Automation Programming

Automation Basics

Introduction to AU programming

An alternative way to macros for accessing ParaVision parameters and commands is possible with Automatic Routines (short Automations or AU’s). These are written in C-language, and are therefore more convenient than macros for applications involving more complicated maths. Automations are also advantageous when a rapid access to large array-parameters is required, and when advanced parameter operations such as re-dimensioning are necessary. Automations combine ParaVision high-level macros with the powerful flexibility and application diversity of the machine-oriented C programming language.

A unique feature of ParaVision is the possibility of inserting a user-programmed filter to the acquisition pipeline. Pipeline Filters can process the raw data on their way from the digitizer to the reconstruction. Pipeline Filters are special types of Automations.

This chapter should provide quick access to the development of ParaVision Automations, a framework to implement repetitive and automated batch command processing as well as just-in-time pre-processing of pipelined acquisition data. A complete AU programming reference is provided by the "AU Programs Reference Manual" available on the ParaVision 5.1 release DVD. This document can also be downloaded from the web site http://www.bruker-biospin.com/documentation_programming.html.

Purpose of Automations

Generally, ParaVision is applied interactively. That is, an operator executes ParaVision commands sequentially either by selecting them from GUI menus or by entering them from the keyboard. Usually, an operator waits for a ParaVision command to be finished before executing the next.
**Automation Programming**

*ParaVision* Automations (AU programs and pipeline filters) provide a mechanism of structuring commands by means of a source code text file written in the C programming language. In order to build AU executable Automations the code must be compiled by the operating systems C compiler and linked with a list of appropriate Standard and BRUKER C / C++ libraries.

Automations provide the following functionality:

- From the *ParaVision* point of view every AU executable is a *TOPSPIN* command. It can be called directly on the *TOPSPIN* command line or by *ParaVision* macros and scripts.

  *ParaVision* commands are usually executed by the function CPR_exec("<command>") or by high-level macros involving the CPR_exec() function call (see the reference list in section “"Index of AU program macros and commands” on page D-11-21). *ParaVision* defines some extensions to the *TOPSPIN* commands which are described in this chapter.

- Automations may use a high-level *PARX* macro interface in order to interact with the *ParaVision* parameter space (see "Parameter handling in Automations“ on page D-11-26).

- Automations may be embedded in a pipelined acquisition in order to perform just-in-time pre-processing of acquired datasets (see “AU filters for Pipelined Acquisition” on page D-11-37). These special AU programs are called AU filters. Their internal structure differs from those of standard AU programs.

- AU programs may run in the background. This is very useful for long-term data post-processing tasks.

- Automations may be used to control automatic adjustments during Traffic Light Acquisition.

- AU programs may be executed with command line parameters (introduced in *ParaVision 3.0*). Thereby, AU programs switch to the so-called Parametric Execution Mode (PEM). This mechanism is explained in section “Calling AU programs with parameters” on page D-11-8.

If not specified otherwise the following sections use the term "AU programs" for implicitly used AU programs, for standalone AU programs and AU pipeline filters.
For ParaVision and TOPSPIN a broad set of C macros is available. Some of them provide access to features of underlying library calls or high-level commands while others are used to hide standard C control structures for less experienced C programmers. It is possible to write AU programs without using any C constructs. On the other hand AU programs may entirely consist of C code. The best way would be to maintain a suitable balance between both approaches.

We recommend

- to use all categories of factory-made macros and commands whenever possible, especially for interactions with the Parameter Space (PARX),
- not to overwrite macros provided by BRUKER,
- to use plain C code for number crunching operations and disk I/O’s NOT related to the PARX interface (see section “Allocating memory” on page D-11-19).

Structure of AU programs

In order to embed AU programs into the ParaVision infrastructure, a programmer has to obey some programming conventions. In terms of C programming, an AU program is nothing else but the implementation of a C function called AU_program() or AU_proc() for AU filters, respectively. These functions are introduced in the

<PvInstDir>/exp/stan/nmr/au/vorspann*

file group.

Therefore, the following rules must be considered (see the following example):

- The programmer must follow the syntax rules of the C programming language
- Every AU program must have a code terminator called QUIT or QUIT-MSG("Message"). This closes the function body of AU_program(). AU filters use the code terminator PIPE_END.
- In order to access the parameter space an AU program must introduce the code body with the PARX_INIT macro. This macro is used to retrieve the parameters for the current dataset by reading out a dedicated PARX parameter object that has already been initialized and opened during the AU startup sequence (actually before entering the corresponding AU_program or AU_proc function).
• In order to access parameter sets without using the PARX interface the GETCURDATA macro may replace PARX_INIT at the top of the program code. Its counterpart redirects the current dataset and is called SETCURDATA.

• Local variables may be declared at the beginning of the AU program. The global identifiers name, name2, name3, type, type2, type3, disk, disk2, disk3, user, user2, user3, expno, expno2, expno3 and procno, prono2, procno3 describe the current dataset (obtained by PARX_INIT or GETCURDATA) and must not be declared.

• Functions can not be defined within the core of an AU program code since C precludes functions within functions. Function definitions must be placed after the code terminators QUIT, QUITMSG or PIPE_END using a forward function declaration at top of the AU program. Local AU program functions must be declared static.

According to many programming books here is a “Hello, world” example of a ParaVision AU program that obeys all the rules mentioned before. A corresponding AU filter example is shown in “Structure of AU filters” on page D-11-38. The source code should be regarded as a good starting point for user-defined AU programs:

```c
/*
   ************************************************************************
   AU program : hello
   ************************************************************************
*/

/*
   add includes and defines here
*/
#define MAXLEN 32

/*
   add local variables here
*/
char msg[MAXLEN] = "Hello, World";
char* ptr;

/*
   add function forward declarations here
*/
```
Building AU executables 11.1.3

For a comprehensive reference programmers refer to section 1.5 of the *AU Programs Reference Manual* which also can be downloaded from the URI [http://www.bruker-biospin.com/documentation_programming.html](http://www.bruker-biospin.com/documentation_programming.html).

To create and edit an AU program the operator has to execute the `edau <AUname>` command on the TOPSPIN command line. If `edau` is called without the argument, a list of selectable AU programs is displayed. The same happens when selecting the Edit/Compile/Execute AUs menu entry of the Spectrometer Control Tool. A convenient way of creating a new AU is to choose a unique file name for it. This prevents potential namespace problems during the compilation. A widespread method is to add prefixes or postfixes to the AU program name, e.g. `<AUname>_user` or `<project>_AUname`. Please omit the `.c` extension since, syntactically, an AU program is not a standalone C program but merely the implementation of a C function called `AU_program()` in which the programmers AU source code is embedded automatically. Anyway, the AU exe-
cutable builder would only support the file extension .au in AU program names. An AU is compiled into an executable by pressing the Compile button. AU executables are finally installed into the directory

\[ <\text{PvInstDir}/\text{prog/au/bin}. \]
Pressing the Execute button executes the selected AU. AU pipeline filters are activated in a different way. This will be explained in “AU filters for Pipelined Acquisition” on page D-11-37.

Readers interested in the internal structure of the compile environment for Automations may read the following indented paragraph. Otherwise skip to the next chapter.

Very valuable information about the internal structures of AU programs can be extracted from the files

\[ <\text{PvInstDir}/\text{exp/stan/nmr/au/vorspann} \text{ (standard AU)} \]
\[ <\text{PvInstDir}/\text{exp/stan/nmr/au/vorspann_p} \text{ (PARX related)} \]
\[ <\text{PvInstDir}/\text{exp/stan/nmr/au/vorspann_f} \text{ (PARX + Pipeline)} \]

Which one of the ‘vorspann’ files will actually be included depends on the list of macros used in an AU program. The workhorse of ParaVision AU compilation is divided into two PERL scripts whose main task is to collect the final list of libraries in order to generate the AU executable. The script

\[ <\text{PvInstDir}/\text{exp/stan/nmr/au/makeau} \]
collects the list of standard AU libraries. It also initiates the final compile and link process. The script

\[ <\text{PvInstDir}/\text{exp/stan/nmr/au/paravision_\text{<version>}.pl} \]
contains the appropriate library list for ParaVision related AU programs. It is called from within makeau. The paravision_\text{<version>}.pl file is considered as an overlay script for makeau which overwrites and / or extends the standard TOPSPIN library list by the list of ParaVision libraries and some subroutines. It allows the AU programmer to handle with ParaVision specific mechanisms such as PARX and Acquisition Pipeline but without being forced to give up the standard TOPSPIN AU programming interface.

**Debugging AU programs**

This section provides a step-by-step recipe in order to debug AU programs. A corresponding debugging guide for AU pipeline filters is explained in “Debugging AU filters” on page D-11-43.
The simplest way to trace the runtime behavior of an AU executable would be to insert `printf` statements at every source code line of interest. The following excerpt facilitates debugging source code by optionally switching debug messages on and off.

```c
#define DEBUG [0|1]  /* Choose 0 to switch off, 1 to switch on */
...
#if DEBUG
printf("value of parameter xyz = %d\n",(int)val);
#endif
```

It is obvious that such a debugging technique

- increases the complexity of the source code
- only shows the program state for a fraction of values at a certain time
- mixes relevant program outputs with sporadic debugging information
- alters AU runtime behavior (decreases performance) if `printf` statements are used excessively.

It is always more efficient to use a source code debugger during AU runtime execution. You can either use the internal command line interface of the GNU Debugger GDB. Alternatively, we strongly recommend to use DDD, the graphical frontend for the GDB. The DDD package is part of every Linux distribution and should exist with DDD version 3.3.1 or higher. Otherwise, refer to the web site of your Linux distributor in order to download / install the current supported RPM package. If not available refer to the original site http://www.gnu.org/software/ddd/ for the latest product version. RPM packages can be installed easily by executing the command

```
rpm -i <RPM_package_name>
```

Once installed the frontend debugging works as follows:

1. The AU program source code must reside in the directory `<PvInstDir>/exp/stan/nmr/au/src`. It must compile and link without errors.
2. Open an internal ParaVision console window by clicking the TERMINAL Icon in the toolbar of the System Control Tool
3. Change into the directory `<PvInstDir>/exp/stan/nmr/au/src`
4. Edit the AU program by the C command `sleep(60);` right above the PARX_INIT macro line at top of the AU code body.
5. Execute the command `../makeau -verbose -debug <AUname>` and finally quit the command line based debugger `gdb` with ‘q’. We will use a
graphical frontend for debugging called ddd. Now, a debug version of the AU program was generated and installed to <PvInstDir>/prog/mod/a.out.

6. Copy the AU program <PvInstDir>/prog/mod/a.out to the directory <PvInstDir>/prog/au/bin

7. Within the terminal window execute the command ddd <PvInstDir>/prog/au/bin/a.out. A debugger window opens. Attach the program a.out to the Debugger by selecting the menu entry File>Attach to Process. A list of potential processes will be offered to the user to attach. Select the highlighted one, i.e. the first entry of a group of a.out processes in this list. Wait some seconds until a CPR object has been created for this process (can be observed in the debugger terminal window at the bottom of the main window).

8. Execute the command pvxau a.out on the TOPSPIN command line. Load the AU program source (File>Load Source>AU name) and define a breakpoint inside the AU code, e.g. at the program code following the PARX_INIT macro. For this, point the mouse over the beginning of the desired code line and click the right mouse button in order to enable the breakpoint from the temporary pulldown menu. A red stop sign should appear at the beginning of the selected code line. Finally press the Continue button or enter the command 'c' into the debuggers command line window.

9. The debugger breakpoint is expected to be reached nearly instantly by displaying a GREEN arrow at the breakpoint position. The debugger is now ready to operate on the AU program.

10. Next: Step into the AU code!

**Calling AU programs with parameters** 11.1.5

An additional feature first introduced with ParaVision 3.0 is the implementation of a command line interface for ParaVision AU programs (not filters). It is called the Parametric Execution Mode (PEM). Flexible and dynamic long-term post-processing tasks with Automations can now be implemented much more easily. Such AU programs may be part of the ParaVision macro infrastructure which would offer a more powerful tool for automatic post-processing of acquired datasets than already possible with the current macro programming techniques. The following code example shows how to use the parameter interface from a programmers point of view (we modified the “Hello, World” example of section “Structure of AU programs” on page D-11-3):
/*
  **************************************************************************
  AU program: hello_dynamic [stringarg intarg]
  **************************************************************************
*/

/*
  add includes and defines here
*/
#define MAXLEN 32

/*
  add local variables here
*/
char msg[MAXLEN] = "Hello, World";
char* ptr;
int val = 0;

/*
  add function forward declarations here
*/
static void printout( char* );

PARX_INIT
/* write on console either default or command line parameter */
if (!use_dialogs) {
  strncpy(msg,(const char*)tlist.tarray[0],(size_t)MAXLEN);
  val = atoi((const char*)tlist.tarray[1]);
} else {
  /* a dialog may be available here or some default operations */
  GETSTRING("Enter axis to trim (x, y, or z): ",msg);
  val = 1;
}
ptr = &msg[0];
printout(ptr);
printf("val = %d\n",val);
QUITMSG("bye-bye")

/*
                        function section
*/
void printout( char* msg) {
    printf( "axis = %s\n",msg);
}

What is different to the introductory example hello?

1. The AU program can be either called with all or no optional arguments
2. A global variable use_dialogs is set to 0 if the AU program is called with optional parameters, otherwise it is set to 1 and the AU program executes the default program flow (e.g. a program dialog).
3. All optional parameters are stored in a list of tokens called tlist which has the following structure:

struct tokenlist
{
    int num;        /* number of tokens */
    char*  str;     /* untokenized command line parameter string */
    char** tarray;  /* pointer to array of tokens */
};
...

struct tokenlist tlist;
...

The optional parameter list must be a single string consisting of a list of parameter values separated by spaces.

4. A tokenizer will extract the options and fill the list with all command line parameters passed to the AU program. The num element holds information about the number of command line parameters actually passed to the AU program. Note, that it does not give any indication of the correct position index of a command line parameter. It is the programmers job to arrange the command line parameters in the correct order.
The following BRUKER standard AU programs use the PEM feature. They should be regarded as templates:

gen_lut, remove_zipper.

Application examples

The previous sections discussed the basics of AU programming. This section is dedicated to the practical aspects of AU development. Based on code excerpts the standard AU programming techniques will be explained in detail. In order to test them programmers may insert the code fragments into one of the listed program templates (see sections “Structure of AU programs” on page D-11-3 and “Calling AU programs with parameters” on page D-11-8) and compile the AU program.

Predefined variables and functions

The following global identifiers can be used throughout the entire AU program and must not be redefined.

```c
/* *******************************************************
/* G L O B A L   V A R I A B L E S */
/* ********************************************************/

int AUERR, /* error variable to be set by the macros */
    lastparflag = 1, /* for commands USELASTPARS, USECURPARS*/
    loopcount1, /* for TIMESn commands*/
    loopcount2,
    loopcount3,
    loopcountinf; /* for TIMESINFINITE cmd*/
char name[DATPATH_NAME_LEN] = "undefined", /* globals for RE, REXPNO etc. */
    type[DATPATH_TYPE_LEN] = "nmr",
    disk[DATPATH_DU_LEN] = "u",
    user[DATPATH_USER_LEN] = "undefined";
int expno=1,
    procn0=1;
char name2[DATPATH_NAME_LEN], /* globals for RE2, REXPNO2 etc. */
    type2[DATPATH_TYPE_LEN],
    disk2[DATPATH_DU_LEN],
    user2[DATPATH_USER_LEN];
int expno2,
```
procno2;
char name3[DATPATH_NAME_LEN], /* globals for RE3, REXPNO3 etc. */
type3[DATPATH_TYPE_LEN],
disk3[DATPATH_DU_LEN],
user3[DATPATH_USER_LEN];
int expno3,
procno3;
char parsettype[10] = "all";

/*--------------------------------------------------------------*
* list definitions...
*--------------------------------------------------------------*/
char namelist[10][DATPATH_NAME_LEN],
dulist[10][DATPATH_DU_LEN],
userlist[10][DATPATH_USER_LEN],
parsetlist[10][15],
pulproglist[10][15];
int expnolist[15],
procnolist[15],
loopcountlist[15];
float vtlist[15];

/*--------------------------------------------------------------*
* list counters...
*--------------------------------------------------------------*/
int xloopcount, /* current index in loop count list */
xpulprog, /* current index in pulprog list */
xparset, /* current index in parset list */
xdataset, /* current index in dataset list */
xvt, /* current index in vt list */
listcount1; /* running loop variable for 'timeslist' */

/*--------------------------------------------------------------*
* Miscellaneous variables...
*--------------------------------------------------------------*/
FILE *textfilepointer,
*debug = NULL; /* used in some functions as external variable */
char longpath[BUFSIZ/2]; /* for long path names */
char Hilfs_string[BUFSIZ/2]; /* used by some AU cmds in aucmd.h */
```c
int use_dialogs; /* interactive (1) or non-interactive (0) execution of AU */

/*--------------------------------------------------------------*
/* parameter space id for PARX...                                *
/*--------------------------------------------------------------*/
int PARX_psid = -1;

/***************************************************************
/*                LOCAL DEFINITIONS                          */
/***************************************************************
#define PATH_LENGTH PATH_MAX
int i1,i2,i3; /* define some local variables for each function */
float f1,f2,f3,f998,f999;
double d1,d2,d3;
char text[101];

Datasets without the PARX interface

Before an AU program can start any data acquisition or processing the dataset on which these commands should act must be defined with one of the following commands (how to handle datasets using the PARX interface is described in “Parameter handling in Automations” on page D-11-26). A complete list of C macros can be found in prog/include/aucmd.h.

- **GETCURDATA** - specifies the dataset which is currently active outside the AU program, and which is displayed on the screen. The dataset is described by the parameters disk, user, name, expno, procno.
- **RE(name)** resets the parameter NAME to the value of name.
- **REXPNO(n), TEXPNO, DEXPNO** resets, increments, decrements expno.
- **RPROCNO(n), IPROCNO, DPROCNO** resets, increments, decrements procno.
- The experienced C programmer may also intend to construct the various elements of the dataset description manually:
  
  procno+=100;
  expno = old_expno;
  sprintf(name,"%s.new",name);
```
• There are also the commands \texttt{GETCURDAT2}, \texttt{GETCURDAT3} which are necessary when working on two datasets simultaneously, either with \textit{TOPSPIN} commands such as \texttt{add}, or data algebra within the AU program itself.

It is often necessary to synchronize the local dataset (AU program scope) with the current one (\textit{ParaVision} scope). Setting the dataset is achieved with the command \texttt{SETCURDATA}. The status of the parameter space is controlled as follows:

• If the dataset is new, the parameters of the last dataset will be used.
• If the dataset already exists, then there are two possibilities: \texttt{USELASTPARS} and \texttt{USECURPARS}.

\texttt{USELASTPARS} causes the parameters to remain unchanged during the change of dataset, and the parameters stored on disk will be ignored. This is useful for overwriting old datasets.

\texttt{USECURPARS} causes the parameter space to be updated from the files stored on disk with the dataset. This is essential when processing previously acquired datasets. By default, the mode is set to \texttt{USECURPARS}. The mode remains effective until changed by a \texttt{USELASTPARS} or \texttt{USECURPARS} command, respectively.

\begin{verbatim}
/*
get information of current dataset, expno is increased by one and new dataset is
prepared for overwriting with parameters of current dataset. This is equivalent
to copying a dataset <expno> to a new or existing dataset <expno+1>
*/
GETCURDATA

... 
IEXPNO
USELASTPARS
SETCURDATA
\end{verbatim}

\section*{Platform independent programming \hfill 11.1.6.3}

With respect to platform independence (LINUX, WinNT) the AU programmer is recommended to use the following wrapper functions in order to obtain the path names of the most important \textit{ParaVision} subdirectories.

A complete list of wrapper functions can be found in the following files:

\begin{verbatim}
<PvInstDir>/prog/include/Path/SystemC.h and
<PvInstDir>/prog/include/Path/XWinNMRC.h
\end{verbatim}
/*retrieves the users home directory (LINUX: /home/<user>)*/
DLL_INTERFACE const char* PathSystemHome (void);

/*retrieves the TOPSPIN temporary directory (LINUX:/tmp ) */
DLL_INTERFACE const char* PathSystemTemp (void);

/*retrieves the path to the ParaVision installation directory */
DLL_INTERFACE const char* PathXWinNMRInst (void);

/*retrieves the program directory path (LINUX:<PvInstDir>/prog) */
DLL_INTERFACE const char* PathXWinNMRProg (void);

/*retrieves the path to the ParaVision application default directory
(LINUX: <PvInstDir>/prog/app-default) */
DLL_INTERFACE const char* PathXWinNMRApp (void);

/*retrieves the experiment directory path (LINUX: <PvInstDir>/exp) */
DLL_INTERFACE const char* PathXWinNMRExp (void);

/*retrieves the config directory path (LINUX: <PvInstDir>/conf) */
DLL_INTERFACE const char* PathXWinNMRConf (void);

/* retrieves the plot directory path (LINUX: <PvInstDir>/plot) */
DLL_INTERFACE const char* PathXWinNMRPlot (void);

/*retrieves the printer directory path (LINUX: <PvInstDir>/print) */
DLL_INTERFACE const char* PathXWinNMRPrint (void);

/* retrieves the path to the file .TOPSPIN-<host> in the user home directory */
DLL_INTERFACE const char* PathXWinNMRDotXWinNMR (void);

/*retrieves the current directory path */
DLL_INTERFACE const char* PathXWinNMRCurDir (void);

/*retrieves the diskless root directory path
(LINUX: <PvInstDir>/usr/diskless/clients/<spectname>/root) */
DLL_INTERFACE const char* PathXWinNMRDiskless (void);
/*retrieves the remote lock directory (LINUX: /usr/diskless/clients/spect/lock) */

dll_interface const char* pathxwinNMRRemLock (void);

Exception handling of an AU program 11.1.6.4

AU programs including the PARX_INIT macro may create error messages and can react to error conditions using exceptions. An exception may be thrown using the following call:

```
EXCEPT_printf(const char * format, ...);
```

This call has the same interface as printf (see man 3 printf). If the exception is not caught by the program (see below) it is printed as an error message and the Automation exits. Used in this way exceptions are a simple means of generating error messages in an Automation before aborting it.

**Example:**

```c
input_buf = (int *)malloc( sizeof_object );
if( input_buf == NULL )
    EXCEPT_printf("Cannot allocate % bytes for 2dseq frame\n",sizeof_object);
```

The user can catch an exception by implementing an exception handler. The exception handler is a special feature in Automations and can also be used in Method programming (see chapter “Method Programming” on page D-8-1):

```
EXCEPT_BEGIN
{
    ...
    if (error_found)
        EXCEPT_printf("This is an error message");
    ...
}
EXCEPT_HANDLE
EXCEPT_WHEN(EXCEPT_OTHERS)
{
    printf("The following exception was thrown: %s\n", EXCEPT_MSG);
}
EXCEPT_END;
```

If the Automation executes the `EXCEPT_printf` statement it jumps directly into the exception handler (beginning with `EXCEPT_HANDLE` and ending with `EXCEPT_END;`) and executes the code after `EXCEPT_WHEN(EXCEPT_OTHERS)`. `EXCEPT_MSG` only exists inside of an exception handler and contains the error
message string as formatted by \texttt{EXCEPT\_printf}. If the Automation does not execute the \texttt{EXCEPT\_printf} statement it will not execute the exception handler.

If an exception is thrown within a function without an exception handler the exception is propagated to the calling function. For example, in the case of a problem the parameter handling function \texttt{PARX\_has\_value} (see “Retrieving Parameter values” on page D-11-33) throws an exception which is not caught inside of this function. This exception is propagated to the caller and may be handled in its exception handler.

Exceptions are important since a lot of parameter handling routines in Automations and Method programming propagate possible errors using exceptions (see “Parameter handling in Automations” on page D-11-26).

An exception can also be re-raised / thrown in an exception handler using the \texttt{EXCEPT\_reraise()} call. This can be used if for example the user must close a file before exiting the program.

Automatic variables will be reverted to the value they had before \texttt{EXCEPT\_BEGIN} when the Automation begins to execute the code of the exception handler. This is not true for static variables.

The following code examples shows the use of \texttt{EXCEPT\_reraise} and tries to show the handling of automatic and static variables for exceptions:

```c
FILE * myfile = fopen(...); /* open a file */
int i = 10;
static int j = 10;
...
EXCEPT\_BEGIN
{
    fprintf(myfile, ...); /* write to the file */
    i = 5;
    j = 5;
    ...
    if (...)
        \texttt{EXCEPT\_printf("ERROR MESSAGE");} /* throw exception */
    ...
    if (PARX\_has\_value("MyParam"))
        /* throws an exception if MyParam does not exist. */
        * It is propagated and caught in the handler below. */
    ...
```
Automation Programming

EXCEPT_HANDLE

EXCEPT_WHEN(EXCEPT_OTHERS) /* catch exception */
{
    fclose(myfile); /* close file */
    printf("%d %d
", i, j); /* prints 10, 5 */ /*
    EXCEPT_reraise(); /* reraise exception, print "ERROR MESSAGE"
    * or message from PARX_has_value and exit */
}

EXCEPT_END;

Interactive AU programming 11.1.6.5

There are occasions when the program requires user input (see prog/include/aucmd.h).

Examples:
1. GETINT( "How many spectra do you want?", number ), where number must be declared as int.
2. GETSTRING( "What is the subject’s name?", name ), where name must be declared as char* or char[] (i.e. char name[16]).
3. GETDOUBLE( "Please enter SW:", sweepwidth ), where sweep width must be declared as double.
4. GETFLOAT( "Please enter LB:", linebroad ), where linebroad must be declared as float.

It is also possible to output messages to the screen using the Proc_err() function. This generates a window in which the specified information is printed. This function is defined as:

Proc_err( ErrorOptions, FormatString, VariableList );

ErrorOptions can take a range of values which are defined in the file

<PvInstDir>/prog/include/erropt.h.

Examples:

Proc_err( ERROR_OPT,"Sweepwidth=\%.2fHz",sweepwidth );

The modality of the message dialog depends on the value of the ackn variable that can be set with the setdef TOPSPIN command. If ‘setdef ackn ok’ is executed the window will remain active until it is closed by pressing the left mouse button or the Enter key. The execution of the AU program will pause during this time. If, however, ‘setdef ackn no’ is executed the AU program
will continue; the window will close automatically as soon as a new window needs to be opened. The opening of the window may or may not be accompanied by a beep depending on the state of the ‘setdef beep’ parameter.

**Example:**

```c
while (<condition>) {
    Proc_err(INFO_OPT, "Waiting for scan %d \n Only %d scans available\n", scan, scans);
}
```

This example is part of a polling loop which is waiting for a particular scan to complete. It provides the operator with the information that the required scan is not available and it is also regularly updated to inform the operator of how far the acquisition has progressed. It may be executed a large number of times and it would be inconvenient if the operator had to acknowledge the message every time before execution of the AU program proceeded.

A further use of `Proc_err()` is to provide the operator with the option to continue or cancel a particular action:

```c
short answer;
answer = Proc_err( QUESTION_OPT, "Do you wish to continue?\nPress OK for Yes or Cancel for No" );
```

In this example the window contains the text of the message, plus 2 boxes, OK and CANCEL. The AU program pauses until the operator clicks on one of the buttons. The variable, `answer`, will then get the value of either `ERR_OK` or `ERR_CANCEL`. A three-button window (ABORT/RETRY/IGNORE) is created with `ACTION_OPT`. The return values are `ERR_ABT`, `ERR_RETRY` or `ERR_IGNORE`.

### Allocating memory 11.1.6.6

Dynamic allocation is performed using the `malloc()` C function which allocates additional memory to the AU program. This memory is released by the operating system when the process terminates. It is a convenient programming practice to free the memory when it is no longer required.

```c
#include <malloc.h>
...
unsigned  sizeof_buf, sizeof_fid, total_data;
char *data_buf;
...
PARX_INIT
...
PARX_get_all_values(
    PARX_psid,
    "ACQ_dim",    &ACQ_dim,
    "NI",   &NI,
    "NR",   &NR,
    "ACQ_size",   ACQ_size,
    (char *)NULL );

...  
/*-----------------------------------------------*/
/* Allocate memory for input data */
/*-----------------------------------------------*/

sizeof_fid = ACQ_size[0];
total_data = sizeof_fid * sizeof(int);
sizeof_buf = (unsigned)(total_data * NR * NI);
data_buf = malloc( sizeof_buf );
if( data_buf == NULL )
    EXCEPT_printf("Cannot allocate % bytes for fid\n", sizeof_buf);

Private Shared Memory is often used to allow memory to be shared between processes.

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int  shmid;
char *input_buffer;
/* Calculate size as in previous example */
if( -1 == ( shmid = shmget( IPC_PRIVATE, (int)datasize, 0600 ) ) )
    EXCEPT_printf( "Couldn’t create shared memory ");
if( (char *)-1 == ( input_buffer = (char *)shmat( shmid, 0 0 ) ) )
    EXCEPT_printf("Couldn’t attach shared memory");
(void)shmctl( shmid, IPC_RMID, (struct shmid_ds *)0 );

Open file and reading data procedure is the same for both malloc’d memory and shared memory.

#include <fcntl.h>
...

int  filedes;
filedes = open( filename, O_RDONLY );
read( filedes, input_buffer, datasize );
close( filedes );
Automation Programming

Index of AU program macros and commands 11.1.7

This section is a reference list of ParaVision related AU-macros and AU-commands used in ParaVision Automations. For a comprehensive list refer to the "AU programs Reference Manual" available on the ParaVision 5.1 release CD. This document can also be downloaded from the registered web site http://www.bruker-biospin.com/documentation_programming.html.

We strongly recommend to use only the capital letter version of a AU program macro in order to avoid possible namespace errors or confusion with variable names. Some TOPSPIN related AU program macros are also available in variants using lower-case letters, they may work, but we strictly discourage you from using them!

A complete list of macros can be found in the subdirectory prog/include. The files are: aucmd.h, aupipe.h, autocmd.h, automo.h, autotyp.h.

Dataset Maintenance 11.1.7.1

GETCURDATA - The first AU program statement when NOT using the PARX mechanism; get the current foreground dataset.
SETCURDATA - Make the current AU dataset available for subsequent AU statements.
USELASTPARS - Overwrite old parameters
USECURPARS - Update parameters from disk
DATASET(....) - Set the current AU dataset
DATASET2(....) - Set the 2nd dataset (like the TOPSPIN command edc2)
DATASET3(....) - Set the 3rd dataset (like edc2)
GETCURDATA2 - Read the 2nd dataset (like edc2)
GETCURDATA3 - Read the 3rd dataset (like edc2)
DEXPNO - Decrease the experiment number by one
IEXPNO - Increase the experiment number by one
REXPNO(i1) - Set the experiment number to the value of i1
DPROCNO - Decrease the processing number by one
IPROCNO - Increase the processing number by one
RPROCNO(i1) - Set the processing number to the value of i1
DDATASETLIST - Decrement to the previous entry in the dataset list
**Automation Programming**

**IDATASETLIST** - Increment to the next entry in the dataset list

**RDATASETLIST**(i1) - Read the dataset at position i1 of the dataset list and make the current AU dataset

**IFEODATASETLIST** - Check if the end of the dataset list is reached. The answer is true if there is no further entry.

**SETDATASET** - Set the current AU dataset to the one currently defined by the dataset list

**DU**(dsk) - Set the disk unit to dsk

**SETUSER**(usr) - Set the user name to usr

**WRA**(i1) - Copy the raw data to the experiment number i1

**WRP**(i1) - Copy the processed data to the processing number i1

**WRPA**(....) - Copy the raw and processed data to the specified dataset

**VIEWDATA** - Show the current AU program dataset in TOPSPIN

**Synchronization macros**  

This set of macros is defined in order to synchronize the current process with the end of a command execution of another application process.

**SYNCH_WITH_PROCESS**(name) **Synchronize current process with application process name**

In order to help programmers define the correct application name, some wrapper macros have been introduced. They should be self-explanatory:

**SYNCH_WITH_PVCMD**

**SYNCH_WITH_PVSCAN**

**SYNCH_WITH_XTIP**

**SYNCH_WITH_PVCAM**

**SYNCH_WITH_PVDATMAN**

Another macro is used in order to synchronize the current process with the end of a pipelined acquisition performed on the currently defined dataset. It should be mentioned, that this macro cannot be used in AU filters but only in AU programs: **SYNCH_WITH_PIPELINE**.

---

D-11-22
Sending commands to applications and modules 11.1.7.3

This set of macros is defined in order to forward commands to ParaVision modules and applications by command queues.

FORWARD_CMD(name, cmd) - Forward the command cmd to application name.

In order to help programmers to define the correct application name some wrapper macros have been introduced. They should be self-explanatory:

PVCMD_CMD(cmd)
PVSCAN_CMD(cmd)
XTIP_CMD(cmd)
PVCAM_CMD(cmd)
PVDATMAN_CMD(cmd)

Handling datasets in the Scan Control Tool 11.1.7.4

This set of macros is defined in order to maintain the dataset list of the ParaVision Scan Control Tool:

SET_PVSCAN_CURDAT - Select the dataset defined by path, disk, user, type, name, expno and procno, i.e. selects and highlights the current dataset.

GET_PVSCAN_CURDAT - Retrieve path, disk, user, type, name, expno and procno of the currently selected and highlighted dataset.

CLONE_PVSCAN_CURDAT - Clone the currently selected and highlighted dataset.

Handling datasets in XTIP 11.1.7.5

This set of datasets is defined in order to maintain the dataset visualization of the Image Display and Processing Tool XTIP.

SET_XTIP_CURDAT - Load the dataset defined by path, disk, user, type, name, expno and procno into the active viewport.

GET_XTIP_CURDAT - Retrieve path, disk, user, type, name, expno and procno of the currently active viewport.

Acquisition and Reconstruction macros 11.1.7.6

This set of macros is defined in order to start the versatile ParaVision Acquisition and Reconstruction pipeline modes. The main macro PVSCANPIPE_CMD(cmd) activates the current dataset by selecting it in the Scan Control List and by waiting.
for the selection to be finished, followed by starting the command \texttt{cmd}. Finally, the macro waits for the acquisition or reco pipeline to be finished.

In order to help programmers define the correct acquisition or reco start command, some wrapper macros have been introduced. They should be self-explanatory:

\begin{verbatim}
PVSCAN_SCAN # traffic light
PVSCAN_GOP
PVSCAN_GSP
PVSCAN_TUNING
PVSCAN_WOBBLE
PVSCAN_GSAUTO(cmd_str)
PVSCAN_RECO
PVSCAN_RECO_WED  # reco with edit
\end{verbatim}

The following old macros are deprecated and should no longer be used in new Automations:

\begin{verbatim}
GOP, GSP, GSAUTO(cmd_val), RECO, RECO_WED
\end{verbatim}

**Macros prompting the operator for input**

- \texttt{GETDOUBLE(text,d1)} - Prompt the operator to enter a double value
- \texttt{GETFLOAT(text,f1)} - Prompt the operator to enter a float value
- \texttt{GETINT(text,i1)} - Prompt the operator to enter an int value
- \texttt{GETSTRING(text,nam)} - Prompt the operator to enter a char* pointer (text string)

**High resolution spectroscopy**

Please refer to the "\textit{AU programs Reference Manual}" available on the \textit{TOPSPIN / ParaVision 5.1} release CD. This document can also be downloaded from the registered web site [http://www.bruker-biospin.com/documentation_programming.html](http://www.bruker-biospin.com/documentation_programming.html).

**PARX related AU macros commands and functions**

For a complete list refer to "Parameter handling in Automations" on page D-11-26.
### Automation Programming

#### Calling AU’s and TOPSPIN commands

- **CPR_exec(....)**: C-function for executing *ParaVision* commands
- **WAIT_UNTIL(....)**: Hold the AU program until the specified date and time
- **XAUA**: Execute the acquisition AU program stored in AUNM (eda). The next line in the AU program is executed after the AU program AUNM has finished.
- **XAUP**: Execute the processing AU program stored in AUNMP (edp). The next line in the AU program is executed immediately after the AU program AUNMP has been started.
- **XAUPW**: Execute the processing AU program stored in AUNMP(edp). Like XAUP, but now the next line in the AU program is executed after the AU program AUNMP has finished.
- **XAU(prog)**: Execute the AU program prog with the wait option.
- **XCMD(cmd)**: Execute the command for which no dedicated macro exists.
- **XMAC(mac)**: Execute a *TOPSPIN* macro mac.

#### BRUKER library functions

- **GetNmrSuperUser()**: Get the name of the current *NMR* superuser
- **getdir(....)**: Get all file names and/or directory names within a directory
- **freedir(....)**: Free memory allocated by getdir
- **dircp(....)**: Copy a file
- **dircp.err(i1)**: Return the error message that corresponds to the error returnvalue of a dircp function call
- **Mkudir(....)**: Create a complete directory path
- **PathXWinNMR()**: A class of functions which return pathnames to certain *TOPSPIN* directories (see section “Platform independent programming” on page D-11-14).
- **pow_next(i1)**: Round i1 to the next larger power of two
- **Proc_err(....)**: Show a message in a window on the *TOPSPIN* screen
- **Except_printf()**: Terminate program and display message
- **Show_status(text)**: Show a string in the status line of *TOPSPIN*
- **showfile(file)**: Show the contents of a file in an *TOPSPIN* window
- **ssleep(i1)**: Pause in an AU program for i1 seconds
**Automation Programming**

**Macros to return from an AU program**

11.1.7.12

- **ABORT** - Abort the AU program or any of its subroutines with the return value of -1.
- **ERRORABORT** - Return from an AU program or any of its subroutines with the value of AUERR if it is less than 0.
- **QUIT** - Return from an AU program with the value of AUERR. QUIT is usually the last statement of the AU program code.
- **QUITMSG(text)** - Print the text message and then return from the AU program with the value of AUERR. This is an alternative to QUIT.
- **STOP** - Stops the AU program with the return value of AUERR.
- **STOPMSG("text")** - Stops the AU program with the return value of AUERR and displays the message "text".

**Parameter handling in Automations**

11.2

**General**

11.2.1

This section describes how ParaVision parameters can be accessed (read or set) from within Automations. ParaVision parameters are handled by the Parameter Space Handler described in “PARX - The Parameter Handling Mechanism” on page D-13-1. This mechanism includes a library of functions and C-macros which constitute an interface to the parameters. This interface is presented in the following sections. It is recommended to use the PARX macros whenever possible.

All examples in the following sections of this chapter are not complete Automations. They only show the use of the described PARX functions.

**Parameters in ParaVision**

11.2.2

ParaVision uses parameters for different purposes:

- parameters describe datasets,
- configuration information may be stored in parameters,
- parameters describe operations in ParaVision (e.g. reconstruction),
- etc.

The handling mechanism of parameters is named PARX, which can be regarded as a simple special purpose database.
There are three ways to access parameters in ParaVision:

1. Parameters can be accessed in Methods (see chapter “Method Programming” on page D-8-1).

2. Parameters can be accessed using the pvmcmd command utility (see “ParaVision Macros” on page D-10-13).

3. Parameters can be accessed in Automations.

This chapter will only deal with the third case. It is not possible to mix these three cases, e.g. it will lead to errors if a developer of an Automation uses functions designed for methods and vice versa.

A short introduction into ParaVision parameters and an overview of the acquisition parameters can be found in chapter “ParaVision Parameters” on page D-13-1.

Every parameter may have an associated Relation (function) which will be called when the parameter has just been changed. It is responsible for the consistency of this parameter in relation to other parameters. Therefore, it may change values of other parameters or redimension arrays silently.

Parameter Classes

Parameters can be combined to form parameter classes. For example, all parameters describing the reconstruction process including in- and output parameters are combined in the parameter class RECO. The existing classes are described throughout this manual. The main purpose of parameter classes is to store all parameters of a class in a file. Most of the files describing a dataset (see “Parameter files” on page D-12-5) are created by storing parameters classes in the corresponding files of the dataset directory.

Parameter Spaces

General

A parameter space is a collection of parameters with their values. Parameter spaces can be dataset oriented, i.e. they contain all parameters of a ParaVision dataset. A parameter space may be local to a process (process local) or shared by multiple processes (global).

Parameters of a process local parameter space can only be accessed in the process that created the parameter space. Changes in these parameters are not seen by other ParaVision processes (e.g. Scan Control).
Parameters of a shared parameter space are visible to all processes that are connected to this parameter space. Changes of values are immediately shared by the different processes, e.g. the change of the \texttt{ACQ\_dim} parameter in an Automation is seen in the Scan Control process.

When the parameter space is initialized no parameter has a value. The initialization routines of the parameter space are called which may initialize some parameters. If a dataset is loaded into the parameter space the parameter files initialize parameters. Having loaded parameter files post loading routines and the relations of the loaded parameters are called (except for COMPLETED datasets).

Within the Automation code a parameter space is identified by a unique parameter space identifier. This identifier is only unique for all open parameter spaces in a specific process.

### Loading associated datasets  

When starting an Automation a dataset path is associated with the Automation. This dataset is loaded into a shared parameter space with 

\texttt{PARX\_INIT}

The \texttt{PARX\_INIT} call is mandatory for Automations using PARX. Please do not use the \texttt{GETCURDATA} macro if the PARX Handler is used in the Automation.

The parameter space identifier for the dataset associated with the Automation is named \texttt{PARX\_psid}. Parameter space identifiers are of type \texttt{int}. \texttt{PARX\_INIT} throws an exception if the parameter space cannot be initialized.

The associated parameter space can be changed using 

\texttt{SETCURDATA}

if the variables \texttt{disk}, \texttt{user}, \texttt{name}, \texttt{expno}, and \texttt{procno} are changed. Before calling \texttt{SETCURDATA} these variables must point to a valid dataset. An exception is thrown in case of an error.

### Shared parameter spaces  

The associated parameter space is a shared parameter space. Additional shared parameter spaces can be created by writing the following code:

\begin{verbatim}
    YesNo acqReIs = No;
    YesNo recoReIs = No;
    int psid = ParxOpenWorkDataset(procno_path, acqReIs, recoReIs);
    if (-1 != psid)
\end{verbatim}
Automation Programming

{  
   /* do something with the parameter space */
}

This code excerpt opens a shared parameter space and loads the dataset located at procno_path. If the dataset is COMPLETED method relations will not be performed, e.g. the parameters acqRels and recoRels will be No.

Local parameter spaces

To create process local parameter spaces the following sequence can be used:

int psid = -1;
CreateUninitializedTemporaryPs(&psid);
if (-1 != psid)
{
   /* do something with the parameter space */
}

To load an arbitrary dataset into this parameter space the following function can be used:

int ParxInitPsFromDatapath(
   int psid,  /* valid parameter space id */
   const char * path, /* path until PROCNO directory */
   YesNo reset, /* reset parameter space before loading ? */
   YesNo exp_only, /* only experiment classes, e.g. no D3 */
   YesNo acqRels, /* perform ACQ relations */
   YesNo recoRels /* perform RECO relations */
)

acqRels and recoRels have the same meaning as in section D-11.2.4.3.

Closing parameter spaces

In principle, all parameter spaces have to be closed before exiting the Automation. The parameter space of an associated data set identified by PARX_psid is closed automatically by ParaVision, therefore, it is not needed to be closed by the automation programmer.

Parameter spaces are closed by

PARX_close(psid)

whereas psid is a valid parameter space identifier.
The parameter space of an associated dataset for an Automation (identified by PARX_psid) is closed automatically by ParaVision.

Accessing Parameters

Parameters can be changed and retrieved from a parameter space. Even though it is possible to set a parameter in a COMPLETED dataset it is strictly not recommended to do so since a parameter space of a COMPLETED dataset contains only parameters stored in the corresponding parameter files. Since only parameter space initialization code was performed (see “Parameter Spaces” on page D-11-27) all parameters not stored in the parameter files will not have values. Therefore, setting parameters in COMPLETED datasets may have unpredictable result.

Qualified parameter names

Parameter names are strings and identify the entire parameter value (e.g. an entire array of ints). Qualified parameter names describe a part of the parameter value. This can be an indexed value of an array or an element of a structure.

For example, the developer defines the following parameter definition:

```c
typedef struct
{
    int elem1;
    int elem2;
} MYSTRUCT;

int parameter MyParm[10];
MYSTRUCT parameter MyP;
```

Then the following qualified parameter names are possible:

"MyParm[1]" describes the 2nd int value of the array MyParm.
"MyP.elem1" describes the element elem1 of parameter MyP.

When parameter name is written in the following sections it can also be a qualified parameter name.

Changing Parameter values

If a parameter value is changed the relation (i.e. the related function specified in the corresponding method relations) of the parameter will be called. The rela-
Automation Programming

...tion may affect the values or dimensionality of other parameters, e.g. the change of the parameter value for `ACQ_dim` redimensions the array `ACQ_size` (and also other arrays).

The general parameter setting function has the following interface:

```c
int PARX_set_all_values(int psid, ...);
```

The first parameter is the parameter space identifier. The remaining parameters are pairs:

parameter name, pointer to parameter value

The parameter name is a string. The parameter value must be of the type defined by the qualified parameter name. This function can set values of an arbitrary amount of parameters. In order to determine the end of the list pairs a 0 must be specified. If a negative value is returned an error appears and the error description can be found in the global variable `PARX_errstr`. For example, to set the parameters `ACQ_dim` and `NI` the following code can be written:

```c
int dim = 3;
int ni = 4;
if (0 > PARX_set_all_values(PARX_psid,
               "ACQ_dim", &dim,
               "NI", &ni,
               0))
    fprintf(stderr, "%s\n", PARX_errstr);
```

There are several macros to set values of special types in the parameter space of the associated dataset:

- `SETIPAR(name, int-value);` /* set value of an integer parameter */
- `SETEPAR(name, enum-value);` /* set value of an enum parameter */
- `SETPPAR(name, double-value);` /* set value of a double parameter */

These functions will throw an exception if it is not possible to set the value.

Example:

```c
SETIPAR("NI", 4); /* set NI to int value 4 */
```

Retrieving and Changing array dimensions 11.2.5.3

*ParaVision* supports parameter arrays with variable length dimensions. Array dimension lengths of parameters may change during the lifetime of a parameter. They can be changed by:
• setting a dimension parameter. The relation of the dimension parameter will redimension arrays. For example, ACQ_dim sets dimension lengths of acquisition arrays like ACQ_size, etc.

• directly redimensioning the array. Some arrays are not redimensioned by dimensional parameters but can be redimensioned directly.

This section describes how a programmer can retrieve and change dimension lengths of array parameters directly.

The length of an array dimension will be retrieved by using the function:

```
int PARX_get_nth_dim(int psid, const char * parname, int dim_number);
```

The length of the \(<\text{dim_number}\)>th dimension of parameter \(\text{parname}\) is returned by this function. The dimension numbering begins with 1. If an error is encountered a negative value is returned and \(\text{PARX_errstr}\) contains the error message. If the parameter is not an array or \(\text{dim_number}\) is greater than the number of dimensions 0 is returned.

The dimension length of an array can be redimensioned with the following function:

```
int PARX_set_dims(int psid, const char * parname, ...);
```

If an error appears the function returns a negative value and \(\text{PARX_errstr}\) contains the error message. The dimension length(s) of the parameter \(\text{parname}\) will be changed. All dimension lengths must follow \(\text{parname}\). For example, \(\text{ACQ_grad_matrix}\) has three dimensions and the first dimension should be redimensioned to length \(10\):

```
PARX_set_dims(PARX_psid, "ACQ_grad_matrix", 10, 3, 3);
```

It is very important that the programmer of the Automation knows the definition of parameters. The \(\text{ACQ_grad_matrix}\) array, for example, is defined as \(\text{ACQ_grad_matrix}[][][3][3]\)

Only the first dimension can be redimensioned, all other dimensions are fixed. Nevertheless, it is necessary to specify all dimensions in a call to \(\text{PARX_set_dims}\).

The following example shows some operations and results with these functions. The example uses \(\text{ACQ_grad_matrix}\) but in reality it is not recommended to change \(\text{ACQ_grad_matrix}\) in an Automation. This may lead to unpredictable results to the geometry of the experiment.

Example:

```
PARX_set_dims(PARX_psid, "ACQ_grad_matrix", 10, 3, 3);
printf ("1st dimension of ACQ_grad_matrix: %d\n",
```
PARX_get_nth_dim(PARX_psid, "ACQ_grad_matrix", 1));
printf("2nd dimension of ACQ_grad_matrix: %d\n",
   PARX_get_nth_dim(PARX_psid, "ACQ_grad_matrix", 2));
printf("3rd dimension of ACQ_grad_matrix: %d\n",
   PARX_get_nth_dim(PARX_psid, "ACQ_grad_matrix", 3));

This code prints
1st dimension of ACQ_grad_matrix: 10
2nd dimension of ACQ_grad_matrix: 3
3rd dimension of ACQ_grad_matrix: 3

Retrieving Parameter values

Parameters can be retrieved with the following general function:

int PARX_get_all_values(int psid, ...);

It has the same interface as PARX_set_all_values of section D-11.2.5.2. In case of an error this function returns a negative value and writes an error description to PARX_errstr. If one of the parameters to be retrieved has no value a 0 is returned.

Example:

int dim;
char subjectid[12];
if (0 >= PARX_get_all_values(PARX_psid,
   "ACQ_dim", &dim,
   "SUBJECT_id", subjectid,
   0))
{
   /* error handling */
}
else
{
   /* do something with the parameters */
}

With the following function it is possible to test if a parameter has a value before trying to retrieve this parameter. The function cannot be called with a qualified parameter name:

int PARX_has_value(int psid, const char * parname);
The function returns 1 if the parameter parname has a value in the parameter space with identifier psid otherwise it returns 0. If no parameter with this name exists an exception is thrown.

Example:
```c
int dim;
if (PARX_has_value(PARX_psid, "ACQ_dim") &&
    0 < PARX_get_all_values(PARX_psid, "ACQ_dim", &dim, 0))
{
    /* the variable dim has the value of the parameter ACQ_dim */
}
```

Macros to retrieve parameters of the dataset associated with the Automation are:

- `GETIPAR(parname)` - return the value of the parameter parname if the parameter is of type int.
- `GETFPAR(parname)` - return the value of the parameter parname if the parameter is of type double.
- `GETEPAR(parname)` - return the value of the parameter parname if the parameter has an enum type.

All these macros throw an exception if the type of the parameter does not correspond to the macro or the parameter has no value.

Example:
```c
int ni = GETIPAR("NI");
double val = GETFPAR("ACQ_grad_matrix[0][0][1]");
```

### Saving Parameters

The following functions save parameters to a file:
```c
int ParxWriteClass(int psid, const char * classname,
                    const char * path, ParxWriteModeEnum mode);
int ParxWriteList(int psid, int cnt, const char ** parlist,
                   const char * path, ParxWriteModeEnum mode);
```

Parameters in the parameter class classname are saved to the file named path with ParxWriteClass. cnt parameters specified in the array parlist are written to the file path with ParxWriteList. The parameter names must not be qualified; mode describes the write operation:

- `ParxWriteNormalMode` - Parameters are written in JCAMP DX format for ParaVision.
Automation Programming

- **ParxWriteVisibilityMode** - Parameters are written in JCAMP DX format for *ParaVision* with visibility information. This information is very important for the different parameter editors in *ParaVision*. Therefore, all standard parameter files for datasets should be written with this mode.

- **ParxWriteForXWinNmrMode** - Parameters are written in JCAMP DX format for *TOPSPIN*. Since *TOPSPIN* does not understand structs and arrays of arbitrary length and dimension this mode can only be used for parameters of simple type. It should be used if it is planned to export these parameters to *TOPSPIN*.

If it is not possible to write the parameters to the file the functions return a negative value and store an error string in PARX_errstr.

**Examples:**

Write the subject class into a file named /tmp/subject with mode ParxWriteVisibilityMode:

```c
if (0 > ParxWriteClass(PARX_psid, "SUBJECT", "/tmp/subject", 
                        ParxWriteVisibilityMode))
    EXCEPT_printf("%s\n", PARX_errstr);
```

Write the parameter ACQ_dim and NI into the file /tmp/mypars.

```c
const char * pars[2];
pars[0] = "ACQ_dim";
pars[1] = "NI";
if (0 > ParxWriteList(PARX_psid, 2, pars, "/tmp/mypars", 
                       ParxWriteNormalMode))
    EXCEPT_printf("%s\n", PARX_errstr);
```

Two convenient macros exist for writing a class or list of parameters of the dataset associated with the Automation:

```c
WRCLS(classname,path),
WRLIST(list,path)
```

These macros write using ParxWriteNormalMode. They throw an exception in case of an error. list must be an array of parameter names as shown in the example above.

**Loading Parameters 11.2.7**

In order to load parameters from a file into a parameter space the following functions are available:
int ParxReadClass(int psid, const char * classname, const char * file, 
    YesNo callsRels);

int ParxReadList(int psid, int cnt, const char ** list, const char * file, 
    YesNo callRels);

ParxReadClass reads all parameters belonging to the parameter class classname from the given file. ParxReadList reads cnt parameters mentioned in list from the given file. Additional parameters stored in the file are not read.

callRels defines if method relations should be called after setting the parameters in the parameter space. It must be No in case of COMPLETED datasets.

Both routines return a negative value in case of an error and store an error description in PARX_errstr.

For example, if the parameters written in the examples in “Saving Parameters” on page D-11-34 should be retrieved in a local parameter space identified by psid the developer may write the following code:

static const char * pars[2] = { "ACQ_dim", "NI"};
if (0 > ParxReadClass(psid, "SUBJECT", "/tmp/subject", No))
    EXCEPT_printf("%s\n", PARX_errstr);
if (0 > ParxReadList(psid, 2, pars, "/tmp/mypars", No))
    EXCEPT_printf("%s\n", PARX_errstr);

Analogous to saving parameters in “Saving Parameters” on page D-11-34 two macros exist to read parameters into the parameter space of the dataset associated with the macros:

RECLS(classname,path,callRels),
RELIST(list,path,callRels)

Editing Parameters

An Automation can open a parameter editor and wait until the editor is closed by the user. This is only possible for parameters in the parameter space of the dataset associated with the Automation. Two macros exist that will open the usual parameter editor:

EDCLS(classname),
EDLIST(list)

EDCLS will display a parameter class in the editor. EDLIST will display a list of parameters in the editor. The user can change this parameters and press the OK button. In this case the values will be set into the parameter space, the Automation will continue, and the developer of the Automation can read the
changed parameters. If an error appears an exception is thrown. For example to edit the parameters NI and NR the following code can be written:

```c
static const char * parnames[2] = { "NI", "NR"};
int ni;
EDLIST(parnames); /* edit parameters */
ni = GETIPAR("NI"); /* fetch new value of NI */
```

Changes in *ParaVision* 5.0, 5.1 compared with 3.0

The following PARX macros have a changed interface in *ParaVision* 3.0.1 and in future versions:

- RECLS
- RELIST

Both macros are described in "Loading Parameters" on page D-11-35.

Since *ParaVision* 3.0.1 the following PARX macros for Automations are removed:

- DECODEPAR
- ENCODEPAR
- EDX

AU filters for Pipelined Acquisition

A major application field for AU programs is to serve as filters within the *ParaVision* Acquisition Pipeline. This kind of AU program is called AU pipeline filter. Pipeline filters allow operators to modify the data after it has been passed on from the acquisition but before it is displayed, written to disk or reconstructed. Applications of AU filters include:

- Accumulation of acquired data in a non-standard manner
- Rearrangement of data between the acquisition and the reconstruction required by techniques which traverse k-space in a non-standard manner
- Provision of extra information to the operator about the progress of the experiment.
- Automatic adjustment of hardware components
Structure of AU filters 11.3.1

An AU filter is written and compiled in the same manner as an AU program (refer to "Building AU executables" on page D-11-5 for more information). In terms of parameter setup, an AU filter is embedded into the acquisition pipeline when the parameter ACQ_user_filter is set to Yes and the parameter ACQ_user_filter_name contains the name of the AU filter.

There is an additional parameter which is used for allocating memory buffer for this filter. The parameter, ACQ_user_filter_memory, can take the values

- For_one_scan
- For_one_PE_step (PE = phase encoding)
- For_one_experiment

The structure of AU filters requires a number of macros. These macros represent the skeleton of every AU filter. The following code structure is generally used and should represent a good starting point for user defines filters:

```c
/**************************************************************************
AU filter : skeleton
**************************************************************************/

/* optional user declarations */

PIPE_INIT

/* optional user declarations */
/* optional user init code */

PIPE_PROC

/* optional user declarations */

/* ------------------------- */
/* THE FILTER CODE PUT HERE */
/* ------------------------- */

PIPE_WRAPUP

/* optional user declarations */
/* optional user wrapup code (free use-allocated memory) */
```
In order to control the data flow of an AU filter some additional macros are defined. They may only be used within the filter code, i.e. between the PIPE_PROC and PIPE_WRAPUP macros. Here is a list of macros controlling the dataflow throughput of an AU filter:

- **PIPE_WAIT_INPUT(x,y)** - Wait for at least x scans to be available in the input buffer, where at least y scans are contiguous (i.e. y scans fit into the buffer without swap around).
- **PIPE_WAIT_OUTPUT(x,y)** - Wait until there is space in the output buffer for at least x scans, where at least y scans are contiguous.
- **PIPE_WAIT_INPUT_OUTPUT(x1,x2,y1,y2)** - Wait for at least x1 scans to be available in the input buffer, where at least y1 scans are contiguous, and wait until there is space in the output buffer for at least x2 scans, where at least y2 scans are contiguous.
- **PIPE_SIGNAL_OUTPUT(x1,x2)** - Signal the system that x1 input scans and x2 output scans have been processed.

The numbers of contiguous scans (parameters y, y1, y2) are returned and may be less, equal or greater than the required numbers (parameters x, x1, x2). The in_ptr and/or out_ptr will be refreshed after execution of the corresponding PIPE_WAIT-macros. Concerning pointers see Table 11.1 on page D-11-41.

There is an additional macro, **PIPE_ABORT**, which will terminate the acquisition pipeline prematurely. This macro may only be used during the PIPE_INIT phase. Using the PIPE_ABORT macro initiates a jump directly to the PIPE_WRAPUP phase in order to free allocated memory and to perform other (user-defined) wrap-up tasks.

An AU filter example

Let us have a look at a real-world example. Some features may be unknown at the moment but they will be explained below. The more interested programmer may have a look into the file

```
<PvInstDir>/prog/inlude/aupipe.h.
```
It contains the definitions of all AU filter macros mentioned above. It also illuminates the underlying code structure of an AU filter when expanding the macro definitions.

/*file: GroupCorr --------------------------------------------------------*/
/* AU for acq. pipeline to correct the group delay of the digital filters*/
/* the last ACQ_size[0] data points to the next filter of the pipeline. */
/* This AU requires CPU-accumulation. CPU-accumulation can be established */
/* by setting GS_disp_update = GO Disp_update = Each_Accum. Please set also */
/* ACQ_user_filter_size[1] = 0. */
/*
/* Written by A. Nauerth, January 1997
/*------------------------------------------------------------------------*/
/* local variables */
static int corrSize;        /* number of bytes to throw away */
static int transferSize;    /* number of bytes to pass through */

PIPE_INIT
PARX_INIT

transferSize = acq_size[0] * sizeof(int);
corrSize = filter_size[0] - transferSize;
if (corrSize < 0) {
    printf(" wrong ACQ_size[0](%d)/ACQ_filter_size[0](%d)",
            acq_size[0],filter_size[0] / sizeof(int));
    PIPE_ABORT
}

PIPE_PROC
int scan_cnt;
int dummy;
while(1)
{
    for (scan_cnt=0; scan_cnt < filter_size[1]; scan_cnt++)
    {
        PIPE_WAIT_INPUT_OUTPUT(1,1,dummy,dummy)
        memcpy(out_ptr, in_ptr + corrSize, transferSize);
        PIPE_SIGNAL_OUTPUT(1,1);
    }
    if (acq_scan_type == Scan_Experiment) break;
scans_in = 0;
}
PIPE_WRAPUP
PIPE_END

There are some parameters that are used but which have not been declared nor initialized within the AU filter code, e.g. `filter_size` or `acq_scan_type`. These are global parameters and may be used in AU filters. Most of them may not be modified since they represent the basic conditions of an experiment. The parameters are either initialized at the commencement of the pipeline (should be read-only) or they are updated by the macros described above (especially scan buffer pointers and counters). Here is the complete parameter list:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Data type</th>
<th>Description</th>
<th>Controlled by</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter_nr</td>
<td>int</td>
<td>Internal number of filter</td>
<td>Initialization</td>
</tr>
<tr>
<td>sizeof_scan</td>
<td>int</td>
<td>Size of single scan in bytes</td>
<td>Initialization</td>
</tr>
<tr>
<td>sizeof_word</td>
<td>int</td>
<td>Size of data word in bytes</td>
<td>Initialization</td>
</tr>
<tr>
<td>scans_per_exp</td>
<td>int</td>
<td>Number of scans per experiment</td>
<td></td>
</tr>
<tr>
<td>scans_per_pe_step</td>
<td>int</td>
<td>Number of scans per PE step</td>
<td></td>
</tr>
<tr>
<td>in_ptr</td>
<td>char*</td>
<td>Start address of CURRENT input data block</td>
<td>PIPE_WAIT*</td>
</tr>
<tr>
<td>out_ptr</td>
<td>char*</td>
<td>Start address of CURRENT output data block</td>
<td>PIPE_WAIT*</td>
</tr>
<tr>
<td>xfer_cnt</td>
<td>int</td>
<td>Number of contiguous scans available</td>
<td>PIPE_WAIT*</td>
</tr>
<tr>
<td>scans_in</td>
<td>int</td>
<td>Number of processed input scans</td>
<td>PIPE_SIGNAL_OUTPUT</td>
</tr>
</tbody>
</table>

Table 11.1: Parameter list for pipeline filters
### Parameter List for Pipeline Filters

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Data type</th>
<th>Description</th>
<th>Controlled by</th>
</tr>
</thead>
<tbody>
<tr>
<td>scans_ou</td>
<td>int</td>
<td>Number of processed output scans</td>
<td>PIPE_SIGNAL_OU TPUT</td>
</tr>
<tr>
<td>acq_dim</td>
<td>int</td>
<td>Value of ACQ_dim</td>
<td>Initialization</td>
</tr>
<tr>
<td>acq_size</td>
<td>int[]</td>
<td>Value of ACQ_size</td>
<td>Initialization</td>
</tr>
<tr>
<td>word_size</td>
<td>enum</td>
<td>Value of ACQ_word_size</td>
<td>Initialization</td>
</tr>
<tr>
<td>filter_size</td>
<td>int[]</td>
<td>Value of ACQ_user_filter_size In case of index 0, the value is multiplied by the wordsize sizeof(int).</td>
<td>Initialization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Can only be used if ACQ_user_filter = Yes ACQ_user_filter_mode = Special</td>
<td></td>
</tr>
<tr>
<td>nr</td>
<td>int</td>
<td>Value of NR parameter</td>
<td>Initialization</td>
</tr>
<tr>
<td>ni</td>
<td>int</td>
<td>Value of NI parameter</td>
<td>Initialization</td>
</tr>
<tr>
<td>acq_scan_type</td>
<td>enum</td>
<td>Defines type of experiment: Scan_Experiment Setup_Experiment Wobble_Experiment</td>
<td>Initialization</td>
</tr>
<tr>
<td>phase_factor</td>
<td>int</td>
<td>Value of ACQ_phase_factor</td>
<td>Initialization</td>
</tr>
</tbody>
</table>

**Table 11.1:** Parameter list for pipeline filters
**Automation Programming**

---

**Allocation of memory**

A filter often requires additional memory in order to manipulate the data and store intermediate results. This is achieved by using the `malloc` function or shared memories. If the user allocates memory using the identifiers

```c
char *buf1, *buf2, *buf3;
```

the memory would be freed automatically during the wrap-up phase. It is the programmers task to free alternative memory, explicitly. Refer to "Allocating memory" on page D-11-19 for further information.

---

**Debugging AU filters**

This section provides a step-by-step method to debug AU programs. A corresponding debugging guide for AU programs is written in "Debugging AU programs" on page D-11-6.

The simplest way to track the runtime behavior of an AU executable would be to insert `printf` statements at every source code line of interest. The following excerpt facilitates debugging source code by optionally switching debug messages on and off.

```c
#define DEBUG [0|1]  /* Choose 0 to switch off, 1 to switch on */
...
#if DEBUG
printf("value of parameter xyz = %d\n",(int)val);
#endif
```

It is obvious that such a debugging technique

- increases the complexity of the source code,
- only shows the program state for a fraction of values at a certain time,
- mixes relevant program outputs with sporadic debugging information,
- alters AU runtime behavior (decreases performance) if `printf` statements are used excessively

It is always more efficient to use a source code debugger during AU runtime execution. You can either use the internal command line interface of the GNU Debugger GDB. Alternatively, we strongly recommend to use DDD, the graphical frontend for the GDB. The DDD package is part of every Linux distribution and should exist with DDD version 3.3.1 or higher. Otherwise, refer to the web site of your Linux distributor in order to download/install the current supported RPM package. If not available refer to the original site [http://www.gnu.org/software/ddd/](http://www.gnu.org/software/ddd/) for the latest product version. RPM packages can be installed...
easily by executing the command

```
rpm -i <RPM_package_name>
```

Once the frontend is installed, debugging is very easy:

1. The AU filter source code must reside in the directory `<PvInstDir>/exp/stan/nmr/au/src`. It must compile and link without errors.
2. Open an internal *ParaVision* console window by clicking the **TERMINAL Icon** in the toolbar of the **System Control Tool**
3. Change into the directory `<PvInstDir>/exp/stan/nmr/au/src`
4. Edit the file `vorspann_f` and activate the line `sleep(30)`. The AU filter start must be delayed for a period of time until the debugger has attached to the runtime process of the AU filter. You can decrease the number of seconds if it seems to be too long for you to wait. Do not forget to restore your changes after debugging.
5. Execute the command `../makeau -verbose -debug <AUname>` and finally quit the command line based debugger `gdb` with ‘q’. We will use a graphical frontend for debugging called `ddd`. Now, a debug version of the AU filter was generated and installed to `<PvInstDir>/prog/mod/a.out`.
6. Activate the AU filter in the **ACQP Class Editor** of the **Spectrometer Control Tool**. Copy the AU filter `<PvInstDir>/prog/mod/a.out` to the directory `<PvInstDir>/prog/au/bin`
7. Start the Pipelined Acquisition. Wait until the message “<AU filter> : in progress” is displayed in the status line of the **TOPSPIN** processing window
8. Within the terminal window execute the command `ddd <PvInstDir>/prog/au/bin/a.out`. A debugger window opens. Attach the filter `a.out` to the Debugger by selecting the menu entry **File>Attach to Process**. A list of potential processes will be offered to the user to attach. Select the highlighted one, i.e. the first entry of a group of `a.out` processes in this list. Wait some seconds until a CPR object has been created for this process (can be observed in the debugger terminal window at the bottom of the main window).
9. Set the ‘Temporary Breakpoint’ at C code line `AU_proc()`, so type ‘`b AU_proc’ in the DDD command shell. A valid C code breakpoint line must be returned.
10. Finally press the **Continue** button or enter the command ‘c’ into the debuggers command line window.
11. The debugger breakpoint is expected to be reached nearly instantly by displaying a GREEN arrow at the breakpoint position. The debugger is now ready to operate on the AU filter.

12. Next: Step into the AU code!

List of BRUKER AU filters

Macro and parameter definitions can be obtained from the files:

<PvInstDir>/prog/include/aupipe.h
<PvInstDir>/exp/stan/nmr/au/vorspann_f

For the list of all AUs delivered with ParaVision 5.1 see "List of ParaVision AU Programs" on page A-7-4.

AU filter macros and commands

This section is a reference list of ParaVision related macros and commands specific to ParaVision AU filters. For a comprehensive list refer to the "AU programs Reference Manual" available on the TOPSPIN / ParaVision 5.1 release CD. This document can also be downloaded from the registered web site http://www.bruker-biospin.com/documentation_programming.html (registration is free of charge).

We strongly recommend you use only the capital letter version of a macro in order to avoid possible namespace errors or confusion with variable names. Some TOPSPIN related macros are also available in variants using lower-case letters, they may work, but we strictly discourage you from using them!

AU filter related macro and parameter definitions can be obtained from the following files:

<PvInstDir>/prog/include/aupipe.h
<PvInstDir>/exp/stan/nmr/au/vorspann_f

AU filter macros

PIPED_INIT - initializes AU filter code;
PIPED_PROC - introduces body of AU filter code;
PIPED_WRAPUP - cleanup at completion of pipeline;
PIPED_END - marks end of AU filter;
PIPED_ABORT - aborts the AU pipeline filter prematurely;
PIPE_WAIT_INPUT(inRequest, inAvailable) - is used to wait for scan to be available for processing. The first argument (inRequest) specifies the number of scan which should be available for processing. The second argument (inAvailable) is the name of a variable which will be set to the return value of this macro. If possible, the marco will return after at least inRequest scans are available for processing. The number of contiguous scans available for processing are returned in variable specified by the second argument.

For technical reasons, there is no guarantee that the required number of scans can be provided. In such a case, the return value will be less than the requested number of scans. Thus, inRequest must be regarded as a request. However, it is guaranteed, that there is at least one scan available for processing after the macro is terminated. After the macro is terminated, the public parameter in_ptr, defined as char *in_ptr, will point to the first data point of the next scan available for processing.

PIPE_WAIT_OUPUT(outRequest, outAvailable) - is used to wait for scans transferable to the successor process of the acquisition pipeline. Similar to PIPE_WAIT_INPUT, one can wait for one or more scans fitting in the output buffer. There is no guarantee that outAvailable is bigger or equal to outRequest. However, it is guaranteed that outAvailable is greater or equal to 1. After the macro is terminated, the public parameter out_ptr, defined as char *out_ptr, will point to the location to which the first data point of the first processed scan should be copied to.

PIPE_WAIT_INPUT_OUTPUT(inRq, inAvail, outRq, outAvail) - is the combination of the macros PIPE_WAIT_INPUT() and PIPE_WAIT_OUPUT(). After the return of the macro PIPE_WAIT_INPUT_OUTPUT(), the variables specified by inAvail and outAvail will be set to the number of scans available for input and output respectively. Also the character pointer variables in_ptr and out_ptr will be set to the start of the memory location for input and output.

AU filter functions 11.3.4.2

void reload_B0_init(void)
The function reload_B0_init() establishes the reload B0 mechanism for locking the B0 field during the runtime of an experiment (see ”Syntax of the gradient commands” on page D-5-78).

The function reload_B0_init() need to be located within the AU filter macros PIPE_PROC and PIPE_WRAPUP.
void reload_B0(double deltaB0)

The function `reload_B0(deltaB0)` transfers the specified $B_0$ correction value `deltaB0` to the real-time sequencer of the BRUKER spectrometer. The new value will be established during the execution of the pulse programming command `reload B0` (see “Syntax of the gradient commands” on page D-5-78).

The $B_0$ offset is automatically set to 0 during the start-up time of an experiment. Thus, the first (navigator-) scan of an experiment can be regarded as a reference scan which corresponds to a correction value of $\text{deltaB0} = 0$.

The $B_0$ correction value may have a value between $-\text{PREEMP}_0\text{cal}_\text{const}$ and $+\text{PREEMP}_0\text{cal}_\text{const}$. Other values are automatically truncated to its limit.

**Note:** The $B_0$ offset mechanism requires a gradient control unit which is equipped with a fourth channel for $B_0$ control (e.g. BGU2). Systems equipped with a former gradient control units simply ignore this functionality.

On *Avance III Biospec* systems $B_0$ and the linear shims are controlled by the Digital Preemphasis (DPP). With the following functions it is possible to reload DPP Shims via an AU filter, e.g. to adapt shims in Real Time experiments.

All these functions return a NULL pointer or an error string if something went wrong. The location of the requested value or values has to be given by a pointer argument.

cchar *GetNDppShims(int *ndppshims)

The function `GetNDppShims()` returns the number of shims supported by the DPP.

cchar *GetDppShimChannels(SHIM_PAR *channels)

The function `GetDppShimChannels()` returns the type of the shims supported by the DPP.

`SHIM_PAR` is an enumeration with the following elements:

```
{ Shim_Z, Shim_Y, Shim_X, Shim_XZ, Shim_YZ, Shim_Z2X, Shim_Z2Y, Shim_X3, Shim_Y3, Shim_Z2, Shim_X2_Y2, Shim_2XY, Shim_Z3, Shim_Z4, Shim_Z_X2_Y2, Shim_Z_2XY, Shim_Z0, Shim_Z5}
```

**Note:** the integer value of Shim_Z, Shim_Y, Shim_X, Shim_Z0 is 0, 1, 2, 16. Channels should point to an array with ndppshims elements (e.g. 4 elements, if $B_0$ and linear shims are supported).
char *GetDppShims(int *values)
The function GetDppShims() returns the current shim values, as they are transferred from the DPP to the corresponding DAC of the Shim or Gradient Unit.

char *ReloadDppShims(int *values)
The function ReloadDppShims() loads the values into the current shim set.
values should point to an array with ndppshims elements.

char *GetDppShim(SHIM_PAR channel, int *value)
The function GetDppShim() returns the current shim value of the specified channel.

char *ReloadDppShim(SHIM_PAR channel, int *value)
The function ReloadDppShim() loads the value into the specified shim channel.

char *GetDppNShimSets(int *nshimsets)
The function GetDppNShimSets() returns the current number of shim sets, load into DPP.

char *ReloadDppNShimSets(int *nshimsets)
The function ReloadDppNShimSets() changes the current number of shim sets within DPP.

char *GetDppShimSets(int *values)
The function GetDppShimSets() returns all shim sets, load into DPP.

char *ReloadDppShimSets(int *values)
The function ReloadDppShimSets() loads all shim sets into the DPP and loads the current shim set into the shim or gradient unit.
values should point to an array with ndppshims * nshimsets elements.

char *GetDppShimListIndex(int *shimlistindex)
The function GetDppShimListIndex() returns the index of the current shim set.

char *ReloadDppShimListIndex(int *shimlistindex)
The function ReloadDppShimListIndex() makes another shim set active.
Example, how to use these functions within an AU-filter:

```c
... int ndppshims;
(void)GetNDppShims(&ndppshims);
(void)printf("ndppshims=%d\n", ndppshims);

SHIM_PAR channels[4];
(void)GetDppShimChannels(&channels[0]);
for (i=0; i<4; i++)
    (void)printf("channels[%d]=%d\n", i, channels[i]);

int dppshim[4] = { 10, 20, 30, 40 };
(void)ReloadDppShims(&dppshim[0]);
(void)printf("dppshims=%d, %d, %d, %d\n", dppshim[0],
              dppshim[1], dppshim[2], dppshim[3]);

int shim_corr = 11;
(void)ReloadDppShim(Shim_X, &shim_corr);
(void)printf("X-Shim=%d\n", shim_corr);
shim_corr=0;
(void)GetDppShim(Shim_X, &shim_corr);
(void)printf("X-Shim=%d\n", shim_corr);
..;
```

The AU filter will produce the following output:

```
ndppshims=4
channels[0]=2
channels[1]=1
channels[2]=0
channels[3]=16
dppshims=10, 20, 30, 40
X-Shim=11
X-Shim=11
```