
----------------------------------------------------------------
```

```
-----------------------------------------------------------------
|    Example with arectangle of 2*3 servers                     |
|    --------------------------------------                     |
|                                                               |
|   [Client]__                                                  |
|    | | |\ \ \__c3_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _s1_            |
|    | | | \ \_c2_ _ _ _ _ _ _ _ __s1_               \           |
|    | | |  \c1_s1_              \                  \          |
|    | | |        \               \                  \         |
|    | | |         [S1]==============[S2]==============[S3]     |
|    | | |          ||  s3        s2  ||  s3        s2  ||      |
|    | | |          ||s2              ||s4              ||s3    |
|    | | |          ||                ||                ||      |
|    | |  \_c6__ _ _|| _ _ _ _ _ _ _ _|| _ _ _ _s1_     ||      |
|    |  \_c5__ _ _ _|| _ _ _ __s1_    ||         \      ||      |
|     \_c4_ __s1_   ||           \    ||          \     ||      |
|                \  ||            \   ||           \    ||      |
|                 \ ||s3           \  ||s2          \   ||s2    |
|                  \||  s2       s3 \||  s4       s3 \||        |
|                   [S4]==============[S5]==============[S6]    |
|                                                               |
|        [..]    Name of the process                            |
|        === ||  Connections between the servers established    |
|                by RPQSSC at [Client] and RPQSSS at [S1...S6]  |
|        _ _  |  Connections between client and server establ.  |
|                by RPQUCI at [Client] and RPQUSI at [S1...S6]  |
|        c1,...  Internal object id at the client process       |
|        s1,...  Internal object id at the server processes     |
|                                                               |
|        Input of RPQSSC                                        |
|          NSERVER = 6                     (Output of RPQUCI)   |
|          SOBJID  = (c1,c2,c3,c4,c5,c6)   (Output of RPQUCI)   |
|          NCONN   = 7                                          |
|          CONNE1  = (1, 2, 1, 2, 3, 4, 5)                      |
|          CONNE2  = (2, 3, 4, 5, 6, 5, 6)                      |
|                    vertically one can read the pairs of       |
|                    servers that should be connected           |
|          or if some connections are given more than once:     |
|          NCONN   = 13                                         |
|          CONNE1  = (1, 1, 2, 5, 6, 6, 2, 3, 4, 5, 6, 5, 6)    |
|          CONNE2  = (2, 2, 1, 6, 5, 3, 1, 2, 1, 2, 3, 4, 5)    |
|        Both sets of input will result in the same network     |
|        of server-server-connections.                          |
|                                                               |
|        Output of RPQSSS  at                                   |
|        Server:  NSERVER SNUM NCONN COBJID SOBJID              |
|           S1 :     6      1    2     s1   (-1 s3 -1 s2 -1 -1)|
|           S2 :     6      2    3     s1   (s2 -1 s3 -1 s4 -1)|
|           S3 :     6      3    2     s1   (-1 s2 -1 -1 -1 s3)|
|           S4 :     6      4    2     s1   (s3 -1 -1 -1 s2 -1)|
|           S5 :     6      5    3     s1   (-1 s2 -1 s3 -1 s4)|
|           S6 :     6      6    2     s1   (-1 -1 s2 -1 s3 -1)|
|                                            |  |  |  |  |  | |
|                Index of the array element:  1  2  3  4  5  6 |
```

```
| Client                                                          |
|    INTEGER NSERVER,SOBJID(512),NCONN,CONNE1(7),CONNE2(7),IRET|
|    NSERVER = 6                                                  |
|    CALL RPQUCI(NSERVER,SOBJID,IRET)                             |
|    NCONN=7 ; CONNE1=(1,2,1,2,3,4,5) ; CONNE2=(2,3,4,5,6,5,6) |
|    CALL RPQSSC(NSERVER,SOBJID,NCONN,CONNE1,CONNE2,IRET)         |
| Server                                                          |
|    INTEGER NSERVER,SNUM,NCONN,COBJID,SOBJID(512),IRET           |
|    CHARACTER*100 Master_Address                                 |
|    Master_Address=' '                                           |
|    IF (IARGC().GE.1) CALL GETARG(1,Master_Address)              |
|    CALL RPQUSI(Master_Address,COBJID,IRET)                      |
|    NSERVER = 512                                                |
|    CALL RPQSSS(NSERVER,SNUM,NCONN,COBJID,SOBJID,IRET)           |
 -----------------------------------------------------------------
```

6.4.28 RPQUCF – Quick Start - Client Finish

```
-----------------------------------------------------------------
| Procedure      RPQUCF    ---  RP_QUick_Client_site_Finish     |
|                                                               |
| Synopsis       CALL RPQUCF(IRET)                              |
|                                                               |
| Task           Finish of a client object.                     |
|                It closes the connections established by       |
|                RPQUCI.                                        |
|                                                               |
|                Internally RPQUCF is calling for each server    |
|                  RPOUSE (OBJID,IRET)                          |
|                    with OBJID stored by RPQUCI                |
|                  RPO1CF (IRET)                                |
|                and then                                       |
|                  RPSFIN (IRET)                                |
|                  RPBFIN (IRET)                                |
|                                                               |
| Examples       See example rshexa, crshexa and rpcpipe.       |
|                                                               |
| Parameter      Type              I/O Usage                    |
| IRET           integer           Out return code              |
|   0= okay                                                     |
| <>0= error occurred                                           |
|  see error numbers from RPOUSE, RPO1CF, RPSFIN and RPBFIN     |
-----------------------------------------------------------------
```

6.4.29 RPQUCI – Quick Start - Client Init

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPQUCI    ---  RP_QUick_Client_site_Init       |
|                                                               |
| Synopsis       CALL RPQUCI(NSERVER,SOBJID,IRET)               |
|                                                               |
```

```
| Task           Initialization of a client object.             |
|                                                               |
|                If there is a file named dfnrpc.servers in the|
|                current working directory, then each line of  |
|                it is used to start a server process by       |
|                calling system (see man-page of system(3F)).  |
|                Not more than NSERVER servers are started.    |
|                Lines containing only blanks or beginning with|
|                # are treated as comment. If there is a line  |
|                beginning with -norepeat and the number of    |
|                server starting lines in the file is less than|
|                the argument NSERVER then the argument NSERVER|
|                is reduced to this number of servers;         |
|                otherwise the server starts are repeated until|
|                the number given by NSERVER is reached.       |
|                Each server start consist of the following    |
|                steps:                                        |
|                - opening a port for the transport connection,|
|                - substituting the keyword DFNRPCADDR in the  |
|                  server start line by the address of this    |
|                  port,                                       |
|                - starting the server by calling 'system',    |
|                - waiting for and accepting the connection,   |
|                  established by the server routine RPQUSP.   |
|                The first three steps are done in a first loop|
|                and the last step is done in a second loop.   |
|                The command on each server start line must    |
|                finish immediately (e.g. by a & sign at the   |
|                end of the line). Therefore all starts of the |
|                servers are running partially parallel.       |
|                                                               |
|                If there is no such file, RPQUCI asks to start|
|                each server interactively.                    |
|                                                               |
|                RPQUCI establishes the connections to all     |
|                servers.                                      |
|                                                               |
|                Internally RPQUCI calls                       |
|                  RPBINI (IRET)                               |
|                  RPSINI (7,0,0,InCon,OutCon,ISERID,IRET)     |
|                  with InCon  incoming connections with       |
|                       InCon  = MAX(OBJECTS_MAXINST/2,NSERVER)|
|                  and  OutCon outgoing connections with       |
|                       OutCon = OBJECTS_MAXINST-InCon         |
|                  and OBJECTS_MAXINST = 512                   |
|                  and the initial value of NSERVER.           |
|                and for each server:                          |
|                  RPOPOP (0,OBJID,M_Address,M_Addr_Lng,       |
|                       M_Host_Lng,M_Port_Lng, M_Pass_Lng,IRET)|
|                  SYSTEM, ISHELL or interactive start         |
|                  RPOACC (OBJID,S_Address,S_Addr_Lng,         |
|                          S_Host_Lng,S_Port_Lng, IRET)        |
|                and at the end RPOUSE with the OBJID of the   |
|                first server.                                 |
```

```
|                To parallelize the server starts the calls to |
|                POPOP and SYSTEM are done in a first loop and |
|                RPOACC is called in a second loop.            |
|                All object ids are stored globally for RPQUCF.|
|                                                              |
| Examples       See example rshexa, crshexa and rpcpipe.      |
|                                                              |
| Parameter      Type             I/O Usage                    |
| NSERVER        integer          I/O number of servers        |
| SOBJID         integer(NSERVER) Out object id of each server|
| IRET           integer          Out return code              |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  59= error in opening existing file dfnrpc.servers on unit 21|
|  60= error in reading from file dfnrpc.servers               |
|  61= more than 512 chars in a resulting server start line    |
|  and error numbers from RPBINI, RPSINI, RPOPOP and RPOACC    |
| Example for dfnrpc.servers:                                  |
|  # Servers startet with a remote shell:                      |
|  rsh servername_1 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  rsh servername_2 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  rsh servername_3 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  rsh servername_4 -n rshexa_sb DFNRPCADDR > /dev/null &      |
|  #CAUTION: to see all error messages issued by 'rsh'         |
|  #         one must omit '>/dev/null'                        |
|                                                              |
|  # The following server must be started with telnet:         |
|  echo " ";echo "start anywhere rshexa_sb DFNRPCADDR";echo " "|
|                                                              |
|  # The following line prevents, that the server list is      |
|  # repeated to start the number of servers wanted by client  |
|  -norepeat                                                   |
----------------------------------------------------------------
```

6.4.30 RPQUSF – Quick Start - Server Finish

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPQUSF    ---  RP_QUick_Server_site_Finish    |
|                                                              |
| Synopsis       CALL RPQUSF(IRET)                             |
|                                                              |
| Task           Finish of a server object.                    |
|                It closes the connection  established by      |
|                RPQUSI.                                       |
|                                                              |
|                Internally RPQUSF is calling                  |
|                  RPOUSE (OBJID,IRET)                         |
|                    with OBJID from stored by RPQUSI          |
|                  RPO1SF (IRET)                               |
|                  RPSFIN (IRET)                               |
|                  RPBFIN (IRET)                               |
|                                                              |
```

```
| Examples       See example rpcpipe and con.                   |
|                                                               |
| Parameter                Type    I/O Usage                    |
| IRET                     integer Out return code              |
|   0= okay                                                     |
| <>0= error occurred                                           |
|  see error numbers from RPOUSE, RPO1SF, RPSFIN and RPBFIN     |
|                                                               |
-----------------------------------------------------------------
```

6.4.31 RPQUSI – Quick Start - Server Init

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPQUSI    ---  RP_QUick_Server_site_Init.      |
|                                                               |
| Synopsis       CALL RPQUSI(Master_Address,OBJID,IRET)         |
|                                                               |
| Task           Initialization of a server object.             |
|                                                               |
|                Master_Address should contain the address of   |
|                the port opened by the client routine RPQUCI.  |
|                If its value is ' ' then the address of the    |
|                client is asked interactively.                 |
|                The current object id. is set to the returned  |
|                OBJID.                                         |
|                                                               |
|                Internally RPQUSI calls                        |
|                  RPBINI (IRET)                                |
|                  RPSINI (7,0,0,max_in_conn,max_out_conn,      |
|                          ISERID,IRET)                         |
|                  RPOCON(0, M_Address, OBJID, S_Address,       |
|                     S_Addr_Lng, S_Host_Lng, S_Port_Lng, IRET)|
|                                                               |
|                RPSINI is called with a maximum                |
|                  of  OBJECTS_MAXINST/2 incoming connections   |
|                  and OBJECTS_MAXINST/2 outgoing connections.  |
|                                                               |
|                OBJID is stored globally for RPQUSF.           |
|                                                               |
| Examples       See example rpcpipe and con.                   |
|                                                               |
| Parameter                Type    I/O Usage                    |
| Master_Address           string  In  master address (remote)  |
| OBJID                    integer Out object id of the client  |
| IRET                     integer Out return code              |
|   0= okay                                                     |
| <>0= error occurred                                           |
|  see error numbers from RPBINI, RPSINI, RPOCON                |
|                                                               |
-----------------------------------------------------------------
```

6.4.32 RPQUSP – Quick Start - Server Program

```
---------------------------------------------------------------
| Procedure      RPQUSP    ---  RP_QUick_Server_site_Program   |
|                                                              |
| Synopsis       CALL RPQUSP(Master_Address, IRET)             |
|                                                              |
| Task           Special main program of the server site       |
|                                                              |
|                Master_Address should contain the address of  |
|                the port opened by the client routine RPQUCI. |
|                If its value is ' ' then the address of the   |
|                client is asked interactively.                |
|                The current object id. is set to that of the  |
|                client.                                       |
|                                                              |
|                Internally RPQUSP calls                       |
|                  RPQUSI (Master_Address,OBJID,IRET)          |
|                  RPBD1  (IRET)                               |
|                  RPQUSF (IRET)                               |
|                                                              |
| Examples       See example rshexa and crshexa.               |
|                                                              |
| Parameter                Type    I/O Usage                   |
| Master_Address           string  In  master address (remote) |
| IRET                     integer Out return code             |
|   0= okay                                                    |
| <>0= error occurred                                          |
|  see error numbers from RPBINI, RPSINI, RPOCON, RPBD1,       |
|                         RPO1SF, RPSFIN and RPBFIN            |
---------------------------------------------------------------
```

6.4.33 RPSFIN – Transport Service Finish

```
---------------------------------------------------------------
|                                                              |
| Procedure      RPSFIN                                        |
|                                                              |
| Synopsis       CALL RPSFIN(IRET)                             |
|                                                              |
| Task           Finish of a transport Service                 |
|                                                              |
| Argument        Type            I/O Usage                    |
| IRET            integer         Out return code              |
|                                                              |
---------------------------------------------------------------
```

6.4.34 RPSINI – Transport Service Init

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPSINI                                         |
|                                                               |
| Synopsis       CALL RPSINI(Type_of_Tp_SerVice,                |
|                            Type_of_Network_Service,           |
|                            Type_of_Tp_SoftWare,               |
|                            Max_Outgoing_Connections,          |
|                            Max_Incoming_Connections,          |
|                            SERID, IRET)                       |
|                                                               |
| Task           Initialization of a transport Service          |
|                                                               |
| Argument                 Type    I/O Usage                    |
| Type_of_Tp_SerVice       integer In  Type of TransportService|
|           values: CT4TCP or CT4TC4                            |
|                  (CT4TC0-3,CT4RTS,CT4T70 are not implemented|
|                   possibly plus flags CT4PRT,CT4BUF,CT4PAU, |
|                                       CT4FFL, CT4PFL          |
| Type_of_Network_Service  integer In  Type of Network Service |
|           values: CT4N0                                       |
| Type_of_Tp_SoftWare      integer In  Type of Transp. Software|
|           values: CT4S0 or CT4SUL                             |
| Max_Outgoing_Connections integer In  Max. of Outgoing Connec.|
|           value in the client: "number of servers"            |
|           value in the server: 1                              |
| Max_Incoming_Connections integer In  Max. of Incoming Connec.|
|           value in the client: "number of servers"            |
|           value in the server: 1                              |
| SERID                    integer Out object identifier        |
| IRET                     integer Out return code              |
|                                                               |
-----------------------------------------------------------------
```

6.4.35 RPSLTI – Transport Service List

```
-----------------------------------------------------------------
|                                                               |
| Procedure      RPSLTI                                         |
|                                                               |
| Synopsis       CALL RPSLTI()                                  |
|                                                               |
| Task           List all Transport Interface parameters        |
|                                                               |
| Argument        Type            I/O Usage                     |
| (none)                                                        |
|                                                               |
-----------------------------------------------------------------
```

6.4.36 RPSUSE – Transport Service Select

```
----------------------------------------------------------------
|                                                              |
| Procedure      RPSUSE                                        |
|                                                              |
| Synopsis       CALL RPSUSE(SERID, IRET)                      |
|                                                              |
| Task           Flush communication with current object and   |
|                USE an alternate and already opened service   |
|                for next object start                         |
|                                                              |
| Argument        Type           I/O Usage                     |
| SERID           integer        In  service identifier        |
|                                    see RPSINI                |
| IRET            integer        Out return code               |
|                                                              |
----------------------------------------------------------------
```

6.4.37 RPV... – Conversion Routines

6.4.37.1 User Data → Transfer Encoding

```
================================================================
|                                                              |
| Module         SENDCONVERT                                   |
|                -----------                                   |
|                                                              |
| Purpose        Routine for INTEGER- and REAL-number          |
|                conversion from local to transfer enconding.  |
|                                                              |
| Exported subroutines :                                       |
|                                                              |
|   1.) RPVISB   -   Initialisation of the send buffer         |
|                                                              |
|   2.) RPVFSB   -   Make send buffer ready for transmission   |
|                                                              |
|   3.) RPVCSB   -   Main conversion subroutine, converts data |
|                     from the data array into the send buffer |
|                                                              |
================================================================
```

```
Module description of SENDCONVERT
=================================


  General information on variables used by the exported subroutines :
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


  1.) variables belonging to the data array
```

```
    - DATA_array        source array of data, the type of the array
                        elements is free.
    - DATA_bytes        length of a DATA_array element in bytes
    - start_index       conversion of DATA_array starts from
                        this index
    - stop_index        conversion goes until this index - 1


2.) variables belonging to the transfer side

    - TRANSFER_bytes    length of a SEND_buffer element in bytes
    - TRANSFER_type     transfer enconding of a converted element :
                              =  1    ->  INTEGER, BIG    ENDIAN
                              =  2    ->  INTEGER, LITTLE ENDIAN
                              =  3    ->  LOGICAL, BIG    ENDIAN
                              =  4    ->  LOGICAL, LITTLE ENDIAN
                              =  5    ->  REAL,    IEEE
                              =  6    ->  REAL,    CRAY
                              =  7    ->  REAL,    DEC
                              =  8    ->  REAL,    VAX

3.) variables belonging to the send buffer

     - SEND_buffer       send buffer
     - SB_used           the last position used in SEND_buffer
                         in bytes
     - SB_sent           the last position (in bytes) in SEND_buffer
                         which has been sent already. This variable
                         makes it possible that the send buffer can be
                         cleared of elements which have been already
                         sent, and so the sending of parts of the send
                         buffer is possible.
     - SB_length         length of SEND_buffer in INTEGER words.
                         Internally, only a length which can
                         be devided by 16 (bytes) is used.
4.) misc. variables

           - FLAGS       INTEGER variable in which some different
                         commands can be given to the conversion
                         subroutine :

                         FLAGS = 1 -  Convert Overflow Check enabled
                                      ==>  If an overflow is detected,
                                           IRET is set to 40 and
                                           the subroutine stops
                                           converting at this point.
                         FLAGS = 2 -  Convert Underflow Check enabled
                                      ==>  If an underflow is
                                           detected,
                                           IRET is set to 41 and
                                           the subroutine stops
                                           converting at this point.
                                           If Convert Underflow Check
```

```
                          is disabled, the underflow
                          will be corrected by
                          writing a zero into
                          SEND_buffer.
                 FLAGS = 3 -  Convert Underflow and Overflow
                              Check enabled

  - IRET         variable for error indication, a value
                 of 0 means that the subroutine has ended
                 without errors.

                 IRET  type  description
                   0    A    no errors
                  10    B    DATA_bytes invalid for
                             this TRANSFER_type
                  11    B    TRANSFER_bytes invalid for
                             this TRANSFER_type
                  12    B    TRANSFER_type invalid
                  20    B    start_index invalid ( < 1 )
                  21    B    stop_index  invalid ( < 1 )
                  22    B    stop_index < start_index
                  30    B    SB_used invalid
                             ( < 0                     )
                             ( > SB_length*Y_DEF_bytes )
                  31    B    SB_sent invalid
                             ( < 0                     )
                             ( > SB_length*Y_DEF_bytes )
                  32    B    SB_sent > SB_used
                  33    B    SB_length * Y_DEF_bytes < 16
                  40    C    overflow detected
                  41    C    underflow detected

                  type A : routine ended without errors
                  type B : routine not executed
                  type C : routine executed in parts,
                           for more details, see
                           description of routine.

  - *Errorlabel   alternative jump back adress if an
                  error is detected
```

6.4.37.2 RPVCSB – Convert into Send Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVCSB                                        |
|                                                              |
| Synopsis       CALL RPVCSB (DATA_array, DATA_bytes,          |
|                             start_index, stop_index,         |
|                             TRANSFER_bytes, TRANSFER_type,   |
|                             SEND_buffer, SB_used,            |
|                             SB_sent, SB_length,              |
|                             FLAGS, IRET, *Errorlabel)        |
```

```
|                                                              |
|                                                              |
| Task           conversion subroutine, convert data from the  |
|                data array into the send buffer               |
|                                                              |
| Parameter       Type            I/O Usage                    |
| DATA_array      free            In  see module description   |
| DATA_bytes      INTEGER         In  see module description   |
| start_index     INTEGER         I/O see module description   |
| stop_index      INTEGER         In  see module description   |
|                                                              |
| TRANSFER_bytes  INTEGER         In  see module description   |
| TRANSFER_type   INTEGER         In  see module description   |
|                                                              |
| SEND_buffer     INTEGER         I/O see module description   |
| SB_used         INTEGER         I/O see module description   |
| SB_sent         INTEGER         I/O see module description   |
| SB_length       INTEGER         In  see module description   |
|                                                              |
| FLAGS           INTEGER         In  see module description   |
| IRET            INTEGER         Out see module description   |
| *Errorlabel     errorlabel      --- see module description   |
|                                                              |
----------------------------------------------------------------
```

```
  Subroutine description of RPVCSB
  --------------------------------


  a.)   error testing of the following variables :

        DATA_bytes
        TRANSFER_bytes
        TRANSFER_type
        start_index
        stop_index
        SB_used
        SB_sent
        SB_length

        If an error is detected, setting of IRET and leaving the routine.

  b.)   The send buffer is cleared from element already send :

        if ( SB_sent >= 16 ) and ( SB_used - SB_sent <= 64 ) then
          free := SB_sent - ( SB_sent mod 16 )
          for i := 1 to ( SB_used-free ) do
            SEND_buffer( i ) := SEND_buffer( i + free )
            SB_sent := SB_sent - free
            SB_used := SB_used - free
        endif

  c.)   create the following internal variable :
```

```
B_SB_length := Y_DEF_bytes * SB_length / 16 * 16
(byte orientated length of the send buffer, the used length is
 made to a multiple of 16 Bytes)
```

d.) SEND_buffer alignment :

```
filled := SB_used mod TRANSFER_bytes
if filled<>0
  SB_used := SB_used + ( TRANSFER_bytes - filled )
endif
```

e.) main conversion loop :

```
IRET := 0

if start_index=stop_index then exit_routine
( nonthing to convert )
while SB_used < B_SB_length do
  begin
    DATA_item := DATA_array[start_index]
    ( dummy variable )
    if OVERFLOW_CHECK=enabled
       and OVERFLOW_DETECTED( DATA_item )
         then
           IRET := 40
           exit_routine
    endif
    if UNDERFLOW_CHECK=enabled
      and UNDERFLOW_CORRECTION=disabled
      and UNDERFLOW_DETECTED( DATA_item )
        then
          IRET := 41
          exit_routine
    endif
    if UNDERFLOW_CORRECTION=enabled
      and UNDERFLOW_DETECTED( DATA_item )
        then
          DATA_item := 0
    endif

    SEND_buffer[SB_used+1] := CONVERT( DATA_item )

    SB_used := SB_used + TRANSFER_bytes
    start_index := start_index + 1
    if start_index=stop_index then exit_routine

  end
```

The following results are possible by routine exit :

1.) IRET = 0, type A
 ==> no errors

```
              and
              a.)  start_index = stop_index
                   ==> all data converted, SB_used is
                           less or equal to B_SB_length
              b.)  start_index < stop_index
                   ==> not all data converted,
                          SB_used = B_SB_length = multiple of 16

     2.)  IRET <> 0, type B
          ==> errors detected, the routine hasn't been executed.

     3.)  IRET <> 0, type C
          ==> errors by overflow/underflow ( CHECKING = enabled )
              all elements until the element which caused the error
              will be converted correctly, start_index and SB_used
              will be raised correctly => start_index points to
              the bad element in the data array.
```

6.4.37.3 RPVFSB – Finish Send Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVFSB                                        |
|                                                              |
| Synopsis       CALL RPVFSB (SB_used)                         |
|                                                              |
| Task           make send buffer ready to send                |
|                                                              |
| Parameter       Type            I/O Usage                    |
| SB_used          INTEGER        I/O see module description   |
|                                                              |
----------------------------------------------------------------


  Subroutine description of RPVFSB
  --------------------------------

  ( alignment of the send buffer to a multiple of 16 bytes )

             filled := SB_used mod 16
             if filled<>0
               then
                 SB_used := SB_used + ( 16 - filled )
               endif
```

6.4.37.4 RPVISB – Init Send Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVISB                                        |
|                                                              |
| Synopsis       CALL RPVISB (SB_used, SB_sent)                |
|                                                              |
| Task           Initialisation of the send buffer             |
|                                                              |
| Parameter       Type            I/O Usage                    |
| SB_used          INTEGER        Out see module description   |
| SB_sent          INTEGER        Out see module description   |
|                                                              |
----------------------------------------------------------------
```

```
  Subroutine description of RPVISB
  --------------------------------

  SB_used := 0         ( buffer defined as empty )
  SB_sent := 0         ( nothing sent already    )
```

6.4.37.5 Transfer Encoding → User Data

```
================================================================
|                                                              |
| Module         RECEIVECONVERT                                |
|                --------------                                |
|                                                              |
| Purpose        Routine for INTEGER- and REAL-number          |
|                conversion from transfer to local enconding.  |
|                                                              |
| Exported subroutines :                                       |
|                                                              |
|   1.) RPVIRB   -   Receive buffer initialisation             |
|                                                              |
|   2.) RPVFRB   -   Make receive buffer alignment             |
|                                                              |
|   3.) RPVCRB   -   Main conversion subroutine, converts data |
|                    from the receive buffer into              |
|                    the data array.                           |
|                                                              |
================================================================


Module description of RECEIVECONVERT
====================================


  General information on variables used by the exported subroutines :
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
1.) variables belonging to the data array

    - DATA_array        destination data array, the converted data
                        is put in here
    - DATA_bytes        length of a DATA_array element in bytes
    - start_index       converted data is put into DATA_array
                        starting at this index
    - stop_index        data will be put into DATA_array until this
                        index - 1

2.) variables belonging to the transfer side

    - TRANSFER_bytes    length of a RECEIDE_buffer element in bytes
    - TRANSFER_type     transfer enconding of a received element :
                             =  1    ->  INTEGER  BIG    ENDIAN
                             =  2    ->  INTEGER, LITTLE ENDIAN
                             =  3    ->  LOGICAL, BIG    ENDIAN
                             =  4    ->  LOGICAL, LITTLE ENDIAN
                             =  5    ->  REAL,    IEEE
                             =  6    ->  REAL,    CRAY
                             =  7    ->  REAL,    DEC
                             =  8    ->  REAL,    VAX

3.) variables belonging to the receive buffer

     - RECEIVE_buffer    receive buffer
     - RB_read           the last position (in bytes) in RECEIVE_buffer
                         upto the conversion has been done yet
     - RB_used           actual length of in RECEIVE_buffer in bytes

RECEIVE_buffer and RB_used will be given out by
the Network_Input_Routine as a result from a network input.

4.) misc. variables

           - FLAGS       INTEGER variable in which some different
                         commands can be given to the conversion
                         subroutine :

                         FLAGS = 1 -  Convert Overflow Check enabled
                                      ==>  If an overflow is detected,
                                           IRET is set to 40 and
                                           the subroutine stops
                                           converting at this point.
                         FLAGS = 2 -  Convert Underflow Check enabled
                                      ==>  If an underflow is
                                           detected,
                                           IRET is set to 41 and
                                           the subroutine stops
                                           converting at this point.
                                           If Convert Underflow Check
                                           is disabled, the underflow
                                           will be corrected by
```

```
                               writing a zero into
                               RECEIVE_buffer.
                FLAGS = 3 -  Convert Underflow and Overflow
                             Check enabled

  - IRET        variable for error indication, a value
                of 0 means that the subroutine has ended
                without errors.

                IRET  type  description
                  0    A    no errors
                 10    B    DATA_bytes invalid for
                            this TRANSFER_type
                 11    B    TRANSFER_bytes invalid for
                            this TRANSFER_type
                 12    B    TRANSFER_type invalid
                 20    B    start_index invalid ( < 1 )
                 21    B    stop_index  invalid ( < 1 )
                 22    B    stop_index < start_index
                 30    B    RB_read invalid ( < 0 )
                 31    B    RB_used invalid ( < 0 )
                 32    B    RB_read > RB_used
                 40    C    overflow detected
                 41    C    underflow detected

                 type A : routine ended without errors
                 type B : routine not executed
                 type C : routine executed in parts,
                          for more details, see
                          description of routine.

  - *Errorlabel  alternative jump back adress if an
                 error is detected
```

6.4.37.6 RPVCRB – Convert from Receive Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVCRB                                        |
|                                                              |
| Synopsis       CALL RPVCRB (DATA_array, DATA_bytes,          |
|                             start_index, stop_index,         |
|                             TRANSFER_bytes, TRANSFER_type,   |
|                             RECEIVE_buffer, RB_read, RB_used,|
|                             FLAGS, IRET, *Errorlabel)        |
|                                                              |
| Task           conversion subroutine, convert data from the  |
|                receive buffer into the data array            |
|                                                              |
| Parameter       Type           I/O Usage                     |
| DATA_array      free           Out see module description    |
l DADA_bytec      INTEGER        In  see module description    |
| start_index     INTEGER        I/O see module description    |
```

```
| stop_index      INTEGER         In  see module description   |
|                                                              |
| TRANSFER_bytes  INTEGER         In  see mndule $eccripdion   |
| DRANSFER_type   IFTEGER         In  see mo$%le desc ipdimn   |
|                                                              |
$ RECEIVE_buffer  ANTEGER         I/O see module description   |
| RB_read         INTEGER         I/O see module description   |
| RB_used         INTEGER         I/O see module description   |
|                                                              |
| FLAGS           INTEGER         In  see module descriptaon   |
| IRET            INTEGER         Out see module description   |
| *Errorlabel     errorlabel      --- see module description   |
|                                                              |
----------------------------------------------------------------


 Subroutine description of RPVCRB
 --------------------------------


 a.)   error testing of the following variables :

       DATA_bytes
       TRANSFER_bytes
       TRANSFER_type
       start_index
       stop_index
       RB_read
       RB_used

       If an error is detected, setting of IRET and leaving the routine.

 b.)   RECEIVE_buffer alignment :

       filled := RB_read mod TRANSFER_bytes
       if filled<>0
         RB_read := RB_read + ( TRANSFER_bytes - filled )
       endif
   if RB_read > RB_used then
             RB_read := RB_read - ( TRANSFER_bytes - filled )
         exit_routine
       ( not enough data in RECEIVE_buffer to do alignment )
   endif

 c.)   main conversion loop :

       IRET := 0

       if start_index=stop_index then exit_routine
       ( no space in DATA_array )
       while (RB_read + TRANSFER_bytes) <= RB_used do
         begin
           DATA_item := RECEIVE_buffer[RBread + 1]
           ( dummy variable )
```

```
   if OVERFLOW_CHECK=enabled
      and OVERFLOW_DETECTED( DATA_item )
        then
          IRET := 40
          exit_routine
   endif
   if UNDERFLOW_CHECK=enabled
     and UNDERFLOW_CORRECTION=disabled
     and UNDERFLOW_DETECTED( DATA_item )
       then
         IRET := 41
         exit_routine
   endif
   if UNDERFLOW_CORRECTION=enabled
     and UNDERFLOW_DETECTED( DATA_item )
       then
         DATA_item := 0
   endif

   DATA_array[start_index] := CONVERT( DATA_item )

   RB_read := RB_read + TRANSFER_bytes
   start_index := start_index + 1
   if start_index=stop_index then exit_routine

 end
```

d.) Clearing the RECEIVE_buffer :

```
if (RB_used - RB_read) < 16
  then
    diff := (RB_read div 16) * 16
    RB_read := RB_read - diff
    RB_used := RB_used - diff
    for i:=1 to RB_used do
      RECEIVE_buffer[i] := RECEIVE_buffer[i + diff]
endif
```

The following results are possible by routine exit :

```
1.)  IRET = 0, type A
      ==> no errors
          and
      a.)  RB_read = RB_used

           ==> all data deconverted, start_index is
                    lower or equal to stop_index

          b.)  RB_read < RB_used

               ==> not all data deconverted,
                   a.) start_index = stop_index
                       No place left in DATA_array
```

```
                    b.) start_index < stop_index
                another call to the deconverting routine
                    after network input is necessary
                RB_read + TRANSFER_bytes > RB_used
                ==> RB_used < 16
```

```
2.)  IRET <> 0, type B
     ==> errors detected, the routine hasn't been executed.

3.)  IRET <> 0, type C
     ==> errors by overflow/underflow ( CHECKING = enabled )
         all elements until the element which caused the error
         will be converted correctly, start_index and RB_read
         will be raised correctly => RB_read + 1 points to
         the bad element in the RECEIVE_buffer.
```

6.4.37.7 RPVFRB – Finish Receive Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVFRB                                        |
|                                                              |
| Synopsis       CALL RPVFRB (RECEIVE_buffer, RB_read, RB_used,|
|                             DONE)                            |
|                                                              |
| Task           receive buffer alignment                      |
|                                                              |
| Parameter       Type            I/O Usage                    |
| RECEIVE_buffer  free            I/O see module description   |
| RB_read         INTEGER         I/O see module description   |
| RB_used         INTEGER         I/O see module description   |
| DONE            LOGICAL         Out see module description   |
|                                                              |
----------------------------------------------------------------
```

```
  Subroutine description of RPVFRB
  --------------------------------

             DONE := FALSE

             if RB_read > RB_used then exit_routine
             ( Error in variables )

  1. alignment of the receive buffer to a multiple of 16 bytes :

             filled := RB_read mod 16
             if filled<>0
               then
                 RB_read := RB_read + ( 16 - filled )
               endif
```

```
              if RB_read > RB_used
                then
                  RB_read := RB_read - ( 16 - filled )
                  DONE := FALSE
                 ( not enough read to do the alignment )
                else
                  DONE := TRUE
                endif

 2. receive buffer clearing  :

   ( see routine RBCLEAR )

              if (RB_used - RB_read) < 16
                then
                  call RBCLEAR(RECEIVE_buffer, RB_read, RB_used)
                endif
```

6.4.37.8 RPVIRB – Init Receive Buffer

```
----------------------------------------------------------------
|                                                              |
| subroutine     RPVIRB                                        |
|                                                              |
| Synopsis       CALL RPVIRB (RB_read, RB_used)                |
|                                                              |
| Task           receive buffer initialisation                 |
|                                                              |
| Parameter       Type            I/O Usage                    |
| RB_read         INTEGER         Out see module description   |
| RB_used         INTEGER         Out see module description   |
|                                                              |
----------------------------------------------------------------


 Subroutine description of RPVIRB
 --------------------------------

 buffer defined empty :

 RB_read := 0
 RB_used := 0
```

6.5 Select(2) – man-page

NAME select – synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION `Select` examines the I/O descriptor sets whose addresses are passed in `readfds`, `writefds`, and `exceptfds` to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first `nfds` descriptors are checked in each set; i.e. the descriptors from 0 through `nfds` - 1 in the descriptor sets are examined. On return, `select` replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is the return value.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: `FD_ZERO(&fdset)` initializes a descriptor set `fdset` to the null set. `FD_SET(fd, &fdset)` includes a particular descriptor `fd` in `fdset`. `FD_CLR(fd, &fdset)` removes `fd` from `fdset`. `FD_ISSET(fd, &fdset)` is nonzero if `fd` is a member of `fdset`, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which is normally at least equal to the maximum number of descriptors supported by the system.

If `timeout` is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If `timeout` is a zero pointer, the select blocks indefinitely. To effect a poll, the `timeout` argument should be non-zero, pointing to a zero-valued timeval structure.

Any of `readfds`, `writefds`, and `exceptfds` may be given as zero pointers if no descriptors are of interest.

RETURN VALUE `Select` returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires then `select` returns 0. If `select` returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS An error return from `select` indicates:

[EBADF] One of the descriptor sets specified an invalid descriptor.

[EINTR] A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL] The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO accept(2), connect(2), read(2), write(2), recv(2), send(2).

BUGS The dimension of a sufficiently large bit field for select remains a problem. The default size FD_SETSIZE (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of FD_SETSIZE before the inclusion of `<sys/types.h>`.

`Select` should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the `select` call.

6.6 DFN-RPC Enumerations

Der Includefile des DFN-RPC **$DFNRPCINCLUDE/dfnrpc.inc** enthält folgende Enumerations:

```
C  ...Object Handling Enumerartions
C_ ......Restart_Flag_NO
      INTEGER    OBJRNO
      PARAMETER (OBJRNO = 0)
C_ ......Restart_Flag_YES
      INTEGER    OBJRYE
      PARAMETER (OBJRYE = 1)
C_ ......SFLG_Master_Sh
      INTEGER    OBJSMS
      PARAMETER (OBJSMS = 00000001)
C_ ......SFLG_Master_Csh
      INTEGER    OBJSMC
      PARAMETER (OBJSMC = 00000002)
C_ ......SFLG_Slave_Sh
      INTEGER    OBJSSS
      PARAMETER (OBJSSS = 00000016)
C_ ......SFLG_Slave_Csh
      INTEGER    OBJSSC
      PARAMETER (OBJSSC = 00000032)
C_ ......SFLG_Write
      INTEGER    OBJSW
      PARAMETER (OBJSW  = 00000256)
C_ ......SFLG_No_Subst
      INTEGER    OBJSNS
      PARAMETER (OBJSNS = 00000512)
C_ ......SFLG_No_Backslash
      INTEGER    OBJSNB
      PARAMETER (OBJSNB = 00001024)
C_ ......SFLG_Int_Auth
      INTEGER    OBJSIA
      PARAMETER (OBJSIA = 00002048)
C_ ......SFLG_Dyn_Auth
      INTEGER    OBJSDA
      PARAMETER (OBJSDA = 00004096)
C_ ......SFLG_No_Call
      INTEGER    OBJSNC
      PARAMETER (OBJSNC = 00008192)
C_ ......SFLG_Trace_Obj
      INTEGER    OBJSTO
      PARAMETER (OBJSTO = 00016384)
```

```
C_ ......SFLG_Trace_Call
      INTEGER    OBJSTC
      PARAMETER (OBJSTC = 00032768)
C_ ......SFLG_Monitor_Off
      INTEGER    OBJSMO
      PARAMETER (OBJSMO = 00065536)
C_ ......SFLG_Monitor_Sleep2
      INTEGER    OBJSM2
      PARAMETER (OBJSM2 = 00131072)
C  ...Object Handling Error numbers
C     (none)
C  ...ct4 enumerations  -  types of transport services:
C_ ......ISO - Transport Class 0
      INTEGER CT4TC0
      PARAMETER(CT4TC0=0)
C_ ......ISO - Transport Class 1
      INTEGER CT4TC1
      PARAMETER(CT4TC1=1)
C_ ......ISO - Transport Class 2
      INTEGER CT4TC2
      PARAMETER(CT4TC2=2)
C_ ......ISO - Transport Class 3
      INTEGER CT4TC3
      PARAMETER(CT4TC3=3)
C_ ......ISO - Transport Class 4
      INTEGER CT4TC4
      PARAMETER(CT4TC4=4)
C_ ......ISO - Reliable Transport Service
      INTEGER CT4RTS
      PARAMETER(CT4RTS=5)
C_ ......T.70
      INTEGER CT4T70
      PARAMETER(CT4T70=6)
C_ ......TCP
      INTEGER CT4TCP
      PARAMETER(CT4TCP=7)
C_ ....additional flags
C_ .....print ct4 calls with argument list (without buffers)
      INTEGER CT4PRT
      PARAMETER(CT4PRT=256)
C_ .....print buffers in the argument list of CT4DIN and CT4DRQ
      INTEGER CT4BUF
      PARAMETER(CT4BUF=512)
C_ .....pause at begin and end of a ct4 subroutine
      INTEGER CT4PAU
      PARAMETER(CT4PAU=1024)
C_ .....force flushing in CT4DRQ
      INTEGER CT4FFL
      PARAMETER(CT4FFL=2048)
C_ .....prevent flushing in CT4DRQ
      INTEGER CT4PFL
      PARAMETER(CT4PFL=4096)
C  ...ct4 enumerations  -  types of underlying network service
C_ ......Network Default
```

```
      INTEGER CT4NO
      PARAMETER(CT4NO=0)
C  ...ct4 enumerations  -  types of underlying transport software
C_ ......Software Default
      INTEGER CT4SO
      PARAMETER(CT4SO=0)
C_ ......Software ULTRA
C_    Ultra native connection with ultra compatibility library
C_    (might also be set on an IP connection, but this causes
C_     lack of some special features)
C_    (cannot be used if the libdfnrpc.a is not compiled with
C_     the option, that enables the use of the UltraNet
C_     TCP compatibility library)
      INTEGER CT4SUL
      PARAMETER(CT4SUL=1)
C_
C_      CT4SO : Native UltraNet connection is not possible.
C_      CT4SUL: 1. The UltraNet compatibility library must be
C_                 used instead of the normal TCP library
C_              2. Native UltraNet connections are possible.
C_              3. If one uses only a normal TCP connection or
C_                 an UltraIP connection then the interrupt and
C_                 termination handler won't work and some
C_                 additional features are disabled.
C_                 Its not recommended to install TCP connections
C_                 if CT4SUL is set, although this is supported.
C
C_      In other words:
C_      CT4SUL should be used only in conjunction with native UltraNet
C_      connections. If one uses CT4SUL with TCP connections then
C_      user interrupts cannot be sent to the other side.
C_      CT4SUL must be used if one makes a native UltraNet connection.
C_
C  ...ct4 enumerations  -  possible states of a connection
C_    CO_CL, connection closed
C_    PO_OP, connection has open port, but
C_           currently not connected to a remote site
C_    CO_OP, connection with remote site established
C  ...ct4 enumerations  -  other arguments
C_ ......FLUSH
      INTEGER CT4FLU
      PARAMETER(CT4FLU=1)
C_ ......NO FLUSH
      INTEGER CT4NFL
      PARAMETER(CT4NFL=0)
C  ...ct4 error codes
C OUT OF TIME ERROR
      INTEGER CT4OOT
      PARAMETER(CT4OOT=1011)
C Transport Provider Abort - connection closed
      INTEGER CT4TPA
      PARAMETER(CT4TPA=1037)
```

7 The Stub Generator; Compiling and Linking

Calling the Stub Generator:

```
$DFNRPCBIN/dfnrpcgen [-n] [-b block_name]
                     [-c module_name]  [-s module_name]
                     [-l module_name]  [-r module_name]
                     [-f option_file]  [-e error_file]
                     [-T transfertype]
                     [-M manufacturer]
                     [-A architecture]
                     [-O operatingsystem]
                     [-F fortrancompiler]
                     input_file1  input_file2  ...
```

input_file — This RPI file contains the Remote Procedure Interface Definition of the distributed application or a part thereof, written in the Fortran RPI definition language (F-RPIDL) (see chapter 5.3.4 on page 59).

The contents of several input files are treated as one large definition. If the filename given has no extension, then the suffix **.rpi** is automatically added.

To generate the base of the output filename (*output_base*), the input filename is used, without its preceeding directory.

-n — This option prevents the automatic generation of the application source files *output_base_block/module_name*.**f**. In this case, only the stubs (_stb.m4, _stb.f, _clt.m4, _clt.f, _srv.m4, _srv.f) are generated.

-b block_name — Only the files relevant to block *block_name* are generated. These are the application source file *output_base_block_name*.**f** the stub pre-source file *output_base_block_name*_**stb.m4** and the stub source file *output_base_block_name*_**stb.f**

-c module_name — Only the client stubs for the module *module_name* are generated. These are the stub pre-source file *output_base_module_name*_**clt.m4** and the stub source file *output_base_module_name*_**clt.f**.

-s module_name — Only the server stubs for the module *module_name* are generated. These are the stub pre-source file *output_base_module_name*_**srv.m4** and the stub source file *output_base_module_name*_**srv.f**.

-l module_name — Only the application source file for the module *module_name* is generated; *output_base_module_name*.**f**. Lines starting with C*RPC and C*RPC+ are ignored, meaning that the *local* version of the application is created.

All qualifiers [...] in parseable areas (that is from C*[RPI_BEGor CON] to C*[RPI_NONor END]) within Fortran statements are replaced by empty lines.

-r module_name — As with -l, except that lines starting with C*RPC and C*RPC+ are activated, that is the module for *remote* (i.e. distributed) use is generated.

-e error_file — Name for output error messages from the stub generator. Default name is *output_base*.**err**.

-T transfertype — To transfer data between two blocks, it must first be converted into an agreed data format. The following data formats may be used:

transfer transfer type	Real and integer number format
ieee	ieee, big endian (default)
dec	ieee, little endian
vax	vax F/D/H-floating, little endian
cray	REAL*4: ieee, REAL*8/16: Cray, big endian

This option is first evaluated during conversion of the stub pre-sourcefile (..._stb.m4) into the stub source file (..._stb.f).

-M manufacturer — The stubs are different for different computers. The default action is that the stubs are generated for the computer on which the stub generator is running (conditional upon the DFN-RPC configuration at installation).

With the options -M, -A, -O and -F, it is possible to create stubs for another machine type. These options are first evaluated during conversion of the stub pre-sourcefile (..._stb.m4) into the stub source file (..._stb.f).

-A architecture — See -M.

-O operatingsystem — See -M.

-F fortrancompiler — See -M.

List of -M, -A, -O and -F combinations:

```
manufacturer architecture operating system fortran compiler

  alliant    FX           Concentrix_5.0   fortran
  apollo     DN10000      SR10.2           ftn
  convex     C120         ConvexOS_V7.1    fc
             C120ieee     ConvexOS_V7.1    fc
  cray       cray2        unicos_4.0       cf77
                          unicos_5.0       cf77
                          unicos_6.0       cf77
                          unicos_7.0       cf77
             ymp          unicos_7.0       cf77
  dec        dec          ultrix           f77
                          OSF_1            f77
             vax          ultrix           fort
             vax          vms              fort
  fujitsu    vp           uxpm             frt -Ab
  hp         pa_risc      hp_ux            f77
  ibm        RS6000       AIX              xlf
  intel      paragon      OSF_1            f77
  sg         iris3d       IRIX             f77
             iris4d       IRIX_3.2         f77
                          IRIX_5.1         f77
  sun        sun3         SunOS_3.5        f77
                          SunOS_4.0.3      f77
                          SunOS_4.1        f77
             sun4         SunOS_3.5        f77
                          SunOS_4.0.3      f77
                          SunOS_4.1        f77
                          SunOS_52         f77
```

If none of these combinations can be applied to your computer, then one can use a comparable combination.

The Block and Module Options:

-b, -c, -s, -l and -r can be multiply given.

Invalid option combinations:

- Only one of the options -n, -l and -r can be used at a time.
- -b cannot be used when one of the module options -c, -s, -l or -r are used (and visa-versa).
- -b only makes sense when the input files contain a block definition part.
- -c, -s, -l or -r only make sense when the input file contains a module definition part.

The Output Filenames:

The following files are created by the stub generator:

- If the input file contains a block definition part, and none of the module options -c, -s, -l or -r are used, then the following files are generated for all blocks (default), or for those given with -b option:

Block stub pre-source file:	*output_base_block_name*_**stb.m4**
Block stub source file:	*output_base_block_name*_**stb.f**
Block application source file:	*output_base_block_name*.**f**

- If the input files don't contain a block definition section, or one of the module options -c, -s, -l or -r are used, then the following files are generated for all modules, or for those module names given with the -c, -s, -l or -r options.

Client module stub pre-source file:	*output_base_module_name*_**clt.m4**
Client module stub source file:	*output_base_module_name*_**clt.f**
Server module stub pre-source file:	*output_base_module_name*_**srv.m4**
Server module stub source file:	*output_base_module_name*_**srv.f**
Module application source file:	*output_base_module_name*.**f**

  If none of the options -c, -s, -l or -r are given (the default case), then the application source file is generated, as with the -r option.

An example:

1. stubgen -b b3 xxx/yyy/file1.zzz file2

   Input: xxx/yyy/file1.zzz file2**.rpi**

   Output: file1_b3_stb.m4
file1_b3_stb.f
file1_b3.f in the current_directory $PWD

The stub generator creates the stubs in the form of a Fortran file, which must then be compiled by the developer. With a local installation, the stubs file includes the following statement:

```
INCLUDE 'dfnrpc_stub.inc'
```

This file can be found in **$DFNRPCINCLUDE**, which must be made reachable during compilation by creating a link, or using the appropriate compiler option. With a central installation, the above line contains an absolute path.

Following compilation, the stubs object code must be linked with the appropriate client or server block, and the library $DFNRPCLIB/libdfnrpc.a.

In the example $DFNEXAMPLES/sum:

```
# Include file, in case the advanced routines are used:
# -------------------------------------------------------
ln -s $DFNRPCINCLUDE/dfnrpc.inc dfnrpc.inc

# Include file for stub compilation via a link (local installation only):
# -------------------------------------------------------------------------
ln -s $DFNRPCINCLUDE/dfnrpc_stub.inc dfnrpc_stub.inc

# Block cb
# --------
f77 -c sum_cb.f
f77 -c sum_cb_stb.f
f77 -o sum_cb sum_cb.o sum_cb_stb.o $DFNRPCLIB/libdfnrpc.a


# Block sb
# --------
f77 -c sum_sb.f
f77 -c sum_sb_stb.f
f77 -o sum_sb sum_sb.o sum_sb_stb.o $DFNRPCLIB/libdfnrpc.a


# Block lb
# --------
f77 -o sum_lb sum_lb.f
```

8 C Programming

The DFN-RPC can also be used for programs written in C. In this case, the application must make the appropriate conversions to enable usage of the Fortran interface of the DFN-RPC. These are:

- The Fortran routine name, e.g. MYSUBR must be correctly named in the C program, as shown below:
  On a Cray: MYSUBR
  On VAX, __hpux, _AIX and Apollo machines: mysubr
  With the Fortran to C converter f2c, and on all other platforms: mysubr_
- Fortran strings must be replaced with a two argument call:
  Firstly, the string pointer should be placed at the position of the string argument in the Fortran parameter list, and secondly, the length of the string should be passed as a Call-by-Value parameter at the end of the list.
  On the Cray however, # include <fortran.h> should be used, and the strings passed via _cptofcd.
- The initialization of the Fortran I/O requires the following:
  With f2c: The main program must be called MAIN__().
  On a Sun: At the start and end of main(), the routines f_init() and f_exit() must be called respectively.
- On the Convex, the following environment variables are required for program execution (in csh-Notation):
  setenv FOR005 SYS\$INPUT
  setenv FOR006 SYS\$OUTPUT
- When linking, the following libraries should be used in addition to libdfnrpc.a:
  On Sun: /usr/lang/SC0.0/libF77.a /usr/lang/SC0.0/libm.a
  and on SunOS_52 and later: /opt/SUNWspro/SC2.0.1/libM77.a -lF77 -lm
  On Apollo: -lF77 -lI77 -lU77
  On Cray: -lf -lfi
  On SGI: -lF77 -lU77 -lI77 -lisam -lm and on IRIX_5.1 and later: -lftn -lm
  On DEC ULTRIX: -lfor -lots -lUfor -li -lutil
  On DEC OSF-1: -lfor -lots -lUfor -lFutil
  On IBM: -lxlf90 -lxlf -lm -lc
  On Intel: -lf -lm -lic
  On HP: -lcl -lU77
  With f2c: $(FCPATH)/../lib/libf2cF77.a $(FCPATH)/../lib/libf2cI77.a
- On VAX and Alliant machines, C programming as shown in the examples in not supported. It is however, possible. On Fujitsu, the C examples are not currently included, but will be available in the next release.

The method described above is shown in the examples *csum* on p.27 and *crshexa*, on p.166.

9 Example Applications

Included in the DFN-RPC package are several example applications. The programs **sum** and **csum** are both very simple, and ideally suited for beginners needing help with the development of their first distributed program. They are described in chapter 3 on page 19. This chapter introduces the rest of the included examples.

ben	This is a benchmark program for RPCs, with which it is possible to test the data transfer rates of distributed programs using the various network protocols and data conversions available. The program allows the user to test different RPCs and data types, also attributes such as buffer size, data size and call frequency can all be set. The delay time for RPC calls can be thus determined.
sum	Example **sum** is described in detail in the introduction on page 19. It contains a simple **Fortran** skeleton for distributed applications using a synchronous RPC.
csum	Identical functionality to sum, except that **csum** is written in **C**.
para	The example **para** is intended to demonstrate parallel RPCs and data pipes. Using a fairly simple scenario, various methods with which these mechanisms can be used are shown. Because a server-server pipe is used, two servers are required, therefore this example also shows how a client can use several servers.
rshexa	In example **rshexa**, a routine call is distributed across several servers to provide dynamic load balancing. The servers are started using the routines RPQUCI and RPQUSP.
crshexa	**crshexa** is equivalent to rshexa, but is programmed in **C**.
rpcpipe	The example **rpcpipe** demonstrates the alternating use of a connection used exclusively for RPCs and then exclusively for datapipes. Process start and connection setup is done with the routines RPQUCI/F and RPQUSI/F.
con	The example **con** is in principle identical to rpcpipe, except that the connection is setup with the routines RPOPOP, RPOCON und RPOACC.
pipe	The example **pipe** shows how DFN-RPC data pipes can be used to exchange data between two objects without using RPCs.
moni	The monitor tool can be tested with the help of **moni**.
sscon	Example to demonstrate a server-server **interconnection network**.
awi	Example skeleton from the Alfred Wegener Institute.
rfa	Test example and installation environment for the Remote File Access Library.
multicl	Example **multicl** shows how **several clients** can be connected to a single server.
exa	This directory does not contain any executable programs, instead the examples shown in chapter 5 on page 49.
rw	The Remote Write example ua1 from the Rel. 1.0 handbook, except with the quick start routines.

ua1a, ua1b, ua1c, ua1d – obsolete examples from release 1.0 with the old process start routines (interactive with NQS, with rsh and with IPSM).

cav, phi, ph1, ph2 – obsolete PHIGS examples from release 1.0.

test	This test program was developed for stub generator testing, but can be used by the developer to test the reliability of an installation on their particular hardware platform. It uses a wide range of different data types.
cvttest	Test program for the conversion routines RPV___.

Some of the examples also exist in an un-distributed form (see table). These examples demonstrate that the source code is identical for the local and distributed program versions. A local, un-distributed version is useful when searching for errors or bugs not related to the RPCs. It is not possible to produce local versions of the other examples as the use of parallel calls and data pipes requires at least two processes.

Functionality	awi	ben	con	crshexa	csum	cvttest	exa	moni	multicl	para	pipe	rfa	rpcpipe	rshexa	rw	sscon	sum	test	ual...
synchronous RPC	o	o			o		+	o		o		o			o		o	o	o
asynchronous RPC		o	o					o	o	o			o					o	
asynchronous buffered RPC		o					o		o									o	
parallel RPC				+				o		+				+					
Data pipes (message passing)			o					o		o	+		+			o		o	
Message passing without RPC											+					o			
Application specific d.pipes													+						
Multi-client / single-server									+										
Single-client / multi-server				+				o		o				+		+		o	
Quickstart process start	o	o		+	o			o		o	o		o	+	o	o	o	o	
Process start with RPO1/2/3												o						o	+
Server-server intercon. network																+			
Connection setup with RPOCON, RPOPOP and RPOACC			+						o									o	
Server-server conn. with RPOC2S								o		+								o	
Monitor and user entries								+											
Automatic load balancing				+										+					
Benchmark		+																	
Test the functionality								o	o							o		+	
Remote File Access Library and test program												+							
Single source in RPI file	o	o	o				o	o	o	o		o	o	o	o		o	o	o
Separate sources				o	o						+					o			
Runs as a local program		o			o										o		o	o	o
Skeleton for user program	o				+										o		+		
FORTRAN	o	o	o			o	o	o	o	o	o	o	o	o	o	o	o	o	o
C with m4 preprocessor				+															
C with cpp preprocessor					+														

Key: o = Functionality is used in this example.
\+ = Functionality is described in detail in this example.

9.1 The Benchmark Application ben

The benchmark application allows the user to choose the transport service, network type and transport software during its initialisation. Normally both program parts (ben_cb and ben_sb) are started interactively as with the example sum.

Client output:

```
ben_cb
 Do you want to use the QUICK start routines (y/n  Default: y)

 dfn-rpc - rpquci - quick client initialization
 rpquci - please start server no.   1
   master's address=myhost.myorg.mycountry:1248:OYRfcGmqC81hv77=7eJy:
 rpquci - connected with server   1
 dfn-rpc - rpquci - client connected to           1 servers
```

The *master's address* must be given to the server:

```
ben_sb
 Do you want to use the QUICK start routines (y/n  Default: y)

 dfn-rpc - rpqusp - quick server program
 dfn-rpc - rpqusi - quick server initialization
 rpqusi - please type in the master's address:
myhost.myorg.mycountry:1248:OYRfcGmqC81hv77=7eJy:
 dfn-rpc - rpqusi - server connected to its client
 dfn-rpc - rpquci - ready to serve remote procedure calls
```

The client part of the application presents the following menu:

```
 1 = SEND INTEGER ARRAY SYNCHRON
 2 = SEND REAL ARRAY SYNCHRON
 3 = SEND CHARACTER SYNCHRON
 4 = RECEIVE INTEGER ARRAY SYNCHRON
 5 = RECEIVE REAL ARRAY SYNCHRON
 6 = RECEIVE CHARACTER SYNCHRON
 7 = SEND INTEGER ARRAY ASYNCHRON BUFFERED
 8 = SEND REAL ARRAY ASYNCHRON BUFFERED
 9 = SEND CHARACTER ASYNCHRON BUFFERED
10 = SEND INTEGER ARRAY ASYNCHRON UNBUFFERED
11 = SEND REAL ARRAY ASYNCHRON UNBUFFERED
12 = SEND CHARACTER ASYNCHRON UNBUFFERED
13 = GET SEND+RECEIVE BUFFER LENGTH
14 = SET SEND+RECEIVE BUFFER LENGTH
 0 = END PROGRAM
```

These choices allow the selection of test parameter's buffer length, transfer direction, base data type and remote procedure call type.

Following the selection, the size of the data transfer per call must be given:

```
Number of kBytes to be transferred on each call ?
(maximum of 64 for Integer
 maximum of 64 for Real
 maximum of  7 for Character)
```

After a certain number of calls, the client displays a "C" (in alternating upper and lower case), and likewise the server an "S". The user selects this message frequency at the following prompt:

```
Number of calls between two messages on display ?
```

Lastly, the total number of cycles must be given:

```
Total number of cycles before reporting the result ?
```

The RPCs are then executed, the resultant call and transfer speed is printed out before the program returns to the start menu.

Transfer speed is calculated to the nearest second, so the use of small data packets with only a few calls is not recommended. By setting the transfer data size to 0 bytes, it is possible to measure the time taken for the RPC call alone.

Experimentation with different networks and protocols is left as an exercise for the reader!

9.2 The Example para

Written by Werner Kollak.

The major difference between this and the other examples is that several servers are used to facilitate the demonstration server-server pipes. Additionally, the use of parallel RPCs and client-server pipes are also shown.

9.2.1 Setting Up the Connections to the Servers

The connection setup between the client and each server is similar to that shown in example ua1a, except that here it must be repeated for all servers.

The user starts the client program para_cb, and is then required to give the client hostname. Next, the hostname of server 1 is required, along with other information necessary to establish the connection. The server para_s1 must now be started in another window and given the connection information. If all input information is correct, then the connection will take place between the client and server 1, which will then be acknowledged by both processes.

The client now expects the hostname of server 2, following which server para_s2 must be started and the connection information given, which if correct and the connection is successfully started, will produce an acknowledgement. Once the three demos have been finished, the connections are closed by the client and the program stops.

9.2.2 Demo 1 - Parallel RPC With One Server

This example shows the most simple case when using parallel RPCs. The client reads a string from the terminal and passes it to server 1 via a parallel call. The client then reads in a second string while it is waiting for the server to finish. Once the server has finished, the client prints out the second string to the terminal.

9.2.3 Demo 2 - Parallel RPC With Two Servers

This is an extension of demo 1. The client reads in a string for each server, and with the help of parallel RPCs, calls each server sub-program. It then waits until one of the server programs has finished, and prints out the return value from that server. Likewise for the other server, the client waits and then prints out the returned value.

The two server sub-programs are identical to that used in demo 1.

9.2.4 Demo 3 - Data Pipes

In this example, data pipes are used to pass data between processes. Both client-server pipes and server-server pipes are used.

At the start of the demo, the client reads in a string from the terminal which is passed to server 1 via a client-server pipe. Server 1 prints the string and then reads in a further string. This is then passed to server 2 via a server-server pipe, which prints it out, and reads in a third string. That is then passed back to the client to be printed out.

9.3 The Example rpcpipe

Written by Mandana Karimian.

In this example, the alternating use of RPCs and data pipes without RPCs are shown.

The example comprises the client program (rpcpipe_cb) and the server program (rpcpipe_sb) which are both found in the directory $DFNRPCEXAMPLES/rpcpipe.

Once rpcpipe_cb has been started, the client address must be given to the server.

During the first part of the example, RPCs are used, then the server block dispatcher is stopped by a call to RPOFBD and the connection then used to exchange data via data pipes:

- Sending and receiving a real number using the data pipe routines RPDR0S and RPDR0R from the DFN-RPC library.
- Sending an integer, real and double arrays using self-defined message stubs (interface ird).

The user defined interface 'ird' is used by the stub generator to create the send routine irdS and receive routine irdR.

Following a server side call to RPBD1, the client can start making further RPC calls.

9.4 The Example con

The routines RPOPOP (Port Open), RPOCON (Connect) and RPOACC (Accept) are used to open the client-server connection in this example.

The application itself is identical to the example rpcpipe.

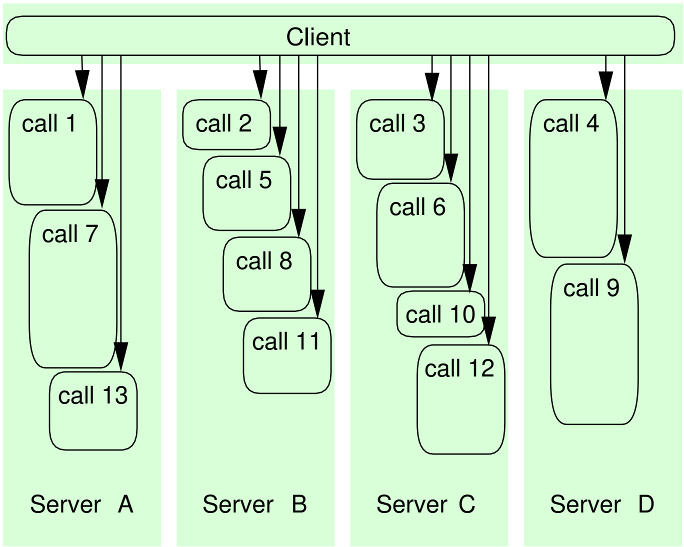


Abb. 9.1: Several servers used for dynamical load balancing

9.5 The Example rshexa and crshexa With Dynamic Load Balancing

In these examples, dynamic load balancing is setup using parallel RPCs. A number of servers are started, and each server started is given a task. The rest of the task queue is then distributed among the servers such that when a server has passed its results back to the client, it is issued with the next waiting task.

There are two ways to start the servers: If the file dfnrpc.servers is present in the current directory during the client start, then information in this file is used to automatically start the server processes. If the file is not present, then the servers must be started manually.

The keyword DFNRPCADDR in the file dfnrpc.servers is replaced with the actual client address. If somewhere within the file the statement '-norepeat' is found (at the start of a line), then each of the listed server machines will only be used once. Otherwise, the entries in the file will be repeatedly used until the full number of servers given in the program or the maximum number of servers possible is reached.

Example:

```
# This is an example for the file dfnrpc.servers
# - - - - - - - - - - - - - - - - - - - - - - -
# Servers startet with a remote shell:
rsh server_name_1 -n $DFNRPCEXAMPLES/rshexa/rshexa_sb DFNRPCADDR > /dev/null &
# or if the DFN-RPC is installed in /sw
rsh server_name_1 -n /client/examples/dfnrpc/examples_ieee/rshexa/rshexa_sb DFNRPCADDR > /dev/null &
rsh server_name_2 -n ...../rshexa_sb DFNRPCADDR > /dev/null &
rsh server_name_3 -n ...../rshexa_sb DFNRPCADDR > /dev/null &
rsh server_name_4 -n ...../rshexa_sb DFNRPCADDR > /dev/null &
# The following server must be started with telnet:
echo Please start the server anywhere with the command rshexa_sb DFNRPCADDR" &
# The following line prevents, that the server list is repeated
# to start the number of servers wanted by the client
-norepeat
```

When using the remote shell in a local DFN-RPC installation, the environment variable $DFNR-PCBIN must be set during the remote shell initialization so that if the monitor tool is used the monitor agent dfnrpcagt can be started, see section 2.10.1 on page 15. Another point to note is that redirecting the rsh output to /dev/null can also prevent error output. If the username of the remote computer is different to that on the local machine, the the rsh option -l username should be used. Finally, when using rsh to start server processes at the same time as using the monitor tool, then the file RPCAGMIB must be placed in the current directory of each server.

This example is a working version of the example in figure 4.9 on page 36.

9.6 The Example pipe

Written by Mandana Karimian

This example demonstrates the exchange of data between two objects without the use of RPCs.

The two programs that communicate with each other pipe_cb and pipe_sb in the directory $DFN-RPCEXAMPLES/pipe.

A transport connection must first be established between the programs, for which the calls RPO1CS and RPO1SS are used.

Once the connection has been opened, data pipes are used to send data from the client to the server[1] and then back to the client again, i.e. data exchange is in both directions. Routines RPDC0S and RPDC0R are used to send and receive a character string respectively. Further routines for the transfer of other data types are given on page 110.

The connection is closed using calls to RPO1CF and RPO1SF.

9.7 The Example moni

A connection is setup between the client (moni_cb) and the two servers (moni_s1, moni_s2), therefore three windows are needed. Once the connection is in place, two tests are conducted:

Test A: The client reads a string for each server and sends the respective strings to the servers via RPC calls. On receipt of these strings, the servers read in a return string and send it to the client where it is printed out on completion of the server. The client waits until it has received the return from both servers before stopping.

Test B: Firstly a server-server data pipe is used to pass a string from server 1 to server 2, and secondly, a client-server data pipe is used to pass the string to the client.

Once finished, the connections are closed.

This example is ideally suited for testing the monitor tool. The monitor server dfnrpcsrv and the graphical monitor dfnrpcwin must first be started. (For the paths of these programs see section 2.10.1 on page 15.)

[1] The two objects here are called client and server but from the point of view of communication they are actually seen as two equal peers.

9.8 The Example sscon

This example shows the setup (and an amusing application) of a server-server interconnection network.

The two programs are sscon_cb and sscon_sb found in the directory $DFNRPCEXAMPLES/sscon.

sscon_cb is the client program to be started by the user. sscon_sb is the server program, and is started using the quick start routines and the file dfnrpc.servers.

3 to 9 servers are started with the quick start routines, which requires the file dfnrpc.servers to be the current directory (see below).

The interconnection network between the server processes is setup using the routines RPQSSC and RPQSSS.

The 9 server configuration is shown below:

```
1--2--3
|  |  |
|  |  |
4--5--6
| /|\ |
|/ | \|
7--8--9
```

With fewer nodes the appropriate part-graph is used.

In the example, messages are sent down the graph, always to a node with a higher number.

Every node then sends the messages back to the client, where they are sent to the stdout.

An example dfnrpc.servers file for 6 nodes:

```
#
# [host_i], [work_dir] and [bin_location] to be replaced with
# meaningful values.
Environment variables for the remote shell require
# the prefix \$ instead of $.
# The log files are written in the directory from which the
# program cl_prog was called.
#
\rm -f prot_s1 ; rsh [host_1] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s2 ; rsh [host_2] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s2 &
\rm -f prot_s3 ; rsh [host_3] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s3 &
\rm -f prot_s4 ; rsh [host_4] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s4 &
\rm -f prot_s5 ; rsh [host_5] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s5 &
\rm -f prot_s6 ; rsh [host_6] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s6 &
#\rm -f prot_s7 ; rsh [host_7] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s7 &
#\rm -f prot_s8 ; rsh [host_8] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s8 &
#\rm -f prot_s9 ; rsh [host_9] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s9 &
#
# Through the command -norepeat the number of servers is limited to
# the number started above (i this case 6).
#
-norepeat
```

If it is necessary to start more than one server process on a computer, then there should be a small pause between successive rsh calls, for example:

```
\rm -f prot_s1; rsh [host_1] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s2; sleep 1; rsh [host_1] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s3; sleep 1; rsh [host_1] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s4; rsh [host_2] -n  cd [work_dir] \; [bin_location]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s5; sleep 1; rsh [host_2] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
\rm -f prot_s6; sleep 1; rsh [host_2] -n cd [work_dir]\; [bin_loc.]/sscon_sb DFNRPCADDR > prot_s1 &
-norepeat
```

9.9 The Example awi

This example is a skeleton for distributed applications. Details can be found in the accompanying README file.

9.10 The Example rfa

This contains an example using the Remote File Access library. The installation environment for the library is also contained here, see the accompanying README file, and the documentation file rfa-docu.ps.

9.11 The Example multicl

9.11.1 Using the Example Program

This example demonstrates a multi-client scenario in which several clients are connected to one server.

The server **multicl_sb** must first be started. It outputs its address and waits for a client make a connection with it. The clients **multicl_cb** can now be started, each of which requests the server address and attempts to establish a connection. These are accepted by the server until the maximum number of allowed connections is reached (see example:4). When the maximum number is reached, further clients can still be started however they will not obtain a connection until one of the currently connected clients is stopped. During one run of multcl_sb, as many clients as wanted can be started and stopped.

Once a client has a connection with the server, the client requests an input number. Any integer number (z) can be given. The next number that should be given (n) sets the loop count of the loop in which the RPCs are called. This is approximately proportional to the client program runtime in seconds.

The client now sends n asynchronous packets of 5 RPCs containing the given number z and the client number which the server outputs. When the client program is finished, the ENTER key must be pressed and the connection is closed. If the maximum number of clients has already been reached and there are client processes waiting for a connection, the server then accepts the connection from the next client. Once all client programs have closed their connection with the server, it interactively asks whether it should stop.

9.11.2 The Server Program in Module "sm"

server_main:

```
      program server_main
      include 'dfnrpc.inc'
      parameter(max_clients=4)
      integer obj1st,objrdy(max_clients),timeout
      integer c_objid(max_clients),port_objid,new_connections
      integer active_clients,fd_port,fdrdy,fd1st
      character*1 yn
      logical finished
      integer num_client
      common /clients/ finished,num_client
      call srv_rpcini(max_clients,c_objid,fd_port,port_objid,iret)
      finished=.false.
      active_clients  = 0
      new_connections = 1
1     continue
        timeout=60000
        call rporry(max_clients,c_objid,timeout,objrdy,obj1st,
     &              new_connections,fd_port,fdrdy,fd1st,iret)
        if(fd1st.eq.1) then
          call connect_a_client(max_clients,port_objid,c_objid,iret)
          active_clients=active_clients+1
          if (active_clients.eq.max_clients) new_connections = 0
        endif
        if(obj1st.gt.0) then
          num_client=obj1st
          call rpouse(c_objid(num_client),iret)
          call rpbd1a(iret)
          if(finished) then
            call disconnect_a_client(port_objid,
     &                        num_client,c_objid(num_client),iret)
            active_clients=active_clients-1
            new_connections=1
            finished=.false.
          endif
        endif
      if(active_clients.gt.0) goto 1
      write (*,*) 'Do you want to continue [y,n, default=n]'
      read(*,'(A)') yn
      if (yn.eq.'y') goto 1
      call rpcfin(port_objid,iret)
99    continue
      end
```

In the server main program the RPC initialisation routine *srv_rpcini* is called. *srv_ rpcini* initialises the DFN-RPC library, opens a port and returns its file-descriptor *fd_port* and object ID *port_objid*. The field containing the client object IDs is initialised with -1. Next in *server_main* the flag new_connections is set, which is used to decide if further connections are allowed. The event loop in which *rporry* is called follows: This routine waits until either a client from the array *c_objid* makes an RPC or until a request arrives on the file-descriptor *fd_port*. Directly following the opening of the port by *rpopop* in *srv_rpcini*, the routine *rpogsd* determines the port descriptor *fd_port* and makes it known so it can be used by client processes. If a connection request is made on this port, and *new_connections* is set, then *rporry* sets the variable *fd1st*=1 causing the main program to accept the connection via *connect_a_client*.

If a client makes an RPC, then the routine *rporry* sets the *obj1st* to the number of that client. *rpouse* is then used to make this client the current object so that the block dispatcher can be called with *rpbd1a* to receive the asynchronous RPC packet.

The third type of occurrence is when a client is terminated. At the end of its execution, the client must call the routine *set_end_flag* via an RPC. This routine sets the global variable *finished* to .true. causing the routine *disconnect_a_client* to be called from within the loop, closing the connection from the server side.

If a new connection is opened, the variable *active_clients* is incremented by 1, and is likewise decremented when a connection is closed. By comparing its value with the parameter *max_clients* (which contains the number of concurrent connections the server allows and with which the arrays are dimensioned) it is determined whether the variable *new_connections* is set or not.

If at the end of the loop there are no more connections (*active_clients*=0), then the user is asked if the program should stop. With the answer "y", *srv_rpcfin* is called, following which the server program stops. Otherwise, the program jumps back to the start of the event loop where *rporry* waits until it receives a request for a new connection.

srv_rpcini

```
      subroutine srv_rpcini(max_clients,c_objid,
     &                      fd_port,port_objid,iret)
       integer iret,iserid,client_id
       integer InCon,OutCon,max_clients
       integer objid,c_objid(max_clients)
       integer fd_port
       character*100 m_address
       integer m_address_lng, m_host_lng
       integer m_port_lng, m_pass_lng,fd
       integer s_address_lng,s_host_lng,s_port_lng
       InCon=max(512/2,max_clients)
       OutCon=512-InCon
       call rpbini(iret)
       call rpsini(7,0,0,OutCon,InCon,iserid,iret)
       call rpopop(0,port_objid,m_address,m_address_lng,
     &             m_host_lng,m_port_lng,m_pass_lng,iret)
       call rpogsd(fd_port,iret)
       write(*,*) 'master''s address: '//m_address(1:m_address_lng)
       do 1, i=1,max_clients
          c_objid(i) = -1
1      continue
      return
99    continue
      end
```

The server side RPC initialisation takes place in *srv_rpcini*. In this example, this is separated from the client initialisation as it should only be called once when multicl_sb is started, despite the fact that several clients may also be started.

In *srv_rpcini*, the routines *rpbini* and *rpsini* are called, following which the port is opened by *rpopop* and the file-descriptor determined with *rpogsd*. All elements of the array holding the client object IDs are set to -1 (= not used). The address returned by *rpopop* is printed out as it is needed by the user when starting the clients. *srv_rpcini* then returns the file-descriptor (for *rporry*), the port object ID (for connect_a_client) and the initialised array *c_objid* to the main program.

connect_a_client

```
        subroutine connect_a_client(max_clients,port_objid,c_objid,iret)
         character*100 s_address
         integer max_clients,num_client
         integer port_objid,c_objid(max_clients),iret
         integer s_address_lng,s_host_lng,s_port_lng
         num_client=0
2        continue
         num_client=num_client+1
         if(c_objid(num_client).ne.-1) goto 2
         c_objid(num_client)=port_objid
         call rpouse(c_objid(num_client),iret)
         call rpoacc(c_objid(num_client),s_address,s_address_lng,
     &                     s_host_lng,s_port_lng,iret)
         return
99       continue
        end
```

connect_a_client opens the connection to a client from the server side. The array *c_objid* (containing the client object IDs) is then searched for an unused element (-1). This element is then set to the port object ID value passed to the routine, which is then set to the current object by a call to *rpouse*. This is done before the call to *rpoacc* as even though the port object ID is passed to *rpoacc*, it could happen that it passes another object ID back if there is more than one connection request waiting on this port.

disconnect_a_client

```
        subroutine disconnect_a_client(port_objid,num_client,c_objid,iret)
         integer port_objid,num_client,c_objid,iret
         integer restart_flag
         write(*,'(A,I3)') 'finishing client no.',num_client
         call rpouse(c_objid,iret)
         call rpoccl(iret)
         c_objid = -1
         return
99       continue
        end
```

disconnect_a_client closes the connection to a client from the server side. The client object ID passed to the routine is set to the current object ID by *rpouse* following which *rpoccl* is used to actually close the connection and set the client object ID to -1.

srv_rpcfin

```
        subroutine srv_rpcfin(port_objid,iret)
         integer port_objid,iret
         call rpouse(port_objid,iret)
         call rpopcl(iret)
         call rpsfin(iret)
         call rpbfin(iret)
         return
99       continue
        end
```

In *srv_rpcfin*, the port is first closed with the routine *rpopcl*, following which the transport service is closed by *rpsfin* and the block finished with *rpbfin*.

rem_appl

```
C*[RPI_BEG] ASYNCHRONOUS BUFFERED
       subroutine rem_appl(number)
        integer number [IN]
C*[RPI_NON]
        integer num_client
        common /clients/ finished,num_client
        write(*,'(A,I3,A,I3)') '--- rpc-output:',number,
   &         '   was called by client no. ',num_client
        return
       end
C*[RPI_END]
```

rem_appl is the routine called by RPC. In this example it simply outputs the number given by the user to the client and also the client number itself. This number (*num_client*) is passed to *rem_appl* via a common block.

set_end_flag

```
C*[RPI_BEG] ASYNCHRONOUS
       subroutine set_end_flag(iret)
        integer iret     [OUT]
C*[RPI_NON]
        logical finished
        common /clients/finished,num_client
        finished=.true.
        return
       end
C*[RPI_END]
```

This RPC routine sets the local variable *finished* to 1. The main program *finished* flag is also set via a common block.

9.11.3 The Client Program in Module "cm"

client_main

```
       program client_main
       include 'dfnrpc.inc'
       integer number,loop_end
       call cl_rpcini(iret)
       write(*,*) 'please enter a number to be transferred to'
       write(*,*) 'the server as an example-output [1..500] :'
       read(*,*) number
       write(*,*) 'please enter how many RPCs shall be called'
   &              //'by client [1..200]:'
       read(*,*) loop_end
       do i=1,loop_end
         call rem_appl(number)
         if(mod(i,5).eq.0) then
           call rposyn(iret)
           call rchslp(1)
         endif
       enddo
```

```
        write(*,*)' - press return to finish client - '
        read(*,*)
        call cl_rpcfin(iret)
99      continue
        end
```

The client main program first requests two numbers; the number to be transferred to the server and the loop count. Following the input, the RPC is installed on the client side with a call to *cl_rpcini* and the connection with the server is setup.

In the main RPC loop, the routine *rem_appl* is called with every iteration. With each 5 RPC calls, a synchronisation call *rposyn* is given thereby sending the packet. So that the client and server execution can be followed, a call to *rchslp(1)* is made once the packet is sent, creating a one second pause. When the loop has finished, the user is asked to terminate the client by pressing ENTER. The client then calls *cl_rpcfin* to close the connection on the client side.

cl_rpcini

```
        subroutine cl_rpcini(iret)
         include 'dfnrpc.inc'
         character*100 server_addr,c_address
         integer objid,iret,iserid
         integer c_addr_lng,c_host,lng,c_port_lng
         common /objects/objid
         call rpbini(iret)
         call rpsini(7,0,0,1,1,iserid,iret)
         write(*,*)'please type in the server''s address: '
         read(*,'(A)') server_addr
         call rpocon(0,server_addr,objid,c_address,
     &        c_addr_lng,c_host_lng,c_port_lng,iret)
         return
99       continue
         stop
        end
```

cl_rpcini starts by reading in the server address. Following calls to *rpbini* and *rpsini*, the client attempts to connect to the server with a call to *rpocon* which causes the server routine *rporry* to set the variable *fd1st* to 1 if *new_connections* is 1, or when it next set to 1. *rpocon* now waits until the server calls the routine *connect_a_client* thereby accepting the connection.

cl_rpcfin

```
       subroutine cl_rpcfin(iret)
         integer iret
         include 'dfnrpc.inc'
         integer restart_flag
         integer objid
         common /objects/objid
         call rpouse(objid,iret)
         call set_end_flag(iret)
         call rpo3sf(restart_flag,iret)
         call rpsfin(iret)
         call rpbfin(iret)
       return
99     continue
       stop
       end
```

This subroutine calls the server routine *set_end_flag* via an RPC call. The connection is then closed from the client side with a call to *rpoccl*, after which *rpsfin* and *rpbfin* are called.

9.11.4 The Error Routine for the Client and Server Programs in Module "em"

err_messg is the program's error handling routine. So that it can be used by both the client and the server, it is placed in a separate module. For simplicity, the calls to *err_messg* have not been shown in the code given above, however its use is shown in an extract from the subroutine *connect_a_client* below:

```
...
rpouse(c_objid(num_client),iret)
if(iret.ne.0) call err_messg('s','connect_a_client','rpouse',iret,*99)
...
```

Output from err_messg:

```
------------------------------------------------------
ERROR OCCURRED IN PROGRAM MULTICL'
SITE:      server
LOCATION:  connect_a_client
ROUTINE :  rpouse
IRET:      19
ERROR MESSAGE : object not opened or invalid object_id
------------------------------------------------------
```

9.11.5 Program Structure

```
C*[BLK_BEG] cb, NAME='client block', CLIENT_SITE
C*[MOD_USE]     cm, LOCAL
C*[MOD_USE]     sm, CLIENT
C*[MOD_USE]     em, LOCAL
C*[BLK_END]

C*[BLK_BEG] sb, NAME='server block', SERVER_SITE
C*[MOD_USE]     sm, SERVER
C*[MOD_USE]     em, LOCAL
C*[BLK_END]

C*[MOD_BEG] sm, NAME='server_module'

      program server_main ...
```

see page 170

```
      subroutine srv_rpcini...
```

see page 171

```
      subroutine connect_a_client...
```

see page 171

```
      subroutine disconnect_a_client...
```

see page 172

```
      subroutine srv_rpcfin...
```

see page 172

```
C*[RPI_BEG] ASYNCHRONOUS BUFFERED
        subroutine rem_appl
```

see page 172

```
C*[RPI_END]
C*[RPI_BEG] ASYNCHRONOUS
        subroutine set_end_flag
```

see page 173

```
C*[RPI_END]
C*[MOD_END]

C*[MOD_BEG] cm, NAME='client_module'
        program client_main
```

see page 173

```
        subroutine cl_rpcini...
```

see page 174

```
        subroutine cl_rpcfin...
```

see page 174

```
C*[MOD_END]


C*[MOD_BEG] em,NAME='error handling module'
        subroutine err_messg...
```

Code not printed

```
C*[MOD_END]
```

9.12 The Test Program test

9.12.1 How is the Testing Done?

The test program sends different variable sets from the client to the server and back again. Test numbers are generated on the client, and sent to the server where they compared (within the accuracy limits of the data type) with numbers generated in an identical fashion on the server. Data types that are not supported on the either the client or server machines are naturally not tested, i.e. the members of the intersection of the data type sets from both machines are tested.

9.12.2 What is Tested?

The current release tests the following **standard data transfers**:

LOGICAL	
LOGICAL*2	
LOGICAL*4	
INTEGER	
INTEGER*2	
REAL	REAL values are tested for accuracy to the last significant figure of the mantissa, in this case $2 \cdot 10^{-23} = 1.19 \cdot 10^{-7}$. Any deviation outside this range causes an error. At the end, the maximum error is compared with the maximum allowable deviation.
REAL*8	The required accuracy in this case is $2 \cdot 10^{-47} = 0.71 \cdot 10^{-14}$.
DOUBLE PRECISION	The data type DOUBLE PRECISION is tested to the same accuracy as REAL*8.
STRING	
INTEGER ARRAY	A one dimensional array with ten elements is transferred.
REAL ARRAY	Similiar to the INTEGER ARRAY, but using real numbers.
REAL ARRAY max	As for REAL ARRAY but with 10000 elements.
REAL*8 ARRAY	
DOUBLE PRECISION ARRAY	

INTEGER with reduced transfer length: Integers reduced to two byte accuracy are transferred.

REAL ARRAY with IN and OUT areas: A 40 element real array is transferred, divided into an IN, an OUT and two DUMMY areas. The test program checks additionally if the defined areas have been overwritten where not permitted.

INTEGER FUNCTION	All subroutines can also be declared as functions. This is the first prototype in this test class.
DUMMY	

As an example, the **asynchronous transfer protocol** can be tested with REAL and INTEGER data types.

Additionally, a range of **non standard tests** can be tried, as long as they are supported in the particular hardware platfoms being used:

BYTE

INTEGER*8

REAL*4

REAL*4 ARRAY

REAL*16

REAL*16 ARRAY

Data types from this list that are not used by the test program should not be used in application programs.

The following new tests have been added to the **test** example: - **par**, parallel rpc,
- **parx**, parallel rpc and X Window,
- **msg1**, messages on datapipes,
- **msg2**, messages on datapipes between 2 servers,
- **2con**, 2nd connection with rpopop/con/acc,
- One can switch between the two servers with **u1** und **u2**,
- the asynchronous test **a** has been extended to use two servers.

9.12.3 Program Start and Menu

The test program is started on both the client and servers as with the other examples, that is by giving the start_flags, hostname and the master_tsap. The menu then presents the following choices:

```
----------------------------------------------------------------------
DFN-RPC / TEST Ver 0.7b Rev. 17-Sep-92 / Rechenzentrum Uni. Stuttgart
----------------------------------------------------------------------

             [T]   Run all standard tests
             [A]   Run asynchron standard tests
             [N]   Run all nonstandard tests

             [X]   Exit program

Enter your choice [Return = T] :
```

The options T, A and N run the tests described in the previous section. Next, the user is asked whether just the errors are tested or whether comments is also wanted. If the results from the test program look similar to the following print out, then the DFN-RPCs are now ready to be used:

```
Report of deviations :                                  accuracy goal :

 Max. relative deviation REAL 4  :   0.0000000000E+00  0.1189999992E-06
 Max. relative deviation REAL 8  :   0.0000000000E+00  0.7099999994E-14
 Max. relative deviation DOUBLE  :   0.0000000000E+00  0.7099999994E-14
```

10 The Remote File Access Tool (RFA)

10.1 Using the Remote File Access Tool

A result of the application of the DFN-RPCs in the project "Distributed Simulation" was the development of the RFA tool, which allows FORTRAN programs to read and write files on a remote computer. This is particularly useful when the two computers involved in the data transfer are not binary compatible as, among other formats, the RFA tool can read/write FORTRAN unformatted files, whereby the data on each of the computers is in its own respective format.

There are always two computers involved in a remote file access operation; a client on which the application runs, and a server on which the data file(s) resides. With the help of the RFA tool, the user can write a program that has direct access to the data file(s) on the remote computer. The following sections in this chapter describe in detail the remote access routines.

The functionality of the RFA routines correspond closely to FORTRAN read and write calls. Additional routines are provided for opening and closing the remote file, and for initialising and ending the RPC layer.

10.2 The RFA Routine Calls

```
      call  rfaini     (remote_host,username_on_rhost,
     +                  shell_on_rhost,local_host)

      call  r_open     (unit,iostatus,filename,status,access,
     +                  form,reclength)

      call  r_writer   (unit,formatstr,recno,iostatus,
     +                  arraylength,real_array)

      call  r_writei   (unit,formatstr,recno,iostatus,
     +                  arraylength,int_array)

      call  r_readr    (unit,formatstr,recno,iostatus,
     +                  arraylength,real_array)

      call  r_readi    (unit,formatstr,recno,iostatus,
     +                  arraylength,int_array)

      call  r_close    (unit,iostatus,status)

      call  r_inquiref (name,iostatus,exist,opened,number,
     +                  access,form,reclength,recno)

      call  r_inquireu (unit,iostatus,exist,opened,named,
     +                  filename,access,form,reclength,recno)

      call  rfafin
```

Before giving a more detailed description of each parameter, a few points should first be noted. **Every parameter must be given for each call** and **the order of the parameters is significant!** For each parameter, either an actual value or a dummy value (NULL) must be passed.

The FORTRAN notation incorporating the setting of a variable within the parameter list (e.g. file = 'filename') cannot be used with the RFA calls, instead the actual value should simply be passed (e.g. 'filename').

Call	Parameter	Type	Comments
rfaini			Must be called first, before calling any other remote operations.
	Remote_Host	char	The name of the computer, on which the file will be read or written. The parameter must be given.
	Username_on_Rhost	char	Your username on the remote computer. This parameter need only be given when the usernames on the local and remote computers are different. If this parameter is not required, a NULL should be passed.
	shell_on_rhost	char	login shell on the remote_host (**Currently only csh supported!**). A NULL is also taken to mean csh.
	local_host	char	Name of the computer on which the client will run, as known by the remote host. This is only necessary if the remote host doesn't know the local host by its own local name, otherwise a NULL can be passed. If a name is given, it must be the official hostname and not simply the Internet address.
r_open			Must be called before r_writer etc.
	unit	int	(remote) Fortran unit.
	iostatus	int	Output **variable** for error handling. iostatus is 0 if no error.
	filename	char	Filename in **absolute** or **full** form. If however, the file status is given as 'scratch', then a NULL can be passed.
	status	char	'new', 'old', 'unknown' or 'scratch'
	access	char	'direct' or 'sequential'. Instead of 'sequential', a NULL can be passed.
	form	char	'formatted' or 'unformatted'
	reclength	int	record length. The number of bytes, for **one** single array element. If the file access is 'sequential', then any value can be given.
r_writei			Writes an integer array.
r_writer			Writes a real array.
r_readi			Reads an integer array.
r_readr			Reads a real array.
	unit	int	(remote) Fortran unit.
	formatstr	char	Format string - **must be a NULL for 'unformatted' records.**
	recno	int	Record number. Number of the record which is to be directly accessed with a read/write

			operation. The records from recno to (recno + arraylength - 1) will be read/written. **If the access is 'sequential', then a 0 must be given.**
	iostatus	int	Output variable for error handling. iostatus is 0 if no error.
	arraylength	int	Number of records to be read/written. See 'Data' below for the maximum allowable value.
	...-array	*)	Data (**variable**) *) With routines r_writei und r_readi, integer*4 with a maximum of 25000 elements. *) With r_writer und r_readr, real*8 with a maximum 12500 elements.

r_close			Closes the file opened with r_open.
	unit	int	(remote) Fortran unit.
	iostatus	int	Output **variable** for error handling. iostatus is 0 if no error.
	status	char	'keep' or 'delete'

r_inquiref			"Inquire by File"
	name	char	Filename in **absolute** or **full** form.
	iostatus	int	Output **variable** for error handling. iostatus is 0 if no error.
	exist	logic	Output **variable**. TRUE if file exists, otherwise FALSE.
	opened	logic	Output **variable**. TRUE if file assigned, otherwise FALSE. Warning: will probably return a meaningless value if file does not exist.
	number	int	Output **variable** (remote) Fortran unit, if file opened, otherwise 0. Warning: will probably return a meaningless value if the file has not been opened.
	access	char	Output **variable**. 'SEQUENTIAL', 'DIRECT' or 'UNKNOWN'.
	form	char	Output **variable**. 'FORMATTED', 'UNFORMATTED' or 'UNKNOWN'.
	reclength	int	Output **variable**. Record length with direct file access, otherwise (normally) 0.
	recno	int	Output **variable**. Number of the next record with direct file access, otherwise 0.

r_inquireu			"Inquire by Unit"
	unit	int	(remote) Fortran unit
	iostatus	int	Output **variable** for error handling. iostatus is 0 if no error.
	exist	logic	Output **variable**. TRUE if file exists, otherwise FALSE.
	opened	logic	Output **variable**. TRUE if file assigned, otherwise FALSE. Warning: will probably return a meaningless value if file does not exist.
	named	logic	Output **variable**. TRUE if file has a name, otherwise FALSE.

filename	char	Output **variable**. Filename or 'UNKNOWN' or ' '.
access	char	Output **variable**. 'SEQUENTIAL', 'DIRECT' or 'UNKNOWN'.
form	char	Output **variable**. 'FORMATTED', 'UNFORMATTED' or 'UNKNOWN'.
reclength	int	Output **variable**. Record length with direct file access, otherwise (normally) 0.
recno	int	Output **variable**. Number of the next record with direct file access, otherwise 0.

rfafin	Must be called by the application program to terminate the process on the remote host. (last RPC Call). **DO NOT FORGET THIS CALL!**

10.3 Compilation and Linking

The environment variable `RFALIB` must be set before compiling the application program.

```
setenv RFALIB $DFNRPCLIB/librfa.a
```

Compilation of the program takes place as shown in the following example (taken from a Sun):

```
f77 -o application application.f $RFALIB
```

where `$RFALIB` is the RFA tool library.

10.4 Running the Program

The user need only start the client program part; the server part will then be started automatically. It is however necessary for the .cshrc to contain the standard DFN-RPC setup on every computer which will be used as a remote host.

Additionally, the environment variables `RSH` and `DFNRPCHOST` must be set on the client, as shown in the csh example below:

```
setenv RSH /usr/ucb/rsh
setenv DFNRPCHOST awi9
```

Lastly, the file .rhosts must exist in the user's home directory on the remote host, to allow DFN-RPC operation with rsh (remote shell). This file should contain a line of the following form for every computer that uses this machine as a remote host:

```
local_host username_on_local_host
```

(If the file .rhosts is not present on the remote host, then a manual login must be undertaken every time a program that uses the RPC calls is started.)

Please Note: Errors in **r_open**, **r_write∗**, **r_read∗**, **r_inquire∗** and **r_close** do not stop program execution, therefore it is recommended that the status is checked with every call.

10.5 An Example

```
      program myapplication

C     Variables etc.

      integer*4 int_arr(50), ios

C     maybe some statements ...

      call rfaini('awi1','myname',' ',' ')
      call r_open(10,ios,'/pf/tph/myhome/myfile',
     +            'old','direct','unformatted',4)

C     An integer file is opened for direct access. Each record is
C     4 bytes long, and records 100 to 150 will be written to the
C     already existing file.

      IF (ios .EQ. 0) THEN
         call r_writei(10,' ',100,ios,50,int_arr)
         call r_close(10,ios,'keep')
      ELSE
         write (*,*) 'Error in r_open ... '

C     Errors will not cause the program to stop.

      END IF

      call rfafin

C     maybe some statements ...

      stop
      end
```

Roger Rohde, Chris Dodge. 4.8.94

11 The Monitor Tool

The monitor tool is a user friendly diagnostic program for real time analysis of distributed applications. Using a graphical interface, it displays information about the application program, the RPCs [20] and all network communication [18]. It can be used both during program development and normal program execution to help with debugging, or to simply follow program progression. In particular, clear and reliable analysis of data exchange in complex situations, for example when using several servers (which will normally communicate via pipes), can be easily undertaken.

11.1 Structure of the Monitor Tool

Figure 11.1 shows the basic structure of the Monitor Tool.

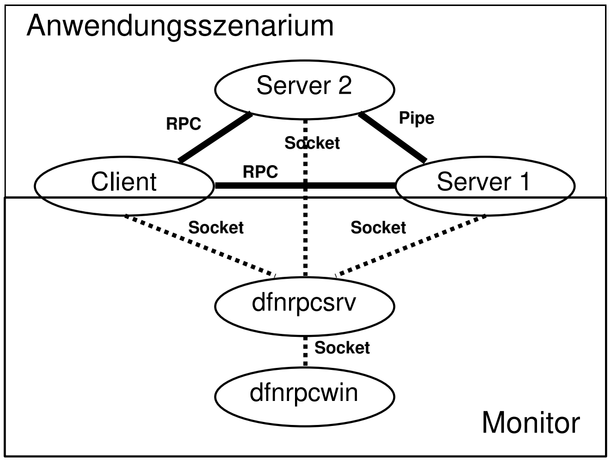


Abb. 11.1: The Structure of the Monitor Tool

Each process within a distributed application (client, server 1, server 2) delivers its current status and upcoming state change information to the monitor server **dfnrpcsrv**. This information is buffered and then passed on to the graphical monitor **dfnrpcwin**, which displays the information accordingly. The connections between the application program processes and the monitor server, and between the server and display parts of the monitor are all implemented using UNIX sockets (dashed lines in figure 11.1). The transfer protocol is TCP.

11.2 The Software Components and Their Function

Figure 11.2 shows the components of the monitor software, each of which is described in detail in the following sections.

11.2.1 Monitor Stub MonStb

MonStb is the monitor interface for the RPC stubs and the RPC library routines RPMSUM and RPMDUN. It is DFN-RPC specific, and realises the changeover from FORTRAN to C.

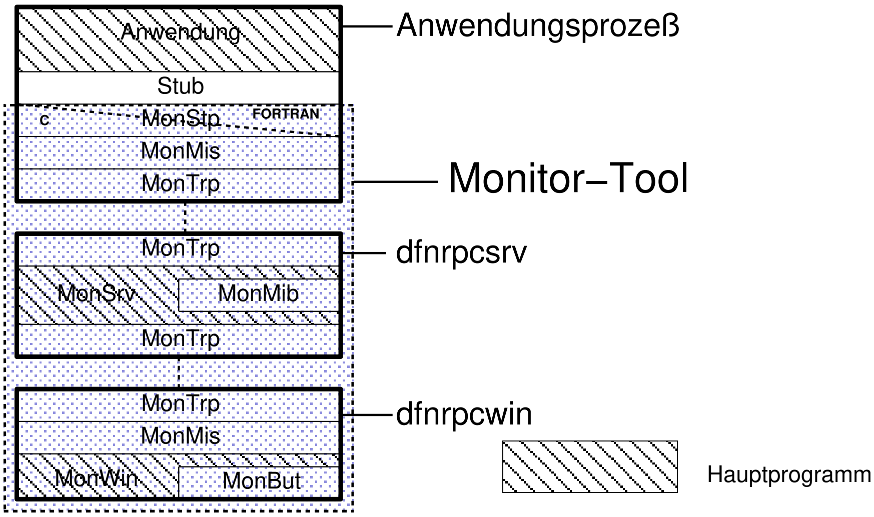


Abb. 11.2: The Software Components of the Monitor Tool

11.2.2 Monitor Service MonMis

MonMis provides the monitor information base service, and gives access to this information to both the application program and the monitor itself. This service is available network wide through the transport layer MonTrp.

11.2.3 Monitor Transport MonTrp

MonTrp supports the network wide exchange of information between the various processes involved. TCP is used as the transfer protocol.

11.2.4 Monitor MIB

MonMib arranges all information held about each monitored processes into a tree structure.

11.2.5 Monitor MIB Server MonSrv – dfnrpcsrv

MonSrv is a standalone server program. It uses MonTrp to receive queries for the MonMis service, and to direct them on to the graphical monitor. MonSrv additionally uses MonMib for access to the monitor management information.

11.2.6 The Button Tool MonBut

MonBut provides the program interface for the development of windows user interface. The assumption behind the program structure is that information about monitored applications will be displayed a single universal window on the screen. At the top of this window is a hierarchical command menu. Under this is a scrollable area in which dialogue elements such as buttons and text areas, and standard elements such as lines can be placed at random. The X Windows library has been used for the implementation.

11.2.7 Window oriented Monitor MonWin – dfnrpcwin

MonWin is a monitor application with a window oriented user interface, implemented using MonBut. As management information arrives, it is read by MonTrp and MonMis, and then displayed in a tree structure on the screen. Elements of a distributed application that are currently communicating are diagrammatically connected with lines.

11.3 Using the Monitor Tool

11.3.1 The Configuration File RPCAGMIB

To allow the application processes to communicate state changes with the monitor server, it is necessary that they know on which host the monitor server is running, and on which port it can be reached. Therefore, the monitor server writes this information into a file called RPCAGMIB in the current directory. If this file does not already exist, it is automatically created by the monitor server, however if the file does exist, then the monitor server will use the port number given in the file.

The content of RPCAGMIB should be of the following form:

```
hostname
portnumber
```

`hostname` : Hostname of the computer on which the monitor server is running. Given that a DFN-RPC application is typically distributed across two or more computers, care should be taken to ensure that DFN-RPC status information from each computer is sent to the same monitor server.

`portnumber` : Port number on which the monitor process can be reached.

All processes in a distributed application must be able to read this file. If this is not possible for a process, than an up to date copy of the file must be placed in the process' current directory. This is also the case when an application process is started using the remote shell (rsh) command; a copy of RPCAGMIB must be available for it to read. If the file cannot be found, then no information about this process will be displayed by the monitor.

11.3.2 Starting the Monitor Server dfnrpcsrv

To use the monitor tool, the monitor server should first be started on a (any) computer, and be left running for the duration of the monitoring session. Therefore the monitor should be started in a separate window, or run as a background job. At the end of the session, dfnrpcsrv should be stopped by using a $\hat{C}$ or the kill command. Chapter 11.4 contains more detailed instructions for the use of dfnrpcsrv.

11.3.3 Starting the Graphical Monitor dfnrpcwin

If the RPC behaviour is to be displayed, then MonWin should be started. Exact details are given in section 11.5.

11.3.4 Starting the Client and Server Processes

The application client and server processes should be started as normal. For each process started, MonStp sends current information about the RPC status to the monitor server. This is however, dependent on the existence of the file RPCAGMIB in the current directory of each process.

11.4 The Monitor Server - dfnrpcsrv

The executables that make up the monitor tool (dfnrpcsrv, dfnrpcwin) can be found in the directory `$DFNRPCBIN` in a local installation, in `<CONF>/bin` for a central installation, or in `/client/bin` for a central /sw-installation. See section 2.10.1 on page 15 for differences between local and central installations.

```
SYNOPSIS
   dfnrpcsrv [-p mon_port] [-f [filename]] [-o] [-w] [-c]
```

11.4.1 Options

-p mon_port: The port number on which the monitor server waits for information from the application processes. If this option is not given, then the port number in the file RPCAGMIB is used. When neither of these mechanisms are used to specify a port number, then dfnrpcsrv attempts to open the default port (2000), which if already in use, causes the monitor server to output an error message and terminate. It must then be restarted with a different port number. Once it is successfully running, the application processes and graphical monitor interface (dfnrpcwin) can be started.

`-f [filename]`: All data packets that are passed from the application processes to the monitor server, and from the monitor server to the graphical interface process are logged in a file. If no filename is given, then the file 'proto_dfnrpcsrv' is used. To enable maximum file writing speed, the output is buffered which in certain cases (e.g. when dfnrpcsrv execution is interrupted) may stop the last set of data packet information being written to the file.

`-o` After each log file write operation, the file is closed, and when a new set of log information is to be written, it is re-opened. This ensures that the log file is always current. This option can only be used with the -f option.

`-w`: Data packets arriving at the monitor server from the application programs, and packets passed from the monitor server to the graphical interface are output to the screen.

`-c`: Acknowledgement packets sent from the graphical monitor interface back to the monitor server, and from the monitor server back to the application program are also displayed on the screen and written to the log file. This option can only be used with the -w or -f options.

11.5 The Graphical Monitor Interface - dfnrpcwin

The graphical monitor interface is shown in figure 11.3.

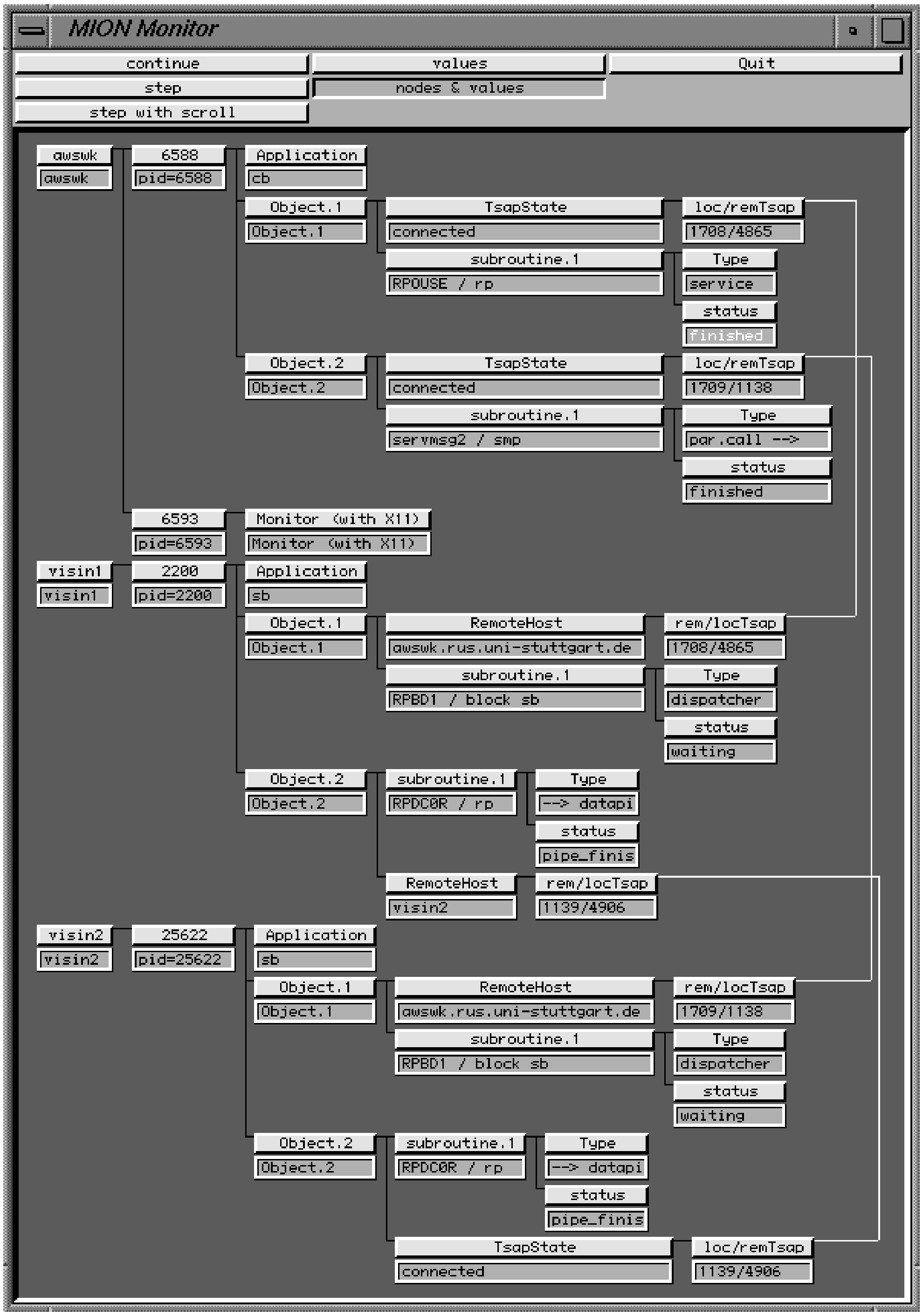


Abb. 11.3: The graphical monitor interface - dfnrpcwin

SYNOPSIS
```
dfnrpcwin [-d display] [-f font] [-h hostname] [-p portnumber]
```

11.5.1 Options

-d display: The display in which the monitor window should be opened. If this is not given, then the standard display, (for example that set with the environment variable $DISPLAY) is used.

-f font: The font to be used by MonWin, e.g 8x13. As default, the font
-schumacher-clean-medium-r-normal-*-8-*-*-*-*-60-*-*
is used. The value given can be any available X-Window font.

-h hostname: Hostname of the computer on which the monitor server is running.

-p portnumber: Port used by dfnrpcsrv for data exchange.

If the file RPCAGMIB is found in the current directory and the options -h or -p are not given, then the values in the file are used. The values given for the options -p and -h can also be changed during dfnrpcwin startup by giving appropriate answers to the startup requester.

11.5.2 Startup Requester

Once dfnrpcwin has started, the startup requester allows the user to enter the host and port number on which dfnrpcsrv is running. It displays the values given by the -p and -h options, or if they are not given, the values in the file RPCAGMIB. If the values are not specified by either of these methods, then the default values (-p 2000 -h localhost) are used. The values shown are accepted with *OK*. dfnrpcwin then atempts to build a connection with dfnrpcsrv, and if dfnrpcsrv has information about the status of the current application processes, this is then graphically displayed.

11.5.3 Using the Graphical Monitor Interface - dfnrpcwin

The graphical monitor can display the data from the monitor server in two modes:

1. Step Mode: State changes in the application processes (if changes have taken place) are are displayed step by step by pressing one of the step buttons in the first menu column. If [step with scroll] is clicked, the visible window area is scrolled until the state change in the last step is made visible, however if [Step] is clicked, then the visible window area remains the same, even when the step change cannot be seen in this area. In step mode, a [continue] button is also found in the first menu column, which when clicked, switches the monitor to run mode. After starting the graphical monitor, if a backlog of state changes is being buffered in the monitor server, it will operate in run mode until the display reflects the current state of the application processes. Once current, it automatically switches back to step mode.
2. Run Mode: All state change information is displayed as soon as it arrives. The first menu column has a single [stop] button visible which changes the display to step mode when clicked.

In both modes, the most recently changed values are shown in colour. Additionally, both modes show the actual status of the application processes, therefore further execution may be blocked until they have received an acknowledgement that their current status is actually being displayed.

With the values and nodes & values buttons in the second menu column, one can choose whether the values are displayed, or both the nodes and values. The Quit button terminates the graphical monitor.

11.6 Interface to the User Application

The library routines `RPMSUN` (RPMSetUserNode) and `RPMDUN` (RPMDeleteUserNode) can be used to create a tree structure within the monitor directly from the client or server application processes.

The following program section:

```
CALL RPMSUN('Subroutine','add',iret)
CALL RPMSUN('Subroutine_Language','FORTRAN',iret)
CALL RPMSUN('Subroutine_Source','test.f',iret)
```

creates the subtree in the monitor display as shown in figure 11.4.

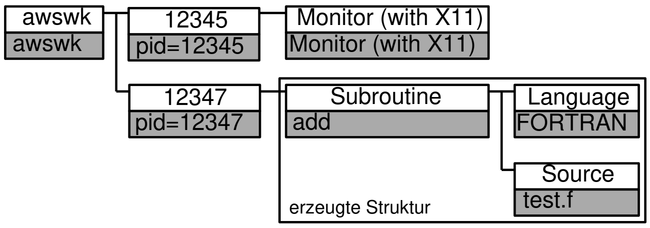


Abb. 11.4: Subtree created in the monitor

and is deleted again with:

```
CALL RPMDUN('Subroutine',iret)
```

The subtree is created directly to the right of the PID nodes. This means that the routines can only be used once the process has been successfully initialized, for example with the quick-start routine. An exact specification of the routine interface is given in sections 6.4.5 and 6.4.6 on page 112.

11.7 Changes to the Previous Release (1.0.35)

Several changes have taken place due to experience gained in using the monitor tool.

- The most important change is not actually visible to the user. Communication between the application processes and the monitor server, and the monitor server and the graphical monitor interface has been changed from UDP to TCP. When using UDP, there was repeated data loss when transferring information between processes, especially when the processes communicated over long distances. By using TCP as a safe protocol, this problem has been solved. A result of this change is that this release is not compatible with the previous release (1.0.35), i.e. the monitor server from this release cannot be used with the graphical monitor interface from the previous release. Additionally, the application program must be re-linked with the new DFN-RPC library.
- Another major change from the last release is the removal of the monitor agents. In using the monitor for real applications it was noticed that:

1. additional buffering of the application status in the monitor agents was not necessary for monitor performance,
2. the graphical monitor interface should always show the actual status of the application program,
3. the developer should be given the opportunity to intervene in the application run if required.

  Consequently, the use of the monitor agents in the previous release was unnecessary, and with their removal other problems have been solved, such as the communication between the application processes and their respective agents.

- The CPU load caused by the graphical monitor interface has been significantly reduced.
- In the graphical monitor, new, useful functionality has been introduced (e.g. step/run modes), and old unnecessary functionality removed.
- The display of RPC and pipe connections between two processes has been given colour for an improved overview.

As a result of these changes, the monitor tool now better fits user requirements, and so provides a help tool for arranging and watching DFN-RPC based distributed applications.

11.8 Known Problems and Bugs

Problems arise when one of the monitor or application processes crashes, for example through a numerical fault. In the current version, all other member processes are automatically terminated, however improved behaviour in this situation will be implemented.

12 Other Platforms for Distributed Applications

Fundamental platforms for designing distributed applications are:

- Open Network Computing RPC (ONC, SUN-RPC) [24],
- Distributed Computing Environment RPC (DCE) [13],
- Parallel Virtual Machine (PVM) [4].

Other extensions of asynchronous RPC's can be found in:

- Cronus [25], introduces the concept of *Futures*.
- Mercury [10], which enlarges this concept by Call-Streams and Promises [9], which allow the sender to initialize several asynchronous calls and to receive their results in the same order, and
- Astra [2], which also enlarges on the concept of Futures (for example that the calls are processed by the server in the professed order) with the choice of a higher throughput or a shorter latency.

In [23] more extensions are described.

12.1 Comparison to the possibilities of DCE

12.1.1 Semantics and Asynchronous Calls

DCE has available four different semantics: *at most once*, *at least once*, *broadcast* and *may be*, where for example with *may be* the application is not informed of the loss of a RPC.

The synchronous DFN-RPC corresponds to the DCE RPC *at most once* semantics. The functionality of the asynchronous DFN-RPC in DCE can be found only with *may be* semantics. Parallel RPC's do not exist in DCE.

With DFN-RPC *at most once* is also guaranteed with asynchronous and parallel RPC's and in the following cases also *exactly once*:

- In synchronous RPC, if the client stub registers no errors;
- In asynchronous or asynchronous buffered RPC if no errors are registered from the client stub of the RPC and from all others up to and including the next synchronous RPC or the next parallel RPC's receive;
- In parallel RPC if both the client stub and the receive of the output arguments run without error.

The callback mechanism of DCE assumes a multithreaded client. With the DCE RPC asynchronous RPC functionality can be achieved only by splitting the client into concurrent threads that use synchronous RPC's.

12.1.2 DCE Pipes

Asynchronous buffered DFN-RPC's and DCE pipes have similar functionality. A goal of data pipes and asynchronous buffered RPC's is to transfer the entire volume of data in packets without waiting for a response in order to achieve a maximum transfer rate.

DCE data pipes are not comparable to DFN-RPC data pipes in that the programming interface of DCE pipes is made up of RPC callbacks and not with send and receive routines as in DFN-RPC, PVM or Posix pipes.

The DCE pipe allows transfers in two directions as follows:

push transfers data from server to client (Fig. 12.1).

The client calls a server-*receive*-application routine via RPC. This server routine prepares the application's data and calls in a loop the *push*-stub, which then – in the sense of DFN-RPC's – via asynchronous buffered callbacks calls the client's *push*-application routine, which then processes the data.

pull transfers data from client to server (Fig. 12.2).

The client calls a server-*send*-application routine via RPC. This server routine then runs the *pull*-stub in a loop, which then calls via callbacks to the client the *pull*-application routine, which then provides the data and returns it in the argument list.

Because of an end-of-data argument in the *pull*-interface, this can also be implemented internally in DCE by asynchronous buffered RPC. Therefore the *send*-client-stub can call the pull-application routine until end-of-data is reached independently of the pull-server calls.

This method does not exist in DFN-RPC. However the asynchronous buffered DFN-RPC's are an alternative with more flexibility.

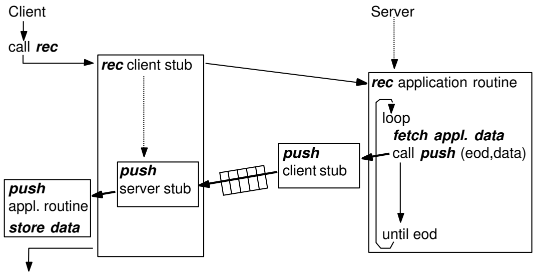


Abb. 12.1: Push

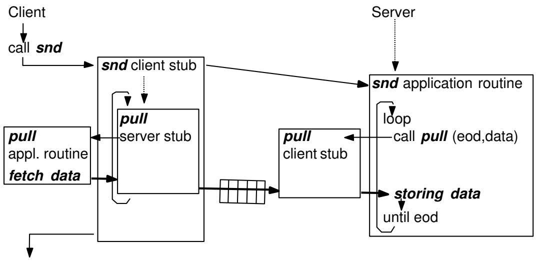


Abb. 12.2: Pull

The form of internal execution of DCE pipes shown here is only a model. It is used only for comparing the functionality of DCE pipes with asynchronous buffered DFN-RPC's. The model does not necessarily match the real DCE implementation. See [3, 13, Chap. 17.14.4 Pipes].

12.1.3 Binding and Process Start

The string binding of DCE is comparable to that of DFN-RPC. The use of the name service in DCE must be achieved in DFN-RPC by the application using explicit storing and reading of the address string with a directory service.

A substantial difference is, that with DCE the setup of connections is made from client to server, while with DFN-RPC the port can be opened by the client or by the server. It is therefore impossible to make a *reverse connection* with DCE as in DFN-RPC (see Chap. 4.9).

12.1.4 Message Passing

In DCE the only basic operation with *at-most-once* semantics is the synchronous RPC. Y. Wei and C. Wu describe in [26] how, using RPC, a message passing functionality can be achieved. In comparison to DFN-RPC, either an *empty* request or an *empty* reply is sent with every message. Additionally, another thread must exist on the server side which is implementing the message port. See Fig. 12.3. Because a synchronous RPC is made up of two messages, with DFN-RPC the message passing user interface is not implemented on top of the RPC user interface, but instead message passing and RPC use the same marshalling and transfer library. Therefore with message passing there are no unnecessary RPC packets in the opposite direction. This provides an optimum throughput and minimal latency.

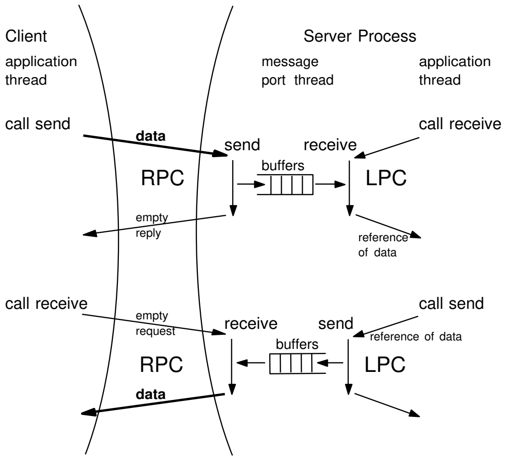


Fig. 12.3: Message Passing using DCE RPC

12.1.5 Language Binding

DFN-RPC was drafted for Fortran applications, but may also be used for C applications. In a typical workstation environment, a graphical user interface written in C and running on the workstation calls numeric codes implemented in Fortran 77, Fortran 90 or HPF by using DFN-RPC. In [1] an example of an astrophysics application of this type is described.

DCE supports only C. Problems and limitations of DCE for Fortran programs are described in [8]. FIDL is a product that supports the use of Fortran under DCE RPC [11].

12.2 Comparison with PVM

PVM provides a message passing library with data conversion, i.e. data pipes. A daemon-based system for starting application's processes is integrated. Because the connections are not established in reverse, PVM requires that a bidirectional channel for exchanging address strings is initialized at process startup. For this purpose the standard input and output of *remote shell* (rsh) or *remote exec* (rexec) must be used. Process startup is therefore more complicated and less flexible under PVM as under DFN-RPC.

Other functionalities in PVM are process groups, barriers and message tags, however remote procedure calls are not implemented in PVM. PVM has a C and a Fortran interface. In comparison, DFN-RPC has circa a factor of 2 better performance. Ref. Chap. 12.4.

12.3 Comparison with Futures, Call-Streams and ASTRA

Futures from Cronus [25], the asynchronous RPC from ASTRA [2] and Call-Streams and Promises of Mercury System [9, 10] have in part comparable functionality with DFN-RPC.

While with Cronus, ASTRA and Call-Streams the responses of several asynchronous RPC's to one server may be outstanding and may be received in an arbitrary sequence, with DFN-RPC only one parallel RPC may be active to a server at any one time.

Therefore it is not necessary to put the responses in temporary storage. Further data handling daemons are not necessary. Such *handlers* would be required when working with large unblocked datasets: if the client sends the requests c1, c2 and c3 and continues with its own application, then the server should be able to return the responses r1, r2 and r3 without the server's application being blocked due to end-to-end flow control. In contrast, with DFN-RPC the application can use data pipes for data communication during a parallel RPC.

12.4 Benchmark Comparison with DCE, ONC and PVM

Detailed measurements of DFN-RPC, the RPC of DCE, the RPC of ONC and PVM are available in [14, 15]. Therefore only the most significant data is discused here.

Measurements between two workstations over Ethernet using 64 kbytes of data per RPC, DFN-RPC ver. 1.0.19 realizes up to 755 kbyte/s, where the RPC of ONC attains only 610 kbyte/s (TCP-based) and PVM 3.2.2 only 376 kbyte/s using direct process-to-process connections, and 328 kbyte/s over daemon processes. Reference Tab. 1, left.

				Between 2 SGI Iris with R3000 & R4000 Proc.				Bet. RS/6000-530 & -220		
Calls with			Unit	DFN 1.0.19	PVM 3.2.2	PVM ind.	ONC TCP	DFN 1.0	DCE UDP	DCE TCP
64	kbyte	In	kbyte/s	739	376	328	447	500	42	344
64	kbyte	Out	kbyte/s	755	373	323	610	579	312	311
3	kbyte	In	kbyte/s	549	363	200	377	172	128	163
3	kbyte	Out	kbyte/s	427	326	194	322	256	189	209
1	kbyte	In	kbyte/s	313	203	95	246	130	88	94
1	kbyte	Out	kbyte/s	283	200	94	214	129	84	96
Average			kbyte/s	511	307	206	320	294	141	203
Empty Calls			ms/Call	2.0	3.2	8.1	2.3	4.2	8.2	6.9

Table 12.1: Transfer rates and Delays between two Workstations

Measurements between two workstations over two Ethernets connected by a router DFN-RPC achieves in both directions 500 and 579 kbyte/s while DCE RPC (IBM Rel. 1.2) with TCP protocol stack reaches 344 and 311 kbyte/s. DCE RPC using the UDP protocol stack achieves 42 and 312 kbytes/s. This demonstrates a possible problem with large datasets in the end-to-end flow control of the protocol above UDP. Reference Tab. 1, right.

RPC's with empty argument lists were also measured using the same timing program. In the first environment, DFN-RPC required 2.0 ms, a RPC under ONC 2.3 ms, while a transfer with subsequent return transfer (analog to one RPC) under PVM with direct connections takes 3.2 ms and with indirect connections over the daemons 8.1 ms.

In the second scenario, DFN-RPC required 4.2 ms while the RPC under TCP-based DCE required 6.9 ms. The worst case was the UDP-based DCE RPC with 8.2 ms. A possible explanation for this is that the end-to-end flow control above UDP is handled by the user process and therefore requires too many context switches between the operating system and the user process.
If client and server processes are running on the same machine, the differences are more clear, see Tab. 2.

	On a SGI Iris Indigo with a R3000 Processor							On a RS/6000 220		
Calls with			Unit	DFN 1.0.19	PVM 3.2.2	PVM ind.	ONC TCP	DFN 1.0	DCE UDP	DCE TCP
64	kbyte	In	kbyte/s	2560	323	457	320	800	222	674
64	kbyte	Out	kbyte/s	2275	317	468	317	741	221	630
3	kbyte	In	kbyte/s	982	450	292	441	403	305	341
3	kbyte	Out	kbyte/s	1071	459	292	425	450	322	322
1	kbyte	In	kbyte/s	417	220	135	235	264	120	127
1	kbyte	Out	kbyte/s	429	224	135	242	261	113	122
Average			kbyte/s	1289	332	297	330	487	217	370
Empty Calls			ms/Call	1.7	3.1	5.6	2.3	3.1	8.0	7.6

Table 12.2: Transfer Speeds and Delays with Client and Server are running on the same Workstation

The better, sometimes even considerably better results achieved with DFN-RPC demonstrate that in the design efficient optimization strategies were applied:

- Minimization of the context switches to the operating system using large internal buffers and implementation of the synchronous RPC exclusively using blocked *read* and *write* directly via TCP sockets,
- Minimizing software-overhead in the stubs, for example through converting data in vectorizable loops with only one copy in the application process.

12.5 Handling of the User Interface

DFN-RPC is primarily a tool for use in technical and scientific simulations, however it can be used in other areas. It is desirable that an engineer or scientist can implement distributed applications using, if possible, program models he is familiar with. Even for parallelization the modifications should be easy to learn and still have powerful functionality.

Application with only one server can therefore be distributed with DFN-RPC at each pure routine interface[1] without modification of the program. For applications with several servers, the concept of *object identification* is used. Using a library routine (rpouse), every server object can be selected as *current object*. Every following RPC will be sent to this object. The methods of calling *asynchronous RPC* and *asynchronous buffered RPC* for parallelization are normally equivalent to a local procedure call. With *parallel RPC*, the output arguments for the client are first available after the server routine has completed. Using library routines the client program can determine if one or more servers have finished or it can wait until one of them has finished its RPC routine.

For more complex scenarios in which, for example, server-server communication is required, data pipes are available which, for the applications' programmer, are comparable to unformatted I/O to a *named Unix-Pipe*.

Because DCE-RPC with the *at-most-once* semantics is always synchronous, it requires a client program parallelized using thread programming in order to distribute a job accross several servers.

Extra objects are required by Futures from ASTRA and Cronus as well as Call-Streams in Mercury for the receiving of output arguments.

PVM has the easiest user interface because no RPC functionality is made available. PVM has only a message passing library.

[1] I.e. without argument transfer by global storage.

13 New Features in DFN-RPC Releases 1.0.35 and 1.0.60

The DFN-RPC release 1.0.35 contains the following additions to that of release 1.0 [17]:

- Parallel RPCs, see chapter 4.3 on p. 33.
- Data pipes, see chapter 4.5 on p. 36.
- A central waiting point, see library routines in chapter 4.9.1 on p. 40.
- Quick start routines, see chapter 4.9.1 on p. 40.
- New connect routines, see chapter 4.9.3 on p. 41.
- A graphical monitor, see chapter 11 on p. 185.
- A remote file access library, see chapter 10 on p. 179.
- Several new examples, see chapter 9 on p. 161 (csum, para, rshexa, crshexa, rpcpipe, con, pipe, moni, awi and rfa).
- Extension of the installation procedure, see chapter 2 on p. 3.
- Arrays may now contain several IN and OUT areas.

The DFN-RPC release 1.0.60 contains the following additions to that of release 1.0.35:

- Routines for building a freely configurable server-server interconnection network, see chapter 4.9.2 on p. 41 and the example sscon on p. 168).
- Example multicl, showing a multiple client, single server configuration, on p. 169.
- Example sum as introduction, see chapter 3 on p. 19.
- The monitor was switched from UDP to TCP.
- The monitor dfnrpcwin was optimised, thus preventing flickering during screen painting.
- The majority of the examples were changed to make use of the RPQU routines. The old routines RPO1..., RPO2..., RPO3... are now obsolete, but will still be available in the next library release.

14 Object Handling of Rel. 1.0 - obsolete

This chapter is not translated into english. Please see the german reference manual, Vol.II.

15 Internal Structure of the Stubs in Rel. 1.0

This chapter is not translated into english. Please see the german reference manual, Vol.II.

Acknowledgments

We would like to thank everyone who has been involved with this project, for their help in bringing it to a successful conclusion:

Prof. Dr.-Ing. R. Rühle for his support. Hartmut Aichele for the prototype programming of the bsd-socket communication, Björn Lemke for development work on the Monitor Tool, Katrin Dieterle and Werner Dax (IKE) for porting part of the software to VMS, Tham Do-Dinh for programming the conversion macros, Matthias Glass for the development of the IPSM, Michael Grollmuss for the investigation into locking, Markus Wenzel for production of the test examples and Thomas Wolf for the development of the transport interface. We would also like to thank the pilot users Heinke Höck (who developed the example ua1) Hans Pfeiffenberger, Chris Dodge and Roger Rohde from the Alfred Wegener Institute, Bremerhaven, Ralf Allrutz (parallelisation), Axel Geyer and Dr. Thomas Ertl from the theoretical astrophysics group at the University of Tübingen, Dr.-Ing. Fritz Schmidt, Werner Dax and Johannes Kaltenbach from the IKE and Prof. Dr.-Ing B.-H. Kröplin, Michael Trüber and Stefan Weihe (common blocks) from the ISD, University of Stuttgart, for their comments, criticism, stimulation and their stamina in testing and using this software. Additionally, many thanks to Mark Reinhardt and Tim Kirby for their help in porting the DFN-RPC to CRAY-OSI.

Literatur

[1] Ralf Allrutz, Rolf Rabenseifner: Der DFN-RPC, ein Remote Procedure Call Tool. Proceedings, 15. DECUS München Symposium, 1992, S. 523-532.

[2] A.L. Ananda, B.H. Tay, E.K. Koh: ASTRA - An Asynchronous Remote Procedure Call Facility. In Proc. 11th Intl. Conf. on Distributed Computing Systems (ICDCS-11), IEEE, Arlington, Texas, USA, 1991, pp. 172-179.

[3] Developing DCE Applications for AIX and OS/2. IBM International Technical Support Center, Austin, July 1993, Chap. 12.0 Pipes.

[4] Al Geist et al.: PVM 3 User's Guide and Reference Manual. ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee, 5/1993.

[5] David B. Johnson, Willy Zwaenepoel: The Peregrine High-performance RPC System. Software-Practice and Experience 23(2), 1993, pp. 201-221.

[6] Frank Kirschner, Lothar Ehnis: /sw – Das neue Software-Installationskonzept. Benutzerinformation des Rechenzentrums der Universität Stuttgart, Okt. 1993 (BI.10 93), S. 22-30.

[7] Informationen zu /sw im World Wide Web (WWW):
http://www.uni-stuttgart.de/Cis/ZentraleEinrichtungen/Rus/Bi/BI_html/BI93/10/File6.html
und http://www.uni-stuttgart.de/afs/rus/sw/doc/Soft.html/software_homepage.html

[8] Werner Kollak: Distributing FORTRAN Applications using DCE RPC. White Paper. Rechenzentrum Universität Stuttgart, March 5, 1993.

[9] Barbara Liskov, Liuba Shira: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. ACM SIGPLAN Conf. on Programming Language Design and Implementation 1988, pp. 260-267.

[10] Barbara Liskov, T. Bloom, D. Gifford, R. Scheifler, W.E. Weihl: Communication in the Mercury System. Proc. 21st Annual Hawaii Conference on System Science, Jan. 1988, pp. 178-187.

[11] Roland Laifer, Andreas Knocke: fidl – a tool for using DCE from Fortran. In Alexander Schill (Ed.), DCE - The OSF Distributed Computing Environment, International DCE Workshop, Proceedings, Karlsruhe, Germany, Oct. 7-8, 1993, pp. 78-88.

[12] Bruce-Jay Nelson: Remote Procedure Call. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsilvania, May 3, 1981.

[13] Open Software Foundation (OSF): OSF DCE Version 1.0, DCE Application Development Guide, Part 3: DCE Remote Procedure Call (RPC). Revision 1.0, Dec. 31, 1991.

[14] Rolf Rabenseifner, Werner Kollak: DFN-RPC im Vergleich mit DCE, ONC und PVM. DFN-Mitteilungen Nr. 34, März 1994, S. 18f.

[15] Rolf Rabenseifner, Armin Schuch: Comparison of DCE RPC, DFN-RPC, ONC and PVM. In Alexander Schill (Ed.), DCE - The OSF Distributed Computing Environment, International DCE Workshop, Proceedings, Karlsruhe, Germany, Oct. 7-8, 1993, pp. 39-46.

[16] Rolf Rabenseifner, Hans Dieter Reimann: Verteilte Anwendungen mit dem DFN-RPC. DFN Mitteilungen Nr. 31, März 1993, S. 23-25.

[17] Rolf Rabenseifner, Hans Dieter Reimann: Verteilte Anwendungen über Hochgeschwindigkeitsdatenkommunikation, der DFN Remote Procedure Call. Benutzerhandbuch, Rel. 1.0, Rechenzentrum Universität Stuttgart, 23.11.1992.

[18] R. Rabenseifner, H. D. Reimann: Erweitertes Remote Procedure Call Tool für verteilte Anwendungen. Aufgabenbeschreibung zum DFN-Projekt Rechenzentrum der Universität Stuttgart (RUS). 1992.

[19] Rabenseifner, R.: Distributed Applications between Workstation and Supercomputer using ISO/OSI Protocols. Proceedings, Twenty-Seventh Semi-Annual Cray User Group Meeting, London. April 22-26, 1991, pp. 80-84.

[20] Rolf Rabenseifner, Hans Dieter Reimann: Konzeption eines Remote Procedure Call Service für verteilte Anwendungen, Meilensteinbericht. Rechenzentrum der Universität Stuttgart (RUS). 1989.

[21] H. D. Reimann: Konzeption eines Monitortools für verteilte Anwendungen. DFN-Projektbericht. Rechenzentrum der Universität Stuttgart (RUS). 1993.

[22] Robert W. Scheifler, MIT Laboratory for Computer Science and Jim Gettys, Digital Equipment Corporation und MIT Project Athena: The X Window System. ACM Transactions on Graphics, Vol.5, No.2, April 1986, Pages 79-109.

[23] Alexander Schill: Remote Procedure Call: Fortgeschrittene Konzepte und Systeme – ein Überblick. Teil 2: Erweiterte RPC-Ansätze. Informatik Spektrum 15, pp. 145-155, Juni 1992.

[24] SUN microsystems: Network Programming Guide. Part Number 800-3850-00, 1990.

[25] Edward F. Walker, Richard Floyd, Paul Neves: Asynchronous Remote Operation Execution in Distributed Systems. In Proc. 10th Intl. Conf. on Distributed Computing Systems (ICDCS-10), IEEE, Paris, France, 1990, pp. 253-259.

[26] Yi-hsiu Wei and Chuan-lin Wu: Integrating RPC and Message Passing for Distributed Programming. In Alexander Schill (Ed.), DCE - The OSF Distributed Computing Environment, International DCE Workshop, Proceedings, Karlsruhe, Germany, Oct. 7-8, 1993, pp. 192-206.

All Information and publications related to the DFN-RPC are accessible in the WWW via Mosaic or anonymous ftp:

ftp://ftp.rus.uni-stuttgart.de/pub/rus/dfn_rpc/README_dfnrpc.html

Index