Developer's Toolkit for DAPL Manual

Command module developer's
toolkit for DAPL 2000
operating system

Version 5.00

Microstar Laboratories, Inc.

Microstar Laboratories, Inc.
2265 116 Avenue N.E.
Bellevue, WA 98004
Tel: (425) 453-2345
Fax: (425) 453-3199

Part Number MSDTDM500-0104

Contents

1. Introduction

The Developer's Toolkit for DAPL contains the software tools required for creating custom command modules for Microstar Laboratories Data Acquisition Processors. Custom command modules are downloadable binary modules that contain custom commands. Custom commands are user-defined processing commands that extend the DAPL 2000 operating system. Most applications require only the data processing functions available as predefined DAPL commands, so the Developer's Toolkit for DAPL is intended primarily for advanced users.

Custom commands are written in C or C++, compiled and stored in the host PC, and downloaded from the PC to a Data Acquisition Processor. Once custom command modules are downloaded, the new commands are used in DAPL processing procedures in the same manner as predefined DAPL commands.

Processing procedure definitions within a DAPL configuration script refer to predefined or custom commands by name. Modules containing custom commands are loaded into the DAPL system before sending the DAPL configuration script. Each reference to a command is called a “task definition” because it results in creation of a processing task when the configuration runs. A task definition specifies a list of parameters, identifying the data sources and data destinations to be used by the task.

When the Data Acquisition Processor receives a `START` command for a processing procedure containing a task definition, the Data Acquisition Processor activates the task, executing the command code. The command first extracts the parameter information provided by the Data Acquisition Processor, checking that the parameters are valid. The command then executes its initialization code. After initialization, the task executes an endless processing loop. This loop reads data from pipes or variables, processes the data, and writes the results to pipes or variables. The task processes data indefinitely until the Data Acquisition Processor is stopped.

Pipes provide the connections for data to move between tasks. The pipes specified in a command parameter list may be communication pipes, input channel pipes, user-defined pipes, or other types. It makes no difference within the command; all pipes are treated uniformly.

This manual explains how to create and use custom command modules. The reader should be familiar with the operation of Data Acquisition Processors and DAPL system, as described in the introductory sections of the DAPL manual.

Supported Systems

The Developer's Toolkit for DAPL version 5 supports DAPL 2000 versions 2.0 and above. Some older DAP models are not supported under the DAPL 2000 version 2 system. The Developer's Toolkit for DAPL version 4, the DAPL 2000 system version 1 or the DAPL version 4 system, and a 16-bit compiler are required for these older Data Acquisition Processors.

16-bit custom commands developed using the Developer's Toolkit for DAPL version 4 will still download and run under the DAPL 2000 version 2 system, but these cannot take advantage of new features of the operating system, and they are subject to certain constraints imposed by the 16-bit programming model. The Developer's Toolkit for DAPL version 5 does not support development of 16-bit custom commands, but ordinarily there is no reason to use a 16-bit environment.

The Developer's Toolkit for DAPL supports the following compilers:

- Microsoft Visual C++ 6.0
- Borland C++ Builder 5.0 or Borland C++ 5.5 command line compiler

The Developer's Toolkit for DAPL provides startup, module registration, and run time service functions. Support is provided for floating point operation, with replacements for non-portable Standard C floating point library functions. Floating point emulation is provided for Data Acquisition Processor models that do not have hardware-supported floating point operations. All features are built into each downloadable module automatically.

2. Installation

The Developer's Toolkit for DAPL is delivered as part of the DAPtools Professional package on CD-ROM. To install, place the CD-ROM into your CD-ROM reader. The Microstar Laboratories Setup Launcher (`SETUP.EXE` at the root of the CD) will run automatically. Select the Developer's Toolkit for DAPL link. You also can use Explorer or Run to browse for the DTD folder. Find the `SETUP.EXE` program in this folder and run it.

The installer program will place the Developer's Toolkit for DAPL software into the folder

```
C:\Program Files\Microstar Laboratories\DTD32
```

by default. If you select an alternative location, note the path that you select. You will need this information to modify some of the configuration files. This manual will usually presume that you have installed in the default location and will refer to this folder as `DTD32`. Keep in mind that this location may be assigned a different name on your system.

After the installation has been completed, you might wish to review the information in the `README.TXT` file. This file will be found in the base directory `DTD32` of the installation. Use any text editor, file viewer or word processor to view this file. It will provide notices of changes or recent corrections not covered by this manual.

The `FILES.TXT` file, which is also found in the base `DTD32` folder, contains descriptions of all of the files in the installation.

Adjusting Project Files

If the Developer's Toolkit for DAPL or compiler systems are installed in locations other than the suggested defaults, some of the example files must be modified to reflect the actual folder locations. The adjustments will depend on the software configuration specified when the software was installed. The adjustments can be made using any text editor, such as the Windows system `NOTEPAD` program.

Compiling Using the Command Line

The `DAPCC.BAT` can be used to make compiling command modules a little easier from the command line. There are three `set` commands near the end of this file that might

need adjustment depending on which compiler you select and where it is installed. The adjustments can be omitted if you never use the command line for compiling custom command projects.

The `NMPATH` macro must point to the folder where the compiler's make utility is located. If the compiler is not installed in the default location, this path information must be modified to match your installation.

The `NMMAKE` macro indicates which *make* utility to run. For Microsoft compilers, you can use the `NMAKE` program, and for Borland compilers you can use the `MAKE` program. It is possible to use either *make* utility with either compiler, but usually it is easiest to use each utility with its associated compiler.

The `NMTYPE` macro indicates which makefile to invoke. Specify `M` with the Microsoft `NMAKE`, or specify `B` with the Borland `MAKE`. The batch file uses this designation to pick which makefile to invoke.

Examples are provided within the `DAPCC.BAT` file comments, showing configurations appropriate for each compiler.

Compiling with Microsoft NMAKE

If you want to invoke the Microsoft compiler from the command line, and if the compiler or the Developer's Toolkit for DAPL is not installed in the default directory location, the *makefile* must be modified. At the top of the `MODMAKEM.MAK` file used with the Microsoft compiler, there are two macro lines:

```
VC6PATH = ...
DTDPATH = ...
```

These paths point to the base directory locations for the compiler system and the Developer's Toolkit for DAPL respectively. If your software is installed at different locations than the defaults, modify these two lines to the correct paths for your system configuration.

Compiling with Borland MAKE

If you want to invoke the Borland compiler from the command line, and if the compiler or the Developer's Toolkit for DAPL is not installed in the default directory location, the *makefile* must be modified. At the top of the `MODMAKEB.MAK` file used with the Borland compiler, there are two macro lines:

```
BCCPATH = ...
DTDPATH = ...
```

These paths point to the base directory locations for the compiler system and the Developer's Toolkit for DAPL respectively. If your software is installed at different locations than the defaults, modify these two lines to the correct paths for your system configuration.

Compiling with the Borland Builder IDE

It is possible to override the project group specifications of the Borland IDE to use a makefile other than the ones automatically generated by the IDE, but this is awkward in a number of ways. If the makefile and IDE configuration are inconsistent, the IDE has a way of eliminating the problem by eliminating the inconsistent files. This can lead to... let’s just say some non-productive use of your time.

The Developer's Toolkit for DAPL provides some “prototype” project files where all inconsistencies are eliminated and all special configurations are added. However, these files contain compiler and linker options very different from the ones that the compiler would choose. For compiling an existing command quickly, the command line is probably easier, but for a more complex application, setting up the IDE development environment is perhaps worthwhile.

If the compiler or the Developer's Toolkit for DAPL is not installed in the default directory location, the prototype project file must be modified. This file is called

```
MODULE.BPR
```

and it is located in the Developer's Toolkit for DAPL folder

```
DTD32\LIB\BC
```

This file can be modified using any text editor, but be careful to make only the desired changes. Use the editor search and replace feature, looking for the text string

```
C:\Program Files\Microstar Laboratories\DTD32
```

and replacing this text with the actual path to your Developer's Toolkit for DAPL base folder. This appears several times; replace each one. Be careful not to remove any termination characters such as slashes or quotes. Be careful not to insert any extraneous blank or other characters. Save your modified file, and as a precaution, set its file attributes to *Read Only* to make it more difficult to lose your changes.

This makes the project configuration available to use, but not ready to use. There are more steps required to prepare for each new project. These are covered in the next chapter.

Compiling with the Microsoft IDE

Fans of the Microsoft IDE who think the hacking of Borland configuration files is funny, now you have a chance to exercise some real humor. Microsoft project configurations are in various binary formats. Each projects is supposed to be set up individually through the IDE, and for the special custom command configurations this is not an easy process. All of the paths and names are compiled-in, and can't be changed. There is no possibility of generating a new project in an automatic fashion.

One thing that the Microsoft environment can do, however, is run Microsoft-format *makefiles*. A side effect of this is that most of the automatic configuration features of the IDE are disabled, and the system configuration is controlled entirely through the *makefile*. You can leave the project in one place and move custom command files in and out of the project to avoid most reconfiguration. Judge for yourself whether this is better than building from the command line.

The following steps will set up one project environment with a *makefile* that can be used to compile any custom command module.

1. Find a location where you want to do all of your custom command development. If you do not have such a place, set up a new folder.
2. If you installed either the Developer's Toolkit for DAPL or the compiler in folders other than the defaults, use a text editor such as `NOTEPAD` to edit the `MODMAKEM.MAK` file in your `DTD32\LIB\MC` folder. Adjust the `VC6PATH` and the `DTDPATH` macros as needed to point to the actual installed locations of the software.
3. In the Visual C IDE, select File | New.
4. In the New dialog box, select the Workspaces tab. The box at the left will show "Blank Workspace."
5. In the Location text box on the right side, select the location where you want to do your custom module builds.
6. In the Workspace Name text box just above that, enter a generic name for the module development project. The name `MODULE` will work and is used for this description.
7. Click the Okay button. An empty workspace with name `MODULE` will appear in the object viewer window in the box at the left.

8. Use Explorer or other means to copy the makefile `MODMAKEM.MAK` from folder `DTD32\LIB\MC` to the just-created `MODULE` folder.
9. Back in the IDE, select from the main menu Project | Insert Project into Workspace, or equivalently, right click on the empty project name in the viewer box.
10. In the Insert Project into Workspace dialog, go to the Files of Type text box at the bottom. Click the down arrow at the right and select Makefiles (.mak).
11. You should see the `MODMAKEM.MAK` file copy in the file list box above. Click on the file name to select it, and then click the Okay button.
12. The environment will complain that the makefile was not built by Visual Studio. Click the Yes button to continue.
13. A small Platforms dialog will appear with the Win32 environment selected. Click Okay.
14. A Save As dialog box will appear, showing a file name `MODMAKEM1` for the IDE's compiled makefile wrapper. Click Save.
15. The new project wrapper file will appear in the object viewer pane on the left. Right click on the new `MODMAKEM1` name.
16. In the pop-up menu, select Settings.
17. A Project Setting dialog window will appear, with the General tab active. On this sheet, two changes are required. In the command line text box, insert some text after the `NMAKE` and before the `-f` flag.

    ```
    MODULE=modname
    ```

    Leave blanks before and after this text, no blanks inside.
18. In the same dialog, go down to the Output File Name text box and replace the name `MODMAKE.EXE` with a generic module name.

    ```
    modname.dlm
    ```

19. Click the Okay button. The `modmakem.mak` file will now be recorded in the project workspace.
20. From the main menu, select File | Save All.

This is all you have to do -- in your wildest dreams. The rest is covered in the chapter Compiling and Loading Modules.

3. Overview

The Developer's Toolkit for DAPL provides a programming interface supporting data acquisition and real time applications. The services provided by the DAPL 2000 operating system include:

- task control
- memory management
- text formatting and messages
- pipes and buffers
- asynchronous output
- software triggers
- FFT, FIR filters, PID control
- math functions

These functions are sometimes called *system functions* because the services are actually performed by the DAPL 2000 operating system, using exactly the same methods that the operating system uses internally. A system of *include* or *header* files defines the interface between C custom commands and the resources of the Data Acquisition Processor.

This chapter discusses the structure of a custom command code module, illustrating how *system functions* are used to build one particular custom command. While the processing in this example is relatively trivial, the structure can be generalized to a broad range of applications. More information about using the *system functions* is found in the chapter Using the Data Acquisition Runtime Library. Complete information about each function is found in the chapter Data Acquisition Runtime Library.

Organization of Custom Command Code

A custom command program can be organized into a sequence of sections:

- get access through the `DTD.H` interface
- identify the module elements
- define the required local data elements
- obtain access to run-time parameters
- initialize structures
- begin continuous run-time processing

An Example Custom Command

The following is a typical example. It can serve as a starting place for developing new custom commands.

This command is called `ZTRUNC`. It defines a task that reads data values from one pipe, limits values to a specified lower limit, and sends the modified data to another pipe. All of the details about the system functions used in this example are covered in later chapters.

To obtain access to the Developer's Toolkit for DAPL interface, include the header file `DTD.H`. Every source code file will include this. This source code module also defines the module name and entry point, so the `DTDMOD.H` file is also included.

```
#include "DTDMOD.H"
#include "DTD.H"
```

The command code must specify to the DAPL system the name of the command and the run-time code to be executed when this command runs. These things are done by setting up some macros at the beginning of the program code file.

```
#define  COMMAND  "ZTRUNC"
#define  ENTRY    ZTRUNC_entry
```

The next few lines implement the module identification sequence. Unless you are doing some advanced module development, these lines never change.

```
int __stdcall     ENTRY (PIB **plib);
extern "C" __declspec(dllexport) int __stdcall
   ModuleInstall(void *hModule)
{ return (CommandInstall(hModule, COMMAND, ENTRY, NULL)); }
```

Now the main body of command code must be defined. A custom task begins execution at the place indicated by the ENTRY macro.

The first section of the run-time code defines local variables needed at run-time. The `ZTRUNC` command needs the following:

- two variables for accessing the task parameter list

  ```
  void **argv;
  int argc;
  ```
- three variables for saving the handles obtained from that list

```
PIPE  * in_pipe;
PIPE  * out_pipe;
VAR   * limit;
```

- two additional variables for run-time processing

```
GENERIC_SCALAR  pipe_value;
short int   current_limit;
```

When task execution begins, the `param_process` function obtains access to the list of task parameters. The handles for the three parameters are assigned to the variables reserved for this purpose. The pipe parameters are defined using the `PIPES` command in the DAPL configuration scrip. The variable parameter is defined using a `VARIABLE` command in the DAPL configuration script

```
argv = param_process (plib, &argc, 3, 3,
    T_PIPE_W, T_VAR_W, T_PIPE_W);
in_pipe = (PIPE *) argv[1];
limit   = (VAR  *) argv[2];
out_pipe= (PIPE *) argv[3];
```

The next code section initializes the task for runtime. Each pipe must be identified to the DAPL system to synchronize data sources and consumers. No special initializer is required to use the shared variable parameter.

```
pipe_open (in_pipe, P_READ);
pipe_open (out_pipe, P_WRITE);
```

The last section defines the continuous run-time processing. Data is fetched from the source data pipe. A computation is applied to determine the output value. The resulting output value is placed into the output data pipe. This sequence is very general. Different commands expect different kinds of data, apply different kinds of computations, and send their results to different locations, but the overall pattern is the same.

For the special case of the `ZTRUNC` command, each data value is compared to a lower limit, and values below the limit are adjusted. The following is the run-time loop of the `ZTRUNC` command.

```
while (1)
{
    pipe_value_get(in_pipe,&pipe_value);
    current_limit = *limit;
    if  (pipe_value._i16 < current_limit)
        pipe_value._i16 = current_limit;
    pipe_value_put(out_pipe, &pipe_value);
}
```

For reference, a complete listing of the `ZTRUNC` command code is provided. The source code is also available in the DTD32\EXAMPLES folder.

```
// ZTRUNCM.CPP - Module for ZTRUNC command
#define  COMMAND  "ZTRUNC"
#define  ENTRY    ZTRUNC_entry

#include "DTDMOD.H"
#include "DTD.H"

int __stdcall     ENTRY (PIB **plib);

extern "C" __declspec(dllexport)
   int __stdcall ModuleInstall(void *hModule)
{ return (CommandInstall(hModule, COMMAND, ENTRY, NULL)); }

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE  * in_pipe;
    PIPE  * out_pipe;
    VAR   * limit;

  // Storage for processing
    GENERIC_SCALAR  pipe_value;
    short int   current_limit;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W, T_VAR_W, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];
    limit   = (VAR  *) argv[2];
    out_pipe= (PIPE *) argv[3];

  // Perform initializations
    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);

  // Begin continuous processing
    while (1)
    {
        pipe_value_get(in_pipe,&pipe_value);
        current_limit = *limit;
        if  (pipe_value._i16 < current_limit)
            pipe_value._i16 = current_limit;
```

```
            pipe_value_put(out_pipe, &pipe_value);
        }
    }
```

In the ZTRUNC example, these identifiers are used: PIB, T_PIPE_W, T_VAR_W, P_READ, P_WRITE, **param_process**, **pipe_open**, **pipe_get**, and **pipe_put**. These terms are defined in the header files included by DTD.H. See the chapter Using the Data Acquisition Runtime Library to learn more about these functions. See the chapter on Include Files for more information about the organization of the header file system.

To run the ZTRUNC custom command, the custom command code must be compiled, linked, and downloaded from the PC to the Data Acquisition Processor. That process is explained in the next chapter, Compiling and Downloading.

After downloading the ztruncm.dlm module, the ZTRUNC command can be used in any processing procedure, as in the following example:

```
; Script to run the ZTRUNC custom command
RESET

PIPES P1
VARIABLE VLIM = 0

IDEFINE A 1
SET IPIPE0 S0
TIME 10000
END

PDEFINE B
  ZTRUNC (IPIPE0, VLIM, P1)
  FORMAT (P1)
END
```

This configuration uses only one ZTRUNC task. Samples are obtained from a single data stream. The ZTRUNC processing is applied, and the results are formatted in text form for display.

The ZTRUNC command could be used to define more than one task in more than one processing procedure. Each ZTRUNC task executes independently.

Many other kinds of processing tasks can be based on the structure of the ZTRUNC custom command. The ZTRUNC command is an example of a general *filter*, a command that produces one output value for each value input. Change the rule for

how the output value is generated from the input value and you have a new kind of *filter* command.

4. Compiling and Loading Modules

This chapter provides information about how to compile custom command modules and download them to your PC system.

After a custom command source code is written according to the instructions provided in this manual, the code must be compiled, and the object code generated by the compiler must be linked to form a single code image. Make sure that you have completed all of the installation steps described in the Installation chapter.

Preparing Files and Environments

This section describes the setup operations that you must apply to prepare each custom command project. (The chapter Installation describes setup operations that are applied only one time when the software is first installed.)

Command Line Environment

For either compiler, locate the folder where all of the required source code modules are located. In the command window, make the drive where the source files are located the default drive. For example,

```
C:
```

Use the `CD` command to make the folder containing the source modules the active directory, for example,

```
CD  \Projects\MyFiles\mymodule
```

Make sure that the `DTD32\LIB` folder is on the execution path if you will be using the `DAPCC.BAT` file. Make sure that the *make* utility is on your execution path if you invoke the *make* utility directly. The execution path can be set up in your `AUTOEXEC.BAT` file if you wish, so that it is automatically set up each time your computer is restarted.

If your custom command project uses more than one source module, modify the appropriate makefile, `MODMAKEM.MAK` or `MODMAKEB.MAK`, to include rules for compiling additional source files.

Microsoft IDE Environment

There are two choices.

1. Move all source files that you need to compile to the project directory set up for building custom commands at install time.
2. Repeat the steps describe in the Installation chapter for setting up a project wrapper to compile your source code in its current location.

If your custom command project uses more than one source module, modify the `MODMAKEM.MAK` *makefile* to include rules for compiling for the additional source files.

Run the IDE and use the main menu File | Open Workspace to open the project file. Select from the main menu Project | Settings. In the Project Settings Dialog, the General page will be active. In the Build Command Line text box, replace the name *module* with the actual name of the command module that you are going to build. In the Output File Name text box, replace the name *module* with the same name that you entered in the Build Command Line box, except retain the *.dlm* extension.

Borland IDE Environment

Copy the `Module.*` project files that are you set up when the Developer's Toolkit for DAPL was installed, placing the copies in the folder with the source code modules you wish to compile. Rename these project files, changing the name from `Module` to the name of the module that you wish to build, retaining the original file extensions. For example, if the new module name is *mymod*,

```
Module.bpr
```

would become

```
mymod.bpr
```

With a text editor such as NOTEPAD, edit each of the three renamed project files, replacing the text "*Module*" with the new name wherever found.

If you have a PERL interpreter, the changes can be made without using a text editor. Specify a PERL command line of the form

```
perl -e s'Module'mymod'g,print  -i -n  projfilename
```

Substitute your desired project name for *mymod*. Apply this command line three times, replacing *projfilename* with `Module.bpf`, `Module.bpg`, and `Module.bpr` respectively.

Compiling From the Command Line

The custom command module is built using a *makefile* utility. The Borland and Microsoft *makefiles* have slightly different formats, so separate files are provided in the `DTD32\LIB\BC` and the `DTD32\LIB\MC` directories respectively. These *makefiles* must be configured when the Developer's Toolkit for DAPL is installed.

The *makefiles* can be run directly from the command line. Specify as the target of the *make* operation the name of the command module (not the command name!). For the Borland compiler, the command line has the form

```
MAKE   MODULE=modulename  -f CMDMAKEB.MAK
```

and for the Microsoft compiler the command line has the form

```
NMAKE   MODULE=modulename  -f CMDMAKEM.MAK
```

Replace `modulename` with the actual name of your source code module, except leave off the three-letter extension. For example,

```
NMAKE   MODULE=ztruncm  -f CMDMAKEM.MAK
```

This will compile the `ztruncm.cpp` module to produce the `ztruncm.dlm` downloadable binary module, using the Microsoft `NMAKE` utility and compiler.

Simplified Command Line with Batch File

If you configured the `DAPCC.BAT` file in the `DTD\LIB` folder when the Developer's Toolkit for DAPL software was installed, this shortcut can be used for compiling with either compiler.

The command line is

```
DAPCC    mymodule
```

substituting the actual module name for *mymodule*, and leaving off the file type extension. This is easier to type and remember, but inside it is really no different than the MAKE or NMAKE command lines shown above.

Compiling from the IDE

Compiling from the Borland IDE

If you use the Borland IDE to develop your custom command projects, you can use the custom command workspace that you set up. Select Project | Make *project*, where the menu system will show your actual configured project name in place of *project*. The compiler will invoke the linker automatically and generate a module that is ready to download.

You cannot use the Run menu or any of its trace and debug options. Unfortunately, the IDE is unaware of the DAPL system, so the code cannot execute in the IDE environment.

Compiling from the Microsoft IDE

If you use the Microsoft IDE to configure your custom command projects, you can use the custom command workspace that you set up. Select Build | Build *module.dlm*, where the menu system will show your actual configured project name in place of *module*. The compiler will invoke the linker automatically and generate a module that is ready to download.

You cannot use the Run menu or any of its trace and debug options. Unfortunately, the IDE is unaware of the DAPL system, so the code cannot execute in the IDE environment.

Adjusting Compiler Optimizations

Code optimization options for either compiler can be adjusted if necessary, but unfortunately, not through the IDE environments.

For the Microsoft compiler, compiler optimizations are specified in the *makefile* and not controlled by the IDE environment. The options can be adjusted using a text editor. Find the `CCFLAGS` macro in the `MODMAKEM.MAK` *makefile*, and select -O options as described in the compiler user's guide.

If the Borland compiler is invoked from the command line, its optimizations can be adjusted in a similar manner. Just edit the `-O` command line options in the `CCFLAGS` macro in the `MODMAKEB.MAK` *makefile*.

The Borland IDE can adjust compiler optimizations, but it should *not* be used to do this. When the Builder IDE updates compiler options, it rewrites the entire `CFLAG1`

value field in its project file. However, in doing so, it changes the special settings required by custom command modules, replacing them with generic settings for building PC applications. It is possible, however, to use a text editor to modify `-O` command line options in the `CFLAG1` value field of the `project.bpr` file.

Downloading the Compiled Modules

Compiled modules can be configured to install when your DAPL system starts. To do this, do the following.

1. Use the Windows system Start button, select Setting | Control Panel | Data Acquisition Processors to activate the server configuration application.
2. Click the Module tab. The page will show the Installed Modules Display Window.
3. Click the Add button.
4. In the text box Select Target DAPs and Options, enter the path to the downloadable module. Or, click the Browse button and navigate the selection tree to find the file; click on the file name to highlight it, and then click the Open button.
5. The full path to the downloadable module will now be in the text box. The options Copy, Load and Replace are checked by default. Deselect the Copy option to use the commands under test from their original location. (After development is completed, the module can be reinstalled using the Copy option if you wish.)
6. Click the Okay button. The Installed Modules Display Window should now show the new downloadable command in the list.
7. Click the Close button to end this server configuration session.

The control panel application will remember the path to the module and automatically fetch the downloadable module each time the DAPL system is restarted.

During development of a new command, you will probably need to make revisions and replace the previous version of the installed module with an updated one. To do this, do the following:

1. Use the Windows system Start button, select Setting | Control Panel | Data Acquisition Processors to activate the server configuration application.
2. Click the Module tab. The page will show the Installed Modules Display Window.

3. In the Installed Modules Display Window, find the module name and click on the module name to select it.
4. Click the Reload button.
5. A reload dialog will show an option box with some options for reloading. Select the Replace option so that the DAPL system knows that it is okay to load the module even though a module with this same name was loaded previously. Select the Force option to make sure that the downloading of the fresh module takes place even if the old version is running currently.
6. The Reload dialog will also show a DAP box displaying all of the Data Acquisition Processors available to the server. If the module should be loaded only to certain selected Data Acquisition Processors, deselect ones that should not receive the downloaded copy.
7. Click the Okay button. The server control panel application will perform the download sequence.
8. Click the Close button to end this server configuration session.

If you need to do many experimental runs that involve module loading, especially if you are working with the command line, you might not want to go through the steps above each time. An alternative is to use a command line utility `MODLOAD`. `MODLOAD` is released as a `DAPIO32` programming example with source code. Look for it in the `DTD32\DapDev\Examples\C\Modload` folder. If you do not find this folder on your system, run the main SETUP program on your DAPtools Professional CD-ROM, and on the splash screen click *DAP Development*. Make sure that the `DTD32\DapDev\Examples\C\Modload` folder is specified on your execution path. Type `MODLOAD` on the command line without specifying a parameter to see a brief display of command line options. Information about how the `MODLOAD` command works is available in the *DAPIO32 Manual*.

To run `MODLOAD`, first make sure that the Data Acquisition Processor system is started. Select the drive containing your module as the default drive, and use the `CD` command to navigate to the folder containing the module to download. Then enter the command line

```
MODLOAD   module.dlm
```

substituting your actual module name for *module.dlm* .

If you want to specify a Data Acquisition Processor other than the default one at location `\\.\Dap0`, you can specify a network path in UNC format. For example,

```
MODLOAD   module.dlm  \\Station4\Dap1
```

If it is possible that a previous version of the module under development is already running, the processing can be forced to stop, to allow the download, by specifying an additional command line flag

```
MODLOAD   /force  module.dlm
```

5. Include Files

A system of function prototypes, structure definitions, data types and macros becomes available when the DTD.H and DTDMOD.H files are included into a source code module. The files discussed in this chapter are found in the DTD32\INCLUDE folder.

The DTDMOD.H File

The DTDMOD.H must be included in one custom command source code module, the one that defines the command identity and entry points. For the most part, this file concerns things that happen automatically. Some special cases are covered in the chapter on Advanced Applications. Most developers will just include this file and not worry about what it contains.

The DTD.H File

The file DTD.H is included with each source code module. It specifies a list of additional *include* files and some function prototypes. All custom command developers will need to use at least a few of these elements.

Besides including these supplementary files, the DTD.H file defines some functions required to support the C++ environment. These functions include:

```
operator new
operator new []
delete
delete []
```

Supplementary Header Files

The DTD.H file includes a set of additional header files. Developers who are developing specialized applications might find the detailed information in these files useful.

```
#include "GENTYPES.H"
```

This file defines some representation-specific and representation-independent data types for use within custom command programs. The data types BYTE, WORD and DWORD are used to refer to data elements of 8-, 16- and 32-bit length respectively. The

union data types `gen_ptr` and `gen_scalar` define elements that can accept values of any scalar type.

```
#include "DTDTYPES.H"
```

This file defines some special data types used in the DAPL system environment. The types `VAR` and `LVAR` are used to access shared variable parameters, while the types `CONSTANT` and `LCONSTANT` are used to access shared constant parameters. The structure type `PIDCOEF` is used by the PID control functions for control loop tuning.

```
#include "DTDHNDLS.H"
```

This file defines some special data types that are used by the DAPL system for accessing internal data management structures. These data types have no direct meaning outside of the DAPL operating system, but they are useful for identifying a particular internal structure within a custom command. These data types include:

- `PBUF` - Handle for pipe buffer control structure
- `PIPE` - Handle for data pipe control structure
- `TRIGGER` - Handle for trigger control structure
- `VECTOR` - Handle for shared vector control structure
- `PIB` - Handle for task parameter access list structure
- `PID` - Handle for PID state structure
- `FFTB` - Handle for FFT control structure
- `FIRB` - Handle for FIR filter control structure

```
#include "DTDFUNCS.H"
```

This file provides function prototypes for all functions available in the Developer's Toolkit for DAPL, as described in the Data Acquisition Runtime Library chapter.

6. Using Developer’s Toolkit Functions

This chapter shows how to use the system functions provided by the Developer's Toolkit for DAPL. Several useful programming techniques to facilitate development of command modules are described.

Header Files

Every source code module for a custom command module will need to include the `DTD.H` file. This file provides access to the macros, data types and system functions that are needed for building custom modules.

```
#include  “DTD.H”
```

One of the source code modules identifies command entry points, as discussed in the next section. This source module must also include the `DTDMOD.H` file.

```
#include  “DTDMOD.H”
```

Registering Commands

Each command module must tell the DAPL system about the commands available in the module. This process is done at the time that the command module is loaded into the DAPL system.

Two functions support this processing. One is provided by the Developer's Toolkit for DAPL, and the other must be coded by the developer.

The first function is `ModuleInstall`. This function is coded by the command developer. The DAPL system will expect this function to be present, otherwise the command module cannot be loaded. Function `ModuleInstall` must accept one parameter. When the DAPL system calls this function, it will provide a handle that identifies the module, to make sure that commands are associated with the correct module.

```
ModuleInstall( hModule );
```

Inside the implementation of the `ModuleInstall` function, the module developer must use the function `CommandInstall` to identify commands to the DAPL system.

```
CommandInstall( hModule, pCmdName, pEntry, pProperties );
```

The function `CommandInstall` is provided by the Developer's Toolkit for DAPL and must be specified once for each command in the module. The arguments for the `CommandInstall` function call are:

1. `hModule`
   The handle passed by the DAPL system.
2. `pCmdName`
   A text string, enclosed in double-quotes, defining the name that the DAPL system will use to identify the command in processing configurations. Most of the command examples provide with the Developer's Toolkit for DAPL use a macro to represent this string.
3. `pEntry`
   The address of a function to be called for run-time processing of a command. Just copy the name of the function to be called. In previous versions of the Developer's Toolkit for DAPL this entry point was always called `main`. But clearly that doesn't work when there are multiple commands in the module, because each entry point must have a distinct name.
4. `pProperties`
   A pointer to a command information structure, usually set to `NULL`. For projects with unusual requirements for storage and stack management, look in the *Advanced Programming Techniques* chapter for more information about setting up a customized command information structure.

The following example is taken from the `ZTRUNCM.CPP` command module code. In this example, the developer has coded a `ModuleInstall` function that calls the function `CommandInstall` function one time to register the processing command called `ZTRUNC`, selecting the `ZTRUNC_entry` function to perform the runtime processing.

```
#define  COMMAND   "ZTRUNC"
#define  ENTRY     ZTRUNC_entry
int  __stdcall   ENTRY(pib **plib);

extern "C"  __declspec(dllexport)  int  __stdcall
    ModuleInstall(void  *hModule)
{ return (CommandInstall(hModule, COMMAND, ENTRY, NULL)); }
```

When the COMMAND and ENTRY macros are specified in this manner, the coding of the registration function is especially easy: it never changes! Simply cut and paste

from any coding example. Adjust the COMMAND and ENTRY macros appropriately, and leave the rest. If the function prototype is coded manually, be sure to preserve the `__declspec(dllexport)` and `__stdcall` qualifiers as shown, or the compiler will not generate correct linkages.

Task Parameters

When the DAPL system begins to execute a processing command from a command module, the DAPL system will dispatch to the specified entry point. As the task begins to run, it must first obtain access to the parameter information specified in the DAPL processing configuration. The parameter information identifies the data sources and destinations.

The entry function is called by the DAPL system with a single parameter, called a parameter information block, or PIB. If the DAPL system invokes multiple instances of this command for different processing tasks, each instance will receive a different PIB and will therefore operate on different data elements.

Unfortunately, the PIB is a handle that is not directly meaningful either to the task dispatcher or to the command code. To obtain access to the parameters, a special system function must be called. This function is **`param_process`** .

`param_process` is illustrated in the following example:

```
int  XCMD_entry (PIB **plib)
{
  void **argv;
  int argc;
  argv = param_process (plib, &argc, 2, 2,
        T_PIPE_W, T_VAR_W);
. . .
}
```

For this command called XCMD, the first parameter to **`param_process`** is `plib`, the `PIB` provided by the `DAPL` system. The **`param_process`** function sets the value of the second parameter variable, an integer `argc`, to the number of parameters in the actual parameter list for the task. The next two parameters respectively specify the minimum and maximum numbers of parameters that the custom task accepts. If the two numbers are the same, the number of parameters is fixed. If the second number is larger, the number of parameters can vary within a range. The remaining parameters are flags that specify the acceptable types for each of the task's parameters.

The pointer list returned by function **param_process** is accessible to the task code. The pointer list is similar to the argv parameter of a Standard C main function, except that here the pointers point to various data types, not just text strings. Pointers to data elements such as PIPE and VARIABLE can be in the list.

```
argv[0]      pointer to name of command
argv[1]      pointer to parameter 1
argv[2]      pointer to parameter 2
  etc.
```

While preparing the parameter access list, the **param_process** function verifies the parameter types. The XCMD example above specifies the data type flags T_PIPE_W and T_VAR_W, so the corresponding pointers would be a pointer to a PIPE and a pointer to a VARIABLE respectively.

If the parameter list is invalid, the DAPL system issues a diagnostic message and suspends the task at this point. For example, suppose that a DAPL configuration incorrectly specifies two XCMD tasks. The XCMD command expects a PIPE parameter and a VARIABLE parameter. Instead, it receives a single pipe parameter in one case, two variables in the other.

```
PIPES P2
VARIABLE V1,V2
PDEF A
  XCMD (P1)
  XCMD (V1,V2)
END
```

When the **param_process** is called for the improperly configured tasks, it generates the following error messages:

```
*** ERROR 1215: XCMD - too few parameters
*** ERROR 1214: XCMD - parameter 1 -
   'V1' should not be a word variable
```

Each error message identifies the command detecting the error, and provides additional information for diagnosing the problem. The task is terminated automatically after issuing the diagnostic message.

Accessing Parameters

Various techniques that can be used to extract the pointers from task parameter list and associate them with data types. One way is to declare auto variables of the

appropriate type, extract each pointer from the list, and apply the appropriate type casts to assign the pointer or associated value. Another way is to cast the list pointer to the Standard C Library type `va_list` and use the C `va_arg` macro to extract pointer values and apply the appropriate type casts. A third way is to specify the items from the pointer list as parameters to an auxiliary function, which then interprets these as pointers of the appropriate type.

The following table lists the DAPL elements, the C data types which correspond to the DAPL elements, and the type flags used by **`param_process`**.

DAPL Type	C Parameter Type	Type Flag
byte pipe	`PIPE *`	`T_PIPE_B`
word pipe	`PIPE *`	`T_PIPE_W`
long pipe	`PIPE *`	`T_PIPE_L`
float pipe	`PIPE *`	`T_PIPE_F`
double pipe	`PIPE *`	`T_PIPE_D`
trigger	`TRIGGER *`	`T_TRIGGER`
word vector	`VECTOR *`	`T_VECTOR_W`
long vector	`VECTOR *`	`T_VECTOR_L`
float vector	`VECTOR *`	`T_VECTOR_F`
double vector	`VECTOR *`	`T_VECTOR_D`
word constant	`short int const *`	`T_CONST_W`
long constant	`long int const  *`	`T_CONST_L`
float constant	`float const *`	`T_CONST_F`
double constant	`double const *`	`T_CONST_D`
word variable	`short int volatile *`	`T_VAR_W`
long variable	`long int volatile *`	`T_VAR_L`
float variable	`float volatile *`	`T_VAR_F`
double variable	`double volatile *`	`T_VAR_D`
region flag	`int const *`	`T_RFLAG`
string	`char const *`	`T_STR`

`PIPE`, `TRIGGER`, and `VECTOR` types are defined when the file `DTD.H` is included. These are handle types, not directly meaningful to the command code, but system functions can use them to locate and access the desired information.

A region flag is a special enumeration that can take one of two values, `R_INSIDE` or `R_OUTSIDE`. These constants are defined when the file `DTD.H` is included. A region flag is always followed by two scalar values, either variables or constants, which define the lower and upper limits of an interval.

A DAPL string is a pointer to a character array. The character array has the same organization as C strings, terminating with a NUL character. DAPL strings are defined using the DAPL system `STRING` command. The contents of DAPL strings must not be modified by custom commands.

Assuming that the `argv` variable has been assigned a value by the `param_process` function as previously described, code sequences such as the following can be used to obtain access to the parameter data.

```
// Pipe data types
PIPE * pPipe = (PIPE *) argv[1];

// Constant data types.
short int const  iConst = *(short int const *) argv[2];
double const  dDouble =   *(double const *) argv[3];

// Variable data types
long int volatile  *plVar = (long int volatile *) argv[4];
float volatile  *pfVar = (float volatile *) argv[5];

// Triggers
TRIGGER  *pTrig = (TRIGGER *) argv[6];

// Vectors
VECTOR  *pVect = (VECTOR *) argv[7];

// Region
short int const  eRegion = *(short int const *) argv[8];

// String
char const * pText = (char const *) argv[9];
```

It is important always to use the `const` or `volatile` qualifiers shown in the preceding table and examples, otherwise compiler code optimizers may make invalid assumptions that result in improper behaviors. For example, suppose that a variable is accessed by means of a pointer. If the `volatile` keyword is omitted, the compiler might observe that the value of the variable is read multiple times but never written; consequently, it might fetch the value of the variable one time, put that value in a register, and then never go back to the variable again. The value of the variable could be changed by another task, but that change would never be observed.

Sometimes a DAPL variable is used to establish initial values when a task starts. In this case, it is convenient to fetch the value of the DAPL variable once, assigning it to a local work variable. In other cases, when it is important to detect changes in the

variable value, it is essential to retain the pointer value and access the shared value through the pointer at each access. For example, suppose that `vlimit` is a pointer to a DAPL variable, and `limit` is a local integer variable:

```
limit = *vlimit;
while (1) {
...
if (limit>10) {
... /* value of limit never changes */
}
if (*vlimit>10) {
... /* value of *vlimit may change */
}
} /* end while */
```

Integer data types are used so often that they have special `typedef` declarations that combine the data types with the `const` or `volatile` qualifiers.

```
typedef const short int     CONSTANT;
typedef const long int      LCONSTANT;
typedef volatile short int  VAR;
typedef volatile long int   LVAR;
```

Auxiliary Functions

An unusual but sometimes useful technique for structuring a custom command and organizing parameters is calling an *auxiliary function*. The types of parameters are defined in the auxiliary function prototype. The argument list for calling the auxiliary function is used to assign a type to each of the extracted task parameters, rather than storing the parameters in declared auto variables. The effect is roughly the same. This technique is particularly useful for simpler custom commands having a minimum of parameter checking requirements.

The example command function below calls the auxiliary function `pval` after checking parameters.

```
void pval (PIPE *p, VAR *v);

void main (PIB **plib)
{
  void **argv;
  int argc;
  argv = param_process (plib, &argc, 2, 2,
                 T_PIPE_W, T_VAR_W);
  pval ((PIPE *) argv[1], (VAR *) argv[2]);
}

void pval (PIPE *p, VAR *v)
{
/* perform processing here ... */
}
```

Advanced Parameter Checking

Some commands permit optional parameters or several different combinations of parameter types. An example of this is the `COPY` command provided by the DAPL system. The `COPY` command will take data from a pipe and copy that data into a list of pipes. Up to thirty two pipes can receive the copies, hence the parameter list can range from two to thirty-three total parameters. The function **`param_process`** allows the specification of a minimum and a maximum number of allowed parameters.

The **`param_process`** function can allow more than one parameter type to appear in any given position. The data type codes for all data types that could possibly be valid are combined using the C bitwise “or” operation. The `COPY` can accept any type of data from its input pipe, so the combined code for its input source pipe is :

```
T_PIPE_W | T_PIPE_L | T_PIPE_F | T_PIPE_D
```

As a more interesting example, consider the parameter list for a task called `XCOM` that has both a variable number of parameters and a mix of data types, subject to the following requirements.

- accept from two to four parameters
- the first parameter must be a vector
- the next parameter can be a word or long integer variable
- the next parameter can be a trigger or a word constant
- the last parameter in the list must always be a float pipe

Consequently, the floating point pipe can appear in the second position, or the third, or the fourth, depending on the length of the task's parameter list. The task parameters can be checked as follows:

```
argv = param_process (plib, &argc, 2, 4,
    T_VECTOR,
    T_VAR_W | T_VAR_L | T_PIPE_F,
    T_TRIGGER | T_CONST_W | T_PIPE_F,
    T_PIPE_F);
```

After parameter processing, a command can perform additional checks. In the above example, suppose that the actual parameter list has three parameters. In this case, the second parameter must be a variable and the third parameter (the last) must be a float pipe. However, **param_process** would not detect the problem if it found a word constant in the third position.

The **param_type** function is useful for analyzing these situations, and the function **param_error_msg** is useful for diagnosing them.

The **param_type** function needs to access the DAPL system's parameter type information, so it uses the `plib` parameter, similar in this respect to the `param_process` function. It reports the specific type code for the task parameter as a return value. For the example above, the problem is to determine whether the last parameter is a floating point pipe.

```
if (param_type(plib,argc)!= T_PIPE_F)
  param_error_msg(pe_TypeInconsistent,argc);
```

The **param_error_msg** function generates a message in the following form, and then terminates the task.

```
*** ERROR 1236: XCOM - parameter 3 - type inconsistent
```

There are several error message codes available, and these will cover almost all situations. (If they do not, use other message-formatting functions to build your own error message texts.) The error codes are listed in the `DTDCNSTS.H` file and are included automatically by the `DTD.H` file.

```
enum ParamErrors {
pe_GeneralError,pe_LengthInconsistent,pe_SizeInconsistent,
pe_TypeInconsistent,pe_ValueInconsistent,pe_ValueOutOfRange,
pe_ValueNotAllowed,pe_OptionNotAllowed,pe_ParamMissing,
pe_ExtraParam, pe_ParamType };
```

Functions **param_process**, **param_error**, and **param_error_msg** are sensitive to the setting of the ERRORQ option in DAPL. If ERRORQ is on and an error is detected, both functions suppress error message printing and set the value of ERRORQ to a nonzero error code.

Tasks often use constant integer parameter values in their parameter lists. There is a hazard to avoid if the values are large. The DAPL system doesn't know what data type were intended when it sees a numeric constant value. It is clear enough that the data type should be a long value if the value is too large to represent as a 16-bit word, but what data type should it assume if the value can be represented as a 16-bit word? The DAPL system chooses to call it a WORD type. Consequently, an explicit numeric parameter value could be a word constant or a long constant, depending on its value. The following code uses the function **param_type** to determine how to fetch the value, and then assigns the value to a long integer variable val.

```
if (param_type(plib,i) == T_CONST_W)
    val = *(short int *) argv[i];
else
    val = *(long int *) argv[i];
```

Vectors

Vectors defined by the DAPL command VECTOR can contain data of type short int, long int, float, or double. A vector defined by DAPL is structured data, and the DAPL system provides special means for determining the properties of the data and accessing the data array.

A task parameter for a vector structure has type VECTOR. A vector parameter is generic in the same manner that a parameter for a PIPE structure is generic. The properties of the VECTOR structure can be tested during parameter processing by the param_process function to verify that the correct data type is present, using type codes T_VECTOR_W, T_VECTOR_L, T_VECTOR_F or T_VECTOR_D. Parameter information can be extracted from a task parameter list and assigned to a VECTOR variable.

The following code checks a task parameter list for a vector containing double precision values, and extracts the vector parameter to a local variable:

```
VECTOR * vect;
argv = param_process(plib, &argc, 1, 1, T_VECTOR_D);
vect = (VECTOR *) argv[1];
```

Once the parameter has been extracted from the parameter list, special functions can be used to determine properties of the vector data. The functions for evaluating vector properties are:

vector_length	determine the number of items in the vector
vector_width	determine the storage size for each vector item
vector_type	determine the data type code for the stored items
vector_start	obtain a pointer to the first item

Length and storage location information are available only using these special functions. The data width (`sizeof` one element) and vector type can be derived from task parameter information, but sometimes the special functions are more convenient.

For example, suppose that a custom command can accept a vector with either `short` or `long int` data. The following code example defers the test of vector type to a later part of the program:

```
VECTOR * vect;
int   vect_len;
argv = param_process(plib, &argc, 1, 1, T_VECTOR_W |
T_VECTOR_L);
vect = (VECTOR *) argv[1];
...
vect_len = vector_length(vect);
if ( vector_type(vect) == T_VECTOR_W )
        process vect_len items as short int ;
else
        process vect_len items as long int ;
```

Continuing this same example, the amount of storage required to contain the vector data can be computed as follows:

```
int  vect_size;
vect_size = vector_length(vect) * vector_width(vect) ;
```

Contents of a `VECTOR` as defined in the DAPL configuration are available to multiple tasks, and should not be altered. One way to protect against accidentally changing vector data is to qualify the contents as `const` so that the compiler will complain if there is any inadvertent attempt to alter the values. Continuing the example, a pointer to the array of `long int` data values might be constructed using the following code:

```
LCONSTANT  * vect_data;
vect_data = (LCONSTANT *) vector_start(vect);
```

Initializations and Allocations

Initializations must be performed after a custom command has extracted and checked its parameters. Most of these initializations are straightforward. All of them are important.

The most important initializations are for `PIPE`, `TRIGGER` and `PBUF` structures. More information will be provided about these structures later. For now, the essential point is making sure that every data structure is initialized before performing the real time processing.

Each initialization function returns a value. Typically, this is a pointer or a handle. The handle or pointer must be stored so that it can be used during later operations.

Each pipe must be opened before performing pipe I/O. The function **`pipe_open`** opens a pipe. This function accepts a pipe pointer and a flag indicating whether the pipe will be read or written. The flag `P_READ` specifies input, and the flag `P_WRITE` specifies output. Almost all custom commands use the **`pipe_open`** function.

The following example shows an initialization of a pipe for reading items individually:

```
PIPE *p;
...
pipe_open (p, P_READ);
```

A custom command that operates on blocks of data rather than individual values needs an additional data management structure for each pipe. First, the **`pipe_open`** operation described above must be performed. Then, a structure called a pipe buffer control block or `PBUF` must be allocated and initialized. This second step is performed using the function **`pbuf_open`**, which returns a handle for a `PBUF` structure. Each `PBUF` structure is uniquely associated with the one task that allocates it. The `PBUF` structure contains information about the number of data values currently available, the location of the storage buffer for these data, the minimum number of values to be placed into the buffer, and the maximum number of values to be placed into the buffer. **`pbuf_open`** supplies some default values, which will be satisfactory in most cases.

The following shows a typical initialization for reading blocks of up to 200 items. First, the pipe is opened for reading data, and then the PBUF is set up for reading the data in blocks.

```
#define BUF_SIZE 200
PIPE *p1;
PBUF *inbuf;
...
pipe_open (p1, P_READ);
inbuf = pbuf_open (p1, BUF_SIZE);
...
```

Tasks that issue or receive software trigger assertions must initialize `TRIGGER` structures using the special function **`trigger_open`**. This function is the same as **`pipe_open`**, except that it requires a trigger parameter rather than a pipe parameter. See Chapter 7 for more information about software trigger initialization.

Sometimes a task requires relatively large blocks of data storage. Such a task can use the function **`ralloc`** to allocate a working memory area from the Data Acquisition Processor bulk memory. The function **`ralloc`** accepts the number of bytes of memory to allocate, and returns a pointer to the allocated memory. If insufficient memory is available, an error message is printed and the task halts. (This rarely occurs in practice, because there usually is not any data stored in the Data Acquisition Processor before tasks are started.)

Specialized initializations are required for some specialized processing, such as PID control or DSP computations. These are discussed in detail in the chapters covering these specialty areas.

Pipe Read and Write Routines

Once a task begins its real time processing loop, it typically receives data from pipes, and places its results into pipes, repeating this cycle.

Some custom commands operate on small amounts of data. They get the data, perform their operations quickly, then quietly wait for the next data to arrive. Other data acquisition tasks need to process large amounts of data efficiently. The pipe operations described in this section apply primarily to the case of small data volumes. However, the principles discussed in this section apply to both cases. Be sure to have a good understanding of this section before covering the section on blocked pipe operations later in this chapter.

The system functions for accessing pipes are **pipe_value_get** and **pipe_value_put**. The function **pipe_value_get** reads one data value from a DAPL pipe. If this task has no remaining data to read when **pipe_value_get** is called, the calling task goes to sleep until the pipe contains data. To place a value into a pipe, the **pipe_value_put** function is used. If the pipe is full and cannot accept more data, the task goes to sleep until pipe capacity becomes available. Some control over pipe behavior is possible using the DAPL system PIPE command when the pipe is defined.

A simple copy operation can use a get and a put operation in sequence.

```
GENERIC_SCALAR   value;
pipe_value_get(pipei1,&value);
pipe_value_put(pipei2,&value);
```

This example introduces a new data type called a GENERIC_SCALAR. To cope with the variety of data types and sizes in the DAPL system, the **pipe_value_put** and **pipe_value_get** functions use a GENERIC_SCALAR as an intermediate storage element that can contain data of arbitrary type. To extract the data and interpret it as the appropriate data type, qualify the generic scalar name:

```
value._word
value._long
value._i16
value._i32
value._float
value._double
```

In some ways the GENERIC_SCALAR is very convenient because it works regardless of data type. In some ways it is dangerous, because if you fetch data of one type, but then extract it from the GENERIC_SCALAR as data of another type, no logical type conversions are performed. You must supply the cast. For example, if the command copies data but converts the representation to float first,

```
GENERIC_SCALAR   value;
pipe_value_get(pipei,&value);
value._float = (float)(value._i16);
pipe_value_put(pipef,&value);
```

The functions **pipe_value_put** and **pipe_value_get** will always work, but for the special cases of fixed-point data, the older style **pipe_get** and **pipe_put** functions can be substituted. These functions operate upon long integer types, but they are aware of pipe data type and will automatically sign-extend short integer values as

necessary. The compiler might complain about “loss of precision” even though the operation is actually safe.

```
awordval = pipe_get(awordpipe);
pipe_put(apipe,awordval);
```

We now have enough information to build a complete processing command. This command PVAL monitors a data stream and makes sure that a shared variable v is updated to the most current value. The module registration portion of the code is omitted here, but you can see the complete source in PVALM.CPP. This code is found in your `DTD32\Examples` folder.

```
// PVAL (p,v)
//    - keeps variable 'v' updated to the most recent
//        value in pipe 'p'

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE  * in_pipe;
    VAR   * shared;

  // Storage for processing
    GENERIC_SCALAR  pipe_value;

  // Access parameters
    argv = param_process (plib, &argc, 2, 2,
      T_PIPE_W, T_VAR_W);
    in_pipe = (PIPE *) argv[1];
    shared  = (VAR  *) argv[2];

  // Perform initializations
    pipe_open (in_pipe, P_READ);

  // Begin continuous processing
    while (1)
    {
        pipe_value_get(in_pipe,&pipe_value);
        *shared = pipe_value._i16;
    }
    return 0;
}
```

Many custom commands can be implemented using only the four system routines: **`param_process`**, **`pipe_open`**, **`pipe_value_put`** and **`pipe_value_get`**. However, more efficient processing is usually possible using the blocked pipe operations described later in this chapter.

Application Examples Using Pipes

This section shows a few more examples of processing commands, along with some additional tips and tricks.

The COPY2M module implements a simplified version of the `COPY` command provided by the DAPL system. It reads integer data from an input pipe and puts copies of the data into two output pipes. It illustrates the “auxiliary function” style.

```
// COPY2 (p1, p2, p3)
//     - places copies of data from pipe 'p1' into
//         pipes 'p2' and 'p3'
//
// Define an auxilliary function for task parameters
void copy2 (PIPE *p1, PIPE *p2, PIPE *p3);

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W, T_PIPE_W, T_PIPE_W);
    copy2 ((PIPE *) argv[1], (PIPE *) argv[2],
           (PIPE *) argv[3]);
    return 0;
}

// Auxilliary function never returns.

void copy2 (PIPE *p1, PIPE *p2, PIPE *p3)
{
  // Storage for processing
    long int d;

  // Perform initializations
    pipe_open (p1, P_READ);
    pipe_open (p2, P_WRITE);
    pipe_open (p3, P_WRITE);

  // Begin continuous processing
    while (1) {
        d = pipe_get (p1);
        pipe_put (p2, d);
        pipe_put (p3, d);
    }
}
```

After compiling and downloading the file as described in the *Compiling and Downloading* chapter, the DAPL system `DISPLAY` command can be used to verify that `COPY2` command is available.

```
#display symbols
  COPY2     type=command
  COPY2M    type=module
#
#display commands
  COPY2     stacksize=4096
#
```

Note: A `RESET` command does not erase command modules. `RESET` can be used between DAPL applications without requiring reloading of custom command modules. The `ERASE` command can erase a module provided that none of the commands that it defines are referenced in any existing processing configuration. The DAP server application is preferred for unloading command modules, otherwise the changes resulting from ERASE can be reversed the next time the DAP is started.

A useful way of testing commands in custom modules is to define a processing procedure that uses the command, then present the task with test data using the DAPL system `FILL` command. To test `COPY2` using Microstar Laboratories DAPview program, enter the following DAPL commands in the interactive DAP window:

```
#pipes p1,p2,p3
#pdef a
  >copy2 (p1,p2,p3)
  >format (p2,p3)
  >end
#start a
#fill p1 4 5 6 7
```

The `FILL` command places data into pipe `P1`. If the custom command is working correctly, the custom command places copies of the data into pipes `P2` and `P3`, which causes `FORMAT` to print:

```
4      4
5      5
6      6
7      7
```

The next example computes a running average over a stream of data values. In addition to the pipe initialization, input, and output functions, this command uses the function **`ralloc`** to obtain a region of temporary storage for data that has been read.

The running average is defined as the sum of the last n data values divided by n. The memory array is used as a circular buffer to store the last n data values.

```
// RAVE (p1, n, p2)
//   - compute the running average of 'n' points
//        from pipe 'p1'
//   - put results into pipe 'p2'
//
int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void  **argv;
    int   argc;
    PIPE  * in_pipe;
    PIPE  * out_pipe;
    short int  N;

  // Storage for processing
    int   i;
    short int   *data, *start_data;
    long int   sum;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W, T_CONST_W, T_PIPE_W);
    in_pipe = (PIPE *) (argv[1]);
    N       = *(short int const *)(argv[2]);
    out_pipe= (PIPE *) (argv[3]);

  // Perform initializations
    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);
    data = start_data = (short int *)
        ralloc(N * sizeof(short int));

  // Special first pass: initialize sum, save data in buffer
    sum = 0L;
    for (i=0; i<N; i++)
    {
        *data = (short int) pipe_get(in_pipe);
        sum += *data;
        data++;
    }
    pipe_put(out_pipe, sum/N);


  // Begin continuous processing
```

```
        while (1)
        {
           data = start_data;
           for (i=N; i--; /*NIL*/)
           {
               sum -= *data;
               *data = (short int) pipe_get(in_pipe);
               sum += *data;
               ++data;
               pipe_put(out_pipe, sum/N);
           }
        }
        return 0;
    }
```

Text Transfer

Several system routines provide text string formatting functions. The function **`printf`** formats and prints a series of characters and values. The output of **`printf`** is sent to the output pipe `$SYSOUT`. The function **`fprintf`** provides the same formatting capabilities as **`printf`** except that the resulting string is sent to a specified byte output pipe. The function **`sprintf`** performs similar formatting, storing the result in a string rather than writing to a pipe. The format conversions are compatible with the Standard C Library. For more information about format conversions, see the descriptions of function **`printf`** in the compiler runtime library manual and Chapter 12 in this manual.

The PRT command reads data from a word pipe and prints the data values with text:

```
// PRT (p1)
//    - reads data from pipe 'p1'
//    - formats data into a string, adding text
//    - sends the string to the PC
//    - output data to pipe 'p2'
int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE  * in_pipe;

  // Access parameters
    argv = param_process (plib, &argc, 1, 1, T_PIPE_W);
    in_pipe = (PIPE *)(argv[1]);

  // Perform initializations
    pipe_open(in_pipe, P_READ);

  // Begin continuous processing
    while (1)
    {
        printf ("Data = %d \n", pipe_get(in_pipe));
    }
    return 0;
}
```

Numbers can be extracted from a message text using the function **sscanf**. This function can be dangerous, so verify that the conversion codes and the data pointers in the parameter list match exactly.

Blocked Pipe Operations

Each pipe ‘get’ or ‘put’ operation cause some operating system overhead. This overhead limits the maximum rate at which data values can be transferred into and out of pipes. Blocked pipe operations increase the pipe input/output rate by operating on blocks of data. The overhead is still there, but it is relatively small compared to the processing required for an entire block.

A blocked get operation reads a number of data values from a pipe into a memory array. A blocked put operation writes data from a memory array into a pipe. Blocked operations using large blocks are typically ten to twenty times faster than non-blocked operations. In most cases, the most efficient processing strategy is to fetch whatever

data are available, up to some maximum amount, process that block, and then repeat for the next block of data. The tradeoff for this efficiency is delay. It takes some time to collect the samples that make up the data block, so the latency increases roughly in proportion to the block length.

As discussed in the section on command initialization, the pipe to be accessed using a blocked operation must use the function **pipe_open** to initialize the pipe, and then function **pbuf_open** to allocate and initialize a PBUF structure for the opened pipe. The function **pbuf_open** can also allocate a storage array of the desired size, automatically, and install it in the PBUF structure.

When real-time processing begins, the function **pbuf_get** reads data from the associated pipe into the storage array of the task's pipe buffer, as in the following example:

```
PIPE *p1;
PBUF *inbuf;
.
.
pipe_open (p1, P_READ);
inbuf = pbuf_open (p1, BUF_SIZE);
.
.
pbuf_get (inbuf);
/* process data array values here... */
```

In the default configuration, the function **pbuf_get** reads as much data as it can, up to the maximum capacity of the PBUF memory. To determine how many samples were obtained, use the function **pbuf_get_cnt**.

The function **pbuf_put** writes data from a task's pipe buffer into the associated output pipe. The command code must first place the data to be written into the storage array, and call the function **pbuf_set_cnt** to specify how many items are present. The following C code writes a block of data to a pipe:

```
PIPE *p2;
PBUF *outbuf;
int  item_count;
...

pipe_open (p2, P_WRITE);
outbuf = pbuf_open (p2, BUF_SIZE);
...

/* process data array values here ... */
pbuf_set_cnt(outbuf,item_count);
pbuf_put (outbuf);
```

The operations of getting data and then finding out how many, and setting the number and then sending that many, typically occur in these combinations. These operations can be combined for convenience and simplicity in the following manner.

```
/* process data array values here ... */
item_count = pbuf_get(inbuf);
...
pbuf_put_set_cnt(outbuf,item_count);
```

The `inbuf` and `outbuf` variables are handles that are used by only one task, so that task is free to reconfigure it at any time. To do this, the Developer's Toolkit for DAPL provides a set of access functions. We have already seen two of these operations.

- The function **`pbuf_get_cnt`** reports the number of data values present in the pipe buffer storage array. This function is sometimes useful after data has been fetched into the storage array by the function **`pbuf_get`**. It is usually more convenient to use the count that function **`pbuf_get`** returns.
- The function **`pbuf_set_cnt`** specifies the number of data values that have been placed into the storage array. This is function is used before calling the function **`pbuf_put`**. It is usually more convenient to use function **`pbuf_put_set_cnt`**.

The storage area used for the pipe operations can be switched.

- The function **`pbuf_get_data_ptr`** returns a pointer to the data array assigned to the PBUF structure.
- The function **`pbuf_set_data_ptr`** assigns a data array to a PBUF structure.

Minimum and maximum numbers can be established. These values are most useful when requesting a number of samples to be read from a pipe.

- The function **`pbuf_get_max_cnt`** reports the maximum number of values that the function **`pbuf_get`** is allowed to fetch from a pipe.

- The function **pbuf_set_max_cnt** establishes the maximum number of values that the function **pbuf_get** is allowed to fetch from a pipe. This function is used mostly for initialization.
- The function **pbuf_get_min_cnt** reports the minimum number of values that must be read into pipe buffer storage before the **pbuf_get** function returns to the caller.
- The function **pbuf_set_min_cnt** establishes the minimum number of values that must be read into pipe buffer storage before the **pbuf_get** function returns to the caller. This function is used mostly for initialization.

The data minimum and maximum count bounds determine how many values should be read into the data array when the function **pbuf_get** is called. If the input pipe contains less than the minimum number of data values, the function **pbuf_get** suspends the task and does not return until sufficient data values are available from the input pipe. The function **pbuf_get** will not transfer more than the specified maximum number of values from the input pipe to the PBUF data storage array. After completing the transfer, the function **pbuf_get** records the current number of samples stored in the storage array and also reports this number to the caller. To request an exact number of samples, specify the same block size when calling **pbuf_set_max_cnt** and **pbuf_set_min_cnt**. The specified number of samples must not be negative and must not be greater than the size of the buffer storage area.

The **pbuf_put** operation takes from the PBUF structure data array the number of values specified by the current data count, placing the values into the associated pipe. The maximum and minimum sample counts are ignored. After copying the values from the PBUF storage buffer into the pipe, the **pbuf_put** operation sets the PBUF data count to zero. The **pbuf_put_set_cnt** function is the same, except that the current data count is set first before performing the **pbuf_put** operation.

Both the minimum and the maximum data count fields are initialized by **pbuf_open**, but may be reprogrammed after **pbuf_open** is called. The default values are a minimum of 1 sample, and a maximum equal to the total storage length specified.

The following C code illustrates reading a block of data values using a **pbuf_get** operation, and referencing individual data values in the PBUF storage area. A pointer to this storage is obtained using the **pbuf_get_data_ptr** function:

```
int count;
short int *p;
...

count = pbuf_get (inbuf);

p = pbuf_get_data_ptr(inbuf);
for (i = 0; i < count; I++)
printf("%d\n", p[i]);
```

A technique that is sometimes useful with blocked pipe input is a *non-blocking* fetch. In this case, the term *non-blocking* means that the task execution is not suspended if no data are available. This behavior is obtained by setting the **pbuf_get_min_cnt** field to zero. If the **pbuf_get** function does not have any data available, it observes that the number available — zero — satisfies the minimum count criterion. It returns and reports a sample count of zero. *Be sure to test for the case that the data count is zero upon return!* Attempting to process data that does not exist can cause all kinds of unexpected and quite wrong behaviors. This technique should be used when a custom command must coordinate among several internal processes, and cannot afford to delay other parts of the processing while waiting for data to arrive.

The second technique is buffer storage sharing. This is done by allocating a PBUF with zero size. This means that there is no storage array associated with this PBUF — at least not at first. The initialization is completed later by using the function **pbuf_get_data_ptr** to copy the storage pointer from another PBUF, and using the function **pbuf_set_data_ptr** to assign that storage pointer to the PBUF that does not have its own storage. The maximum and minimum data counts must then be adjusted accordingly, using the **pbuf_set_max_cnt** and **pbuf_set_min_cnt** functions to make the buffer size information in the PBUF consistent. This technique allows data to be modified in place, using a single buffer, achieving very good processor efficiency.

Blocked pipe input and output are illustrated by the BCOPY2 custom command. The BCOPY2 command copies the contents of an input pipe into two output pipes. BCOPY2 is functionally equivalent to the unblocked COPY2 command, illustrated earlier in the chapter, but it transfers data much faster. Thanks to the buffer-sharing technique, the block input operation puts the data where it is needed for the block output operation, and no processing is required!

```
// BCOPY2(p1, p2, p3)
//   - places copies of data from pipe 'p1' into
//       pipes 'p2' and 'p3'

#define BUF_SIZE 128

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;

  // Storage for parameters
    PIPE *p1, *p2, *p3;

  // Storage for processing
    PBUF *inbuf, *outbuf1, *outbuf2;
    void *databuf;
    int  bufcount;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W, T_PIPE_W, T_PIPE_W);
    p1 = (PIPE *)(argv[1]);
    p2 = (PIPE *)(argv[2]);
    p3 = (PIPE *)(argv[3]);

  // Perform initializations
    pipe_open (p1, P_READ);
    pipe_open (p2, P_WRITE);
    pipe_open (p3, P_WRITE);

    /* Allocate only one storage area */
    inbuf   = pbuf_open (p1, BUF_SIZE);
    outbuf1 = pbuf_open (p2, 0);
    outbuf2 = pbuf_open (p3, 0);

    /* Share input and output storage */
    databuf  = pbuf_get_data_ptr(inbuf);
    pbuf_set_data_ptr(outbuf1,databuf);
    pbuf_set_data_ptr(outbuf2,databuf);
    pbuf_set_max_cnt(outbuf1,BUF_SIZE);
    pbuf_set_max_cnt(outbuf2,BUF_SIZE);
```

```
    // Begin continuous processing
      while (1)
      {
          /* No data transfer necessary for output copies */
          bufcount = pbuf_get(inbuf);
          pbuf_put_set_cnt(outbuf1,bufcount);
          pbuf_put_set_cnt(outbuf2,bufcount);
      }
      return 0;
  }
```

Another example of blocked pipe operations is a more efficient version of the `ZTRUNC` command that was introduced as *the typical processing command* in the *Overview* chapter. This variation reads and processes data in blocks rather than one value at a time, applying a fixed lower data bound of zero. Notice that the defaults, minimum data count one and maximum data count `BUF_SIZE`, are used both for the input and the output pipes.

```
// BZTRUNC (p1, p2)
//    - read data from pipe 'p1'
//    - truncate any numbers below 0
//    - output data to pipe 'p2'
//

#define  BUF_SIZE 128

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE  * in_pipe, * out_pipe;

  // Storage for processing
    PBUF  * inbuf,  * outbuf;
    short int   * datain, * dataout;
    int   datacount;
    int   i, tmp;

  // Access parameters
    argv = param_process (plib, &argc, 2, 2,
        T_PIPE_W, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];
    out_pipe= (PIPE *) argv[2];

  // Perform initializations
    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);
    inbuf   = pbuf_open (in_pipe, BUF_SIZE);
    outbuf  = pbuf_open (out_pipe, BUF_SIZE);
    datain  = (short int *) pbuf_get_data_ptr(inbuf);
    dataout = (short int *) pbuf_get_data_ptr(outbuf);

  // Begin continuous processing
    while (1)
    {
        datacount = pbuf_get(inbuf);

        for (i=0; i<datacount; i++)
        {
            tmp = datain[i];
```

```
            if (tmp < 0)  dataout[i] = 0;
            else          dataout[i] = tmp;
        }

        pbuf_set_cnt(outbuf,datacount);
        pbuf_put(outbuf);
    }
    return 0;
}
```

Other Pipe Functions

The function **pipe_num_complete** accepts a pipe pointer and returns the number of data values currently stored in the pipe, up to a specified limit. The function **pipe_num** is a useful alternative for determining whether some data are available, when an accurate count is not required.

The function **pipe_width** accepts a pipe pointer and returns the size of one element from the pipe, in bytes. A `word` pipe has a width of two bytes, a `long` pipe or a `float` pipe has a width of four bytes, and a `double` pipe has a width of 8 bytes.

The function **pipe_rem** efficiently removes data from a pipe. This is sometimes useful when it is determined that the pipe contains data that do not require processing. This function normally is not needed at the end of processing, since the DAPL `STOP` command automatically empties all system pipes.

Task Control

When a task is executing, the task is competing for CPU time with all other active tasks. When the function **task_switch** is called, the processor temporarily suspends the current task. Other active tasks are given CPU time before the CPU returns to the original task. If a task is waiting for an event, the **task_switch** system call should be used to release the CPU so that other tasks can be served.

Suppose, for example, one task sets the value of a global variable and another task waits for the global variable to change to a nonzero value. (This technique can be used to implement inter-task message passing via global variables.)

If the variable pointer is `v`, one version of the message receiving code is:

```
while (!*v)  /* do nothing */ ;
```

This code is inefficient. The task wastes CPU time waiting for the value of the variable to change, but the variable value cannot change while this task is executing the loop. A better solution is for the task to release the CPU to other tasks before rechecking the value of the variable:

```
while (! *v)
task_switch() ;
```

The signaling task usually performs a **task_switch** also. After the signaling task changes the value of the variable, a task switch forces the CPU to immediately give the receiving task an opportunity to recognize the message.

Occasionally, execution of a custom task simply needs to be stopped. An inefficient way of doing this would be:

```
while (1)
;
```

A better way is to call the function **exit**; the task then is terminated and will not be scheduled to run again.

Direct Output Functions

A custom command can send a value to a DAC by calling the function **dac_out**. The first parameter specifies the DAC number (0 or 1). The second parameter specifies the data value to write to the DAC. The data value is interpreted as a 16-bit number. See the chapter 'Voltages and Integers' in the DAPL Manual for an explanation of the relationship between 16-bit numbers and analog voltages.

If external analog output expansion hardware is connected to the Data Acquisition Processor, DAC numbers greater than one may be specified in **dac_out**. DAC output expansion is enabled using the DAPL `OUTPORT` command.

Note: The function **dac_out** provides a low-latency method of updating the digital-to-analog converter in an asynchronous manner. The exact time at which this occurs depends on the time that the task runs, which is subject to the timing uncertainties of multitasking. For precise timing between DAC updates, it is recommended that a custom command write DAC data to an output channel pipe. An output procedure then can read the channel data and update the DAC at precisely-timed intervals.

Digital output lines can be controlled using function **digital_out**. The first parameter of **digital_out** specifies the port number of the on-board digital output

port. This number is zero. The second parameter specifies a 16-bit data value that is written to the digital output port.

Two additional functions, **digital_set_bit** and **digital_toggle_bit**, allow control of individual bits of the digital output port.

If external digital output expansion hardware is connected to the Data Acquisition Processor, digital port numbers greater than zero and digital bit numbers greater than sixteen may be specified by the digital output functions. Digital output expansion is enabled using the DAPL OUTPORT command.

Note: The function **digital_out** provides a low-latency method of updating the digital port lines in an asynchronous manner. The exact time at which this occurs depends on the time that the task runs, which is subject to the timing uncertainties of multitasking. For precise timing of digital output port updates, it is recommended that a custom command write digital output data to an output channel pipe. An output procedure then can read the channel data and update the digital output port synchronously.

Real Time Clock

A custom command may need to determine the current time or may need to pause for a specified period of time. The function **sys_get_time** returns the number of milliseconds since the Data Acquisition Processor was powered on. The function **task_pause** causes the current task to pause for a specified number of milliseconds.

The following custom command illustrates the use of **task_pause** to generate a one Hertz square wave at the least significant bit of the digital output port..

```
/*  SGEN
*       - generates a square wave on the digital
*          output port
*/
#include <cdapcc.h>
void gen_square (void);

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 0, 0);
    gen_square ();
}


void gen_square (void)
{
    while (1)
    {
        digital_out (0, 0);
        task_pause (500);
        digital_out (0, 1);
        task_pause (500);
    }
}
```

The real-time clock has good long-term stability and accuracy, but the frequency and jitter of a waveform generated in the manner of this example cannot be guaranteed. Depending on the number and activity level of processing tasks, there could be a variable-length delay between the time that the `task_pause` function completes its timing cycle and the time that the processing task gets it next opportunity to run and update the digital port.

7. Software Triggering Support

This chapter discusses special functions and useful programming techniques for building custom commands for software triggering.

The Developer's Toolkit for DAPL provides a set of special system routines which give access to all software triggering features of the DAPL 2000 operating system.

Most applications can use the basic triggering commands built into the DAPL 2000 operating system, and do not need extended triggering capability. For example, a simple threshold (`LIMIT`) might be adequate to determine whether something significant is present in a data stream. Other systems might need to apply a more complex analysis to identify important data. When the flexibility of a built-in DAPL command is needed, but triggering capabilities of built-in commands are not sufficient, custom triggering commands should be considered.

There is a design tradeoff between optimizing one application and building a generally useful component. Individual applications can usually apply ordinary programming techniques in a custom command to avoid software triggering. This does not necessarily make the programming task less complex. It achieves equivalent results, gaining efficiency by giving up flexibility.

Triggering is somewhat complex, because it combines:

- analysis of a data stream to recognize special events
- communication of these events between tasks
- processing of a data stream in response to the special events

Software triggering is a powerful inter-task signaling and data selection capability. Before developing custom commands that use software triggering, a review of the software triggering material in the DAPL manual is strongly recommended. Familiarity with `LIMIT` and `WAIT` commands and other triggering commands is also helpful.

Establishing the Connection

A typical trigger configuration consists of one task that asserts a trigger and one or more tasks that wait for trigger assertions. These are called the signaling task and the receiving tasks, respectively. Sometimes “asserting a trigger” is described as “writing a trigger,” because information about an event is written into a trigger structure.

Similarly, “waiting for assertions” is sometimes called “reading a trigger” because information about an event is extracted from the trigger structure.

Each task that uses a software trigger is associated directly or indirectly with a data stream. A signaling task reads and analyzes data from its data stream, and writes trigger assertion information into the trigger. A task responding to the trigger assertion reads that triggering information, and uses the information to extract the desired samples from its associated data stream.

A trigger control structure resides in the operating system area, and contains a pipe and a status field. The pipe is used to queue assertion information. The status field is used to communicate operating status among trigger readers and writers.

A custom command task first uses the **`trigger_open`** function to establish a connection with a trigger, in much the same manner that a **`pipe_open`** command is used to access a data pipe. When either a trigger receiving or trigger signaling task calls the **`trigger_open`** function, it receives a handle to a system trigger control structure. This `TRIGGER` structure is defined by a DAPL `TRIGGER` command. Though not directly accessible, the returned handle has the form of a `TRIGGER` pointer. The following shows the code to initialize two `TRIGGER` handles, one for writing and one for reading:

```
TRIGGER    *Twrite, *Tread;
...
trigger_open(Twrite,P_WRITE);
trigger_open(Tread,P_READ);
```

Using the Trigger Functions

A trigger’s pipe shares many properties with ordinary data pipes, hence, there are many similarities between trigger and pipe operations. Assertions placed into the trigger’s pipe have the form of a 32-bit unsigned number. Continuing the previous example, the following operations can be used to extract trigger assertion information from one trigger and copy it into the other:

```
unsigned long   assertion;
 ...
/* DANGER! */
assertion = trigger_get(Tread);
trigger_put(Twrite,assertion);
```

Unfortunately, this is not all of the story. The above code has subtle dangers. While it moves assertion information from one trigger to another successfully, it does not keep

the trigger status fields current. This can cause some serious complications. Fortunately, there are easy solutions.

The trigger's status field announces to the DAPL system the sample number of the most recently processed sample in the associated data stream. Updating the status indicates that the task has completed all processing associated with the corresponding sample. Because each sample can be processed only once, the status field is strictly increasing. Samples are numbered starting with sample 0. This numbering does not have a direct relationship to the sampling clock, and software triggering can operate without any active input sampling procedures.

It is essential for both trigger reading and writing tasks to keep the status field current. Writing an assertion to a trigger automatically updates the status to match the asserted sample number. For a signaling task with no new assertion, or in all cases for a receiving task, one of the following two methods can be used to update the trigger status:

```
unsigned long  new_status, increment;

/* Method 1 -- Compute a new status number explicitly */
new_status = trigger_get_status(Tread) + increment;
trigger_set_status(Tread,new_status);

/* Method 2 -- Increment the old status number */
trigger_updt_status(Tread,increment);
```

The example above shows code for a receiving task, but the code is similar for a signaling task when no events are asserted.

If a sample corresponding to a trigger event is detected, a trigger signaling task has two ways that it can signal the event:

```
unsigned long  new_event, increment;

/* Method 1 -- Assert at an explicit sample number */
new_event = trigger_get_status(Twrite) + increment;
trigger_put(Twrite,new_event);

/* Method 2 -- Increment the old status and assert*/
trigger_updt_put(Twrite,increment);
```

Note that the process of fetching the old trigger writer status, updating it, and asserting the new value is so common that these operations are combined in the function **`trigger_updt_put`**.

It should be apparent now why using the **trigger_get** function alone can be dangerous. If a trigger reader task tries to get an assertion from its trigger structure, but no assertion is present, the trigger reader task must wait. While the task is waiting, it does not update its status, and a backlog can occur in the trigger reader's associated data pipe. The data backlog can lead to inefficiencies or to a memory overflow condition.

A solution to this problem is to use the special **trigger_wait** function, which keeps the trigger reader's status current and discards unneeded data as it waits for an assertion to arrive. When **trigger_wait** returns, the next sample in the associated data pipe is the first sample corresponding to the asserted event. The following is a recommended way to detect a trigger assertion without causing a data backlog:

```
/* RECOMMENDED! */
unsigned long   assertion;
PIPE   * data_pipe;
...
assertion = trigger_wait(Tread,data_pipe,0,1);
```

There are some situations, however, when a custom command will not want to wait for an assertion to arrive. For these situations, the **trigger_get_immediate** function is an alternative to the **trigger_wait** function. Function **trigger_get_immediate** returns immediately with a value which is either the first available assertion or the most current status. To determine which value is received, an extra variable is passed to the **trigger_get_immediate** function:

```
int  assert_flag;
...
assertion = trigger_get_immediate(Tread,&assert_flag);
if (assert_flag)
    { /* process the assertion */ }
else
   { /* update status and do other processing */ }
```

To summarize, the responsibilities of a signaling task which processes data individually are:

- Call the function **trigger_open** to initialize the trigger.
- Read a data value from the associated data pipe. Check for triggering conditions.
- Assert each trigger event, placing the corresponding sample number into the trigger.
- Increment the trigger status by one for each sample scanned from the associated data pipe without an assertion. The **trigger_updt_status** function is useful for this.

The above process can be described slightly differently for the case where data values are scanned in blocks:

- Call the function **trigger_open** to initialize the trigger.
- Read blocks of data from the associated data pipe. For each block, scan through the data samples testing for triggering conditions.
- Assert each trigger event to place the corresponding sample number into the trigger.
- Update the trigger status by the number of samples remaining after the last trigger event is asserted, or by the block size if there are no assertions.

The responsibilities of a receiving task are:

- Call **trigger_open** to initialize the trigger.
- To obtain the next assertion without waiting, call **trigger_get_immediate** to receive either an assertion or a status count. Use or discard data from the associated data pipe explicitly. Update the trigger status for each item used or discarded.
- To wait for the next assertion, call **trigger_wait**. When it returns, take data values from the input pipe, and update the trigger status for each value taken.

Though dangerous, the **trigger_get** function is sometimes useful. It can be called safely if the **trigger_num** function is called first to verify that a trigger assertion is available. If an assertion is present in the trigger, **trigger_get** reads that assertion value, and does not block task execution.

Special Trigger Modes

Some triggering commands might require a trigger with a special operating mode, or that has a HOLDOFF or other important operating property. The **trigger_get_opmode** and the **trigger_get_property** functions can be used to verify that the trigger was correctly defined. See the DAPL manual for information about trigger properties and operating modes.

Triggering Command Examples

This section provides a number of programming examples, showing typical trigger signaling and receiving tasks.

The following example command, LIMIT2, is a simplified form of the LIMIT command in the DAPL operating system. LIMIT2 is a signaling task.

```
// LIMIT2 (p1, region, t1)
//     - asserts trigger 't1' when data from pipe
//       'p1' satisfies the 'region' condition

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE     * in_pipe;
    short int rflag;
    short int low;
    short int high;
    TRIGGER  * t;

  // Storage for processing
    GENERIC_SCALAR  d;

  // Access parameters
    argv = param_process (plib, &argc, 5, 5,
         T_PIPE_W, T_RFLAG, T_CONST_W, T_CONST_W,
         T_TRIGGER);
    in_pipe = (PIPE *) argv[1];
    rflag   = *(const short int *) argv[2];
    low     = *(const short int *) argv[3];
    high    = *(const short int *) argv[4];
    t       = (TRIGGER *) argv[5];

  // Perform initializations
    pipe_open (in_pipe, P_READ);
    trigger_open (t, P_WRITE);

  // Begin continuous processing
    while (1)
    {
        pipe_value_get (in_pipe,&d);
        if (rflag == R_INSIDE)
        {   /* INSIDE region */
            if   ((d._i16 >= low) && (d._i16 <= high))
                trigger_updt_put(t,1);
            else
                trigger_updt_status(t,1);
        }
        else
```

```
            {   /* OUTSIDE region */
                if  ((d._i16 < low) || (d._i16 > high))
                    trigger_updt_put(t,1);
                else
                    trigger_updt_status(t,1);
            }
        }
        return 0;
    }
```

The preceding trigger example can be modified easily to create custom commands that detect different trigger conditions. It is necessary only to change the '`if`' statements that determine whether **`trigger_updt_put`** or **`trigger_updt_status`** is called. Notice how every sample is accounted for. Either an assertion is posted, or the trigger is informed that no assertion occurs.

The next example, the `WAIT2` command, is a simplified version of the `WAIT` command which is part of the DAPL operating system. `WAIT2` is a trigger receiving command.

```
//  WAIT2 (p1, t1, n1, n2, p2)
//    - FETCH n1+n2 values from pipe p1
//        when a trigger assertion arrives in trigger t1
//    - n1 data values preceed the trigger event
//    - n2 data values are at or subsequent to the event
//    - place selected data into pipe p2

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE  * in_pipe;
    TRIGGER * t;
    short int  pretrigger;
    short int  posttrigger;
    PIPE  * out_pipe;

  // Storage for processing
    GENERIC_SCALAR  pipe_value;
    int  i;

  // Access parameters
    argv = param_process (plib, &argc, 5, 5,
        T_PIPE_W, T_TRIGGER, T_CONST_W,
        T_CONST_W, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];
    t       = (TRIGGER *) argv[2];
    pretrigger = *(short int const *) argv[3];
    posttrigger= *(short int const *) argv[4],
    out_pipe= (PIPE *) argv[5];

  // Perform initializations
    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);
    trigger_open (t,P_READ);

  // Begin continuous processing
    while (1)
    {
        trigger_wait(t, in_pipe, pretrigger, 1);
        for (i=0; i < (pretrigger+posttrigger); i++)
        {
```

```
            pipe_value_get (in_pipe,&pipe_value);
            pipe_value_put (out_pipe,&pipe_value);
        }
        trigger_updt_status (t,(pretrigger+posttrigger));
    }
    return 0;
}
```

The following example is a combination of the DAPL system's TSTAMP and FORMAT commands. TSTAMP2 waits for trigger assertions and prints the sample count of each assertion. This command is unusual because it is not directly associated with a data stream. This means it is safe to use the otherwise dangerous **trigger_get** function to suspend task execution until an assertion appears.

```
// TSTAMP2M (t)
//    - fetches trigger assertions from trigger t
//    - prints these as integer 'timestamp' values

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    TRIGGER *  trig;

  // Storage for processing
    unsigned long int  assertion;

  // Access parameters
    argv = param_process (plib, &argc, 1, 1, T_TRIGGER);
    trig = (TRIGGER *) argv[1];

  // Perform initializations
    trigger_open(trig,P_READ);

  // Begin continuous processing
    while (1)
    {
        assertion = trigger_get(trig);
        trigger_set_status(trig,assertion);
        printf ("Assertion at timestamp=%ld \n",assertion);
    }
}
```

The last custom command example is a "watchdog time-out". In this application, an event should occur at least once every N samples. If N samples pass without a trigger assertion appearing, there is a fault condition, which is to be indicated by signaling another trigger. Action is critical when a sample does not arrive, hence examining trigger status is important to this application.

```
// WATCHDOG (tin, N, tout)
//   - examines the status of trigger tin
//   - does nothing if at least one assertion occurs
//       every N samples
//   - otherwise, signals failure in trigger tout
//       and terminates

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    TRIGGER  *tin;
    TRIGGER  *tout;

  // Storage for processing
    unsigned long int  baseline, status;
    unsigned long int  N;
    int  flag;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
      T_TRIGGER, T_CONST_W, T_TRIGGER);
    tin  = (TRIGGER *) argv[1];
    N    = *(int *) argv[2];
    tout = (TRIGGER *) argv[3];

  // Perform initializations
    trigger_open (tin,P_READ);
    trigger_open (tout,P_WRITE);
    baseline = 0xFFFFFFFF;

  // Begin continuous processing
    while (1)
    {
        /* Watch status until an event is asserted */
        status = trigger_get_immediate(tin,&flag);
        if ((status-baseline)>N)
        {
            /* Timeout.  Raise alarm, hang task here */
            trigger_put(tout,status);
            while (1)  task_switch();
        }
        trigger_set_status(tin,status);
```

```
            trigger_set_status(tout,status);
            if (flag)
                baseline = status;
            else
                task_switch();
        }
        return 0;
    }
```

8. Floating Point Support

This chapter describes the Developer's Toolkit for DAPL support for floating point computing.

Most custom commands do not need floating point. The data obtained from the analog section analog-to-digital converters is naturally represented by fixed-point values with 16-bit precision. Also, most of the processing capabilities built into the DAPL operating system are designed for direct operations on the 16-bit data. However, there are some situations in which widely-used numerical techniques are easier to represent in a floating point notation, and the extra overhead of floating point computation is a secondary consideration.

Floating point `float` and `double` data types, constants, casts, functions, and conversions are fully supported. The 80-bit `long float` type can be used, but is not supported by Developer's Toolkit for DAPL math library functions. The DAPL system provides access to shared constants, shared variables, and data pipes of `float` and `double` type.

Floating point computation is supported in one of two ways, depending on the hardware capabilities of the Data Acquisition Processor on which the custom command runs. Some of Data Acquisition Processor products feature processors with an on-chip floating point unit (FPU). When a hardware FPU is available, the FPU executes all floating point operations. When a hardware FPU is not available, floating point emulation software steps in, taking control temporarily and performing the floating point operations using software services. The primary difference is a dramatic difference in speed. If speed is not an issue, the floating point emulation may be completely satisfactory.

Floating point support is completely automatic. There are no separate floating point and non-floating point command libraries. There is nothing to configure.

Floating Point Library Functions

The floating point library functions provided with the supported compilers will not work in the Data Acquisition Processor environment. These libraries use incompatible calling conventions and incompatible exception handlers.

The Developer's Toolkit for DAPL replaces the compiler’s standard floating point library functions with equivalent functions. These functions, in addition to working

well in the DAPL environment, are smaller, faster, and make better use of a hardware FPU when available.

Ordinarily, it is necessary to include the C Standard Library header `MATH.H` when using mathematical functions in C or C++ programs. This is not necessary in the DTD environment. The math functions are defined automatically when the `DTD.H` file is included by each source code module.

All of the Standard C math functions are supported. In addition, non-standard hyperbolic functions are available.

```
cosh      sinh      tanh
acosh     asinh     atanh
```

Floating point features are accessible at the assembly language level if necessary for specialized applications. Coding may be done by inline assembly or using independently-compiled source code module. See the appropriate assembly language programming manual for details of the FPU instruction set for your particular processor family.

Pipes, constants and variables are available for `float` and `double` data types. These all work much the same. The only difference in the DTD support is that the **`pipe_get`** and **`pipe_put`** functions cannot be used with floating point data types. The new **`pipe_value_get`** and **`pipe_value_put`** functions must be used when accessing single values. The `PBUF` and buffered pipe operations work the same way for all data types.

Some commands in the DAPL system do not yet recognize `float` and `double` data types. If processing is not available on the Data Acquisition Processor, it is always possible to transfer the floating point data to the PC using a `COPY` or `MERGE` command. The difference is that the COPY command only works efficiently on one data type at a time, while MERGE is less efficient but can handle a mix of data types.

```
PIPE FL1 FLOAT, FL2 FLOAT

PDEF A
  ...
  COPY(FL1,$BINOUT)  ; Sends FL1 to the PC
  MERGE($BININ,FL2)  ; Receives FL2 from PC
...
END
```

When using the function **param_process** to check 32-bit `float` data parameter types, use the `T_PIPE_F, T_VAR_F` or `T_CONST_F` type codes. To check 64-bit `double` data types, use the `T_PIPE_D, T_VAR_D` or `T_CONST_D` type codes.

Formatting floating point numbers into ASCII strings is supported by the **printf**, **fprintf**, and **sprintf** functions. These functions have the same form as their Standard C Library counterparts, except that the output stream is a text pipe rather than a `FILE`. The *standard output* goes to the DAPL system's `$SysOut` text pipe and from there to the PC. The **fprintf** function writes to a specified text pipe rather than to the default stream.

The format conversions for the `FP` library are mostly compatible with Standard C. However, there are some differences for safety. For example, if you attempt to display a number such as 10^{302} using the `%f` formatting option in Standard C, expect to see a 1 character followed by three hundred more digits — if you are lucky and don't get a protection fault first. Instead, the DAPL system limits field sizes, and items too large to display in the available field will display as a row of asterisks, in the manner of FORTRAN or BASIC.

The DAPL operating system will maintain floating point processor state information and work areas. There is no need to worry about saving and restoring the FPU state when more than one custom command uses the FPU, but there is some additional overhead. To keep this overhead to a minimum, it is best to use floating point processing in a minimal number of processing tasks.

Floating Point Example

The following custom command illustrates the use of floating point to compute a complex algebraic function of two fixed-point pipes, writing the result to a float output pipe:

```
// FLOAT (p1, p2, fp3)
//     - reads data from pipes 'p1' and 'p2'
//     - computes a scientific function in floating point
//     - sends the results to pipe 'fp3'

#define   SCALE_FACTOR  3.0518e-5

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;

  // Storage for processing
    PIPE *p1, *p2, *fp3;
    float  x1, x2;
    GENERIC_SCALAR  pipe_value;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W, T_PIPE_W, T_PIPE_F);
    p1  = (PIPE *) argv[1];
    p2  = (PIPE *) argv[2];
    fp3 = (PIPE *) argv[3];

  // Perform initializations
    pipe_open (p1,P_READ);
    pipe_open (p2,P_READ);
    pipe_open (fp3,P_WRITE);

  // Begin continuous processing
    while (1)
    {
        /* read integer input values and scale */
        pipe_value_get(p1,&pipe_value);
        x1 = (float)(pipe_value._i16 * SCALE_FACTOR);
        pipe_value_get(p2,&pipe_value);
        x2 = (float)(pipe_value._i16 * SCALE_FACTOR);

        /* compute a function of the two input values */
        pipe_value._float = (float) (exp(3.0 * x1) /
              ((1.224e-2 * x2 + 7.5) * x2) );

        /* send the floating point result */
```

```
            pipe_value_put(fp3,&pipe_value);
        }
        return 0;
    }
```

In this example, the fixed point input values, ranging from -32768 to +32767, are scaled to fractions between -1 and 1. Then the computations are performed. The final result is written to a floating point pipe.

Floating Point Error Handling

The Developer's Toolkit for DAPL supports the `errno` feature in the manner defined in the Standard C library file `ERRNO.H`. Unfortunately, implementations provided by the supported compilers are incompatible with the DAPL environment. Do not include the file `ERRNO.H` in your source code modules. The `DTD.H` file includes these features automatically. You can see the implementation in the `MATH32.H` file. There is no penalty in codes size or efficiency if you do not use the `errno` feature.

Note: Floating point features of the DAPL system are undergoing rapid change. The C standards leave the use or non-use of the `errno` feature unspecified, at the discretion of the function libraries. The library functions might not fully utilize the `errno` feature. If this is important to you, contact Microstar Laboratories for the latest information.

The idea of the `errno` feature is very straightforward. A storage location known both to the software module and to the math function library is reserved. The software clears this location and then calls various math library functions. Upon return from the library functions, the software checks whether the stored value remains zero. If so, the library functions have flagged no error conditions.

The following error codes are used:

```
DOMAIN 33 Invalid input arguments
RANGE  34 Output overflow or underflow
```

This special storage location reserved for the error codes is called the `errno` variable. In the jargon of the system programmer, `errno` is actually a thunk, a location provided by a function call, but treat it as a simple 32-bit `int` variable.

The following is an example of error checking applied to a sequence of computations.

```
#include “dtd.h”
double a,b,c,d;
...

errno = 0;
b = exp(a);
c = exp(-a);
d = tanh((b-c)/(b+c));
if (errno)
{
    printf("Computation of term d failed.\n");
}
```

In this example, an invalid input value or an extremely large or extremely small output value will cause the `exp` library function to fail. The value of `errno` will remain zero unless one or more failure events occurs and changes its value.

There is nothing excluding the developer from using the `errno` feature for detecting many other types of errors. The C standard leaves the list of error codes unspecified and implementation-dependent. That leaves open the possibility of adding application-specific `errno` codes and corresponding operations to update `errno` when errors occur.

The `FP` library does not support the non-standard `matherr` functions. These higher-level features are coupled into incompatible operating system dependencies.

The DAPL environment initializes the FPU (or its emulated equivalent) in the default initialization mode. That is, executing the `fpinit` instruction is harmless to the DAPL system, and correctly provides the benefits of clearing the FPU and setting it to a consistent initial state. The initial state masks floating point exceptions, and standard fixes are applied after such errors as division by zero, overflow, and loss of precision.

The occurrence of errors is flagged in the FPU status word. Specialized applications can examine this word using inline assembly to determine the exact nature of the error at the instruction level. The value of the code stored in the `errno` variable is not directly related to the code in the status word, because a library function can do various fixups that alter the status after recording the diagnosis in `errno`. The following shows an example of inline assembly to extract floating point status information.

```
int statcode;
_asm
{
  fnstsw ax
  mov statcode,ax
}
if (statcode&0x20) { PRECISION_ERROR; }
if (statcode&0x10) { UNDERFLOW_ERROR; }
if (statcode&0x08) { OVERFLOW_ERROR; }
if (statcode&0x04) { ZERODIV_ERROR; }
if (statcode&0x02) { DENORMAL_ERROR; }
if (statcode&0x01) { INVALIDOP_ERROR; }
```

In the above example, the macros `PRECISION_ERROR`, `UNDERFLOW_ERROR`, and so forth, represent user defined actions. Be sure to clear the error flag bits to zero after processing so that the next error can be detected.

The trap mechanism defined in the compiler library file `SIGNAL.H` for floating point errors is not supported. If you change the control word bits to enable interrupts on floating point errors, the DAPL operating system will intercept the errors, issue a diagnostic message, and terminate the task. In general, changing the FPU exception masks is not recommended.

9. Digital Signal Processing Support

The Developer's Toolkit for DAPL provides Digital Signal Processing (DSP) functions for waveform construction, Finite Impulse Response (FIR) digital filtering, and Fast Fourier Transform (FFT) operations. The DSP functions provide access to the same optimized algorithms used by built-in DAPL commands, but with a greater degree of flexibility.

Building Custom Waveforms

Waveforms are frequently required for signal modulation operations, custom FFT "window operators," and signal generation. One way to construct waveforms is by calling the **`isine`** and **`icosine`** functions, storing the returned values in a table. An easier way is to use the **`icoswave`**, **`isinewave`**, or **`icplxwave`** function to construct a complete waveform in one operation.

These functions have a similar form:

```
icoswave ( length, cycle, size, scale, storage );
isinewave( length, cycle, size, scale, storage );
icplxwave( length, cycle, size, scale, storage );
```

The `length` and `cycle` parameters specify the amount of data generated.

- `length` specifies the number of samples to be placed into the table.
- `cycle` specifies the number of samples necessary to exactly cover one complete waveform cycle.

The table `length` may be smaller or larger than the `cycle`. For example, if one cycle of an output signal is to be covered by 100 samples, and the cycle is to be repeated five times, then the cycle `length` parameter should be 100, and the table `length` parameter should be 500.

Another example of `length` and `cycle` is for a lookup table that is to be constructed for a control application. For this system, torque applied to a pivoting object is dependent on the sine of the angle of the applied force vector. A table is used to quickly evaluate the sine function. A full cycle of tabulated data is not necessary, because ¼ cycle contains sufficient information. For example, a table of 1000 entries could be built by specifying a table length of 1000 samples and a cycle length of 4000 samples.

The `size` parameter determines the type of data generated. If `size` is set to `eWaveWord`, then two-byte (short, 16-bit) values are generated. If `size` is set to `eWaveLong`, then four-byte (long, 32-bit) values are generated.

The `scale` parameter is an unsigned value specifying the absolute magnitude of the waveform. If `scale` is one or zero, the maximum range is used for maximum precision. (The representable range is -32768 to 32767 for 16-bit data, or -2147483647 to 2147483647 for 32-bit data. The value -2147483648 is not allowed.)

When the waveform has the full magnitude, it can be treated either as a very large value or as a "normalized" signed binary fraction with the binary point immediately after the sign bit. Sometimes this representation is awkward, and other scaling is preferable. For example, specifying a `scale` parameter of 1000000 constructs a waveform which ranges from -1000000 to +1000000, for a resolution of one part in 10^6.

The data are placed into the storage location indicated by the `storage` parameter.

The **`icoswave`**, **`isinewave`**, and **`icplxwave`** functions can all generate waveform data for a full wave cycle, multiple wave cycles, or any desired fraction of a wave cycle. A storage area sufficient to contain this data must be set up by the custom command prior to constructing the waveform. Waveforms may be placed into arrays with automatic, static, or dynamic storage class. For long waveforms, it is best to allocate memory blocks dynamically using the **`ralloc`** function. For example, to set up a 32-bit waveform with 1,000 values, use the following:

```
longwave = (long *)ralloc( 1000 * sizeof(long) );
```

Strictly speaking, only one of the three functions is really necessary. A sine function contains the same information as a cosine function, except shifted by ¼ cycle. A complex waveform also contains the same information, only packed differently. Use whichever function is most convenient.

Other phase angles can be obtained by shifting either sine or cosine wave data. This property can be used to generate a table for any phase shift. For example, suppose that one full waveform of sinusoidal data is desired, with steps corresponding to 1/400 of a cycle. The **`isinewave`** function is called to construct a waveform of exactly two cycles with 400 samples-per-cycle, or 800 total samples. Phase shifts can then be established by setting a pointer to selected locations in the first 400 elements of the table. For example, a phase shift of 1/16 cycle is obtained at an offset 400/16, or 25 samples from the beginning of the data block:

```
short int  *shifted_wave;
shifted_wave =  storage+25;
first = shifted_wave[0];
second = shifted_wave[1];
```

Sine and cosine values are often needed in pairs for specialized modulation and custom transform operations. Using the **icplxwave** function, a data table can be constructed with corresponding cosine and sine terms stored pairwise. These can be considered the real and imaginary parts of a complex-valued sinusoid (equivalently, an exponential function with imaginary-valued exponent). Or, they may be considered two real numbers that are conveniently stored in a double-entry lookup table.

The following example illustrates construction of a special test waveform required to drive an output procedure. The wave is full magnitude. The output is updated every 25 microseconds. The wave consists of 1/10 second of 400 Hz baseline tone, followed by a 1/40 second tone burst of 4th harmonic tone, followed by another 1/10 second of 400 Hz tone. That is, 4000 samples of baseline tone, 1,000 samples of tone burst, then another 4000 samples of baseline are needed. At 400 Hz with 25 microsecond updates, one complete cycle requires 100 synchronous output updates. Build this special waveform with the following sequence of instructions:

```
/* Reserve 18K of memory */
short int  * tone_buffer;
int   errcode;
tone_buffer = (short int *)ralloc( 9000*sizeof(short int) );

/* Construct the three parts of the waveform */
errcode  = isinewave( 4000, 100, sizeof(short int),
   1, tone_buffer);
errcode |= isinewave( 1000, 100/4, sizeof(short int),
   1, (tone_buffer+4000) );
errcode |= isinewave( 4000, 100, sizeof(short int),
   1, (tone_buffer+5000) );
if (errcode)
{
   printf("Waveform construction failed!\n");
   exit(1);
}
```

Performing FFT Transforms

Functions provided by the Developer's Toolkit for DAPL give access to the 16-bit fixed-point transforms implemented in the DAPL system. In contrast to FFT

operations performed by a built-in DAPL FFT task, FFT operations in custom commands are performed on demand. All of the capabilities of the FFT computing engine are available to custom commands, plus many additional processing options.

FFT computations are set up and evaluated by means of the following functions:

- **fft_init** Initialize FFT processing
- **fft_request** Perform FFT processing

FFT Initialization

The **fft_init** function defines the properties of an FFT in an information structure called an FFTB, maintained by the DAPL system. This structure defines where data is stored and which processing options to apply.

There are many options for configuring an FFT operation. All information required to specify these processing options is collected into the FFTB structure. The **fft_init** function builds this structure, and returns a pointer for use by subsequent function calls.

The parameter list of the **fft_init** function has the form:

```
fft_init( size, realbuf, imagbuf, window, direction,
    solution, post, options );
```

The parameters size, realbuf and imagbuf define the data storage for the FFT operation. The window, direction, solution, post, and options parameters provide various configuration options. Each of these parameters will be discussed in detail in the next few sections of this chapter.

The **fft_init** function returns a pointer to an FFTB configuration block. If an error is detected in the function parameter list, a NULL (zero) pointer is returned. Errors are diagnosed when there is no possible interpretation of an argument value, for example a post-transform operation code which is not defined. Many inconsistencies between parameter options cannot be detected, because of the wide range of potentially valid combinations.

FFT Storage

The size parameter of the **fft_init** function specifies the length of the FFT, and consequently, determines the size of the required data areas. The size parameter specifies the number of complex input items N of the FFT, where $N = 2^M$ for some integer M. M is a number in the range 2 to 14. This range may be restricted for

particular Data Acquisition Processor models and certain DAPL versions. Note that the built-in `FFT` command provided by DAPL uses M rather than N to specify the FFT size.

In general, an FFT operation is applied to complex input data, and storage must be provided for both real and imaginary terms. Data are usually delivered to the custom command in DAPL pipes, so the buffer storage used for the blocked pipe operation can also serve as storage for FFT data. Real and imaginary parts typically arrive in separate data streams, and for this case, two storage buffers are required, with locations specified by the pointers `realbuf` and `imagbuf`.

The `size` parameter specifies the number of complex input terms—it does not specify the number of bytes of storage required. For example, suppose that a 1024 point FFT is performed on complex input data with separate real and imaginary input data streams.

```
#define FFTSIZE   1024
// wrong size!
real_buf = (short int *)ralloc(FFTSIZE);
imag_buf = (short int *)ralloc(FFTSIZE);
// correct size!
real_buf = (short int *)ralloc(FFTSIZE * sizeof(short int));
imag_buf = (short int *)ralloc(FFTSIZE * sizeof(short int));
```

In some cases, it is convenient for the FFT to operate upon complex data with real and imaginary terms stored as contiguous pairs of numbers in a single buffer. An FFT operation can be configured to use this data format by setting a flag in the `option` parameter, as will be discussed later in this chapter. For pairwise storage of complex data, the `realbuf` pointer must point to a data area which is twice as large, in order to contain twice as much data per FFT input element. The `imagbuf` parameter can be set to `NULL`.

```
#define  FFTSIZE   1024
// wrong for complex!
real_buf =
    (short int *)ralloc(FFTSIZE * sizeof(short int));
imag_buf =
    (short int *)ralloc(FFTSIZE * sizeof(short int));
// correct for complex!
real_buf =
    (short int *)ralloc(FFTSIZE * 2*sizeof(short int));
imag_buf = NULL;
```

The FFT configuration options can specify a number of output processing options that replace the input data with the FFT output data. In this case, the same buffer storage is used both for input and output values. The storage areas indicated by the `realbuf` and `imagbuf` pointers must be set up by the custom command programmer to cover all of the requirements for both input and output data. For example, an FFT can be configured to take N real input values and replace them with N 32-bit long power values. In this situation, the memory storage indicated by the `realbuf` parameter must be sufficiently large to contain N 32-bit long output values, twice as much storage as required by the input data.

```
#define  FFTSIZE   1024
short int  * input_real;
long  * output_long;

output_long = (long *)ralloc(FFTSIZE * sizeof(long));
input_real  = (short *) output_long;
```

If the FFT configuration options specify that the input data is real-valued or complex-valued but stored pairwise in a single buffer, and if the processing options select output processing that yields a real-valued result, then the `imagbuf` parameter is not needed and can be set to `NULL`.

The **`fft_init`** function should be called only once for each type of FFT transform. For instance, if the custom command computes transforms of size 256, 512, or 1024 points, three **`fft_init`** operations should be performed during command initialization, one for each size.

FFT Window Operations

The `window` parameter specifies a window operation to be applied to the data prior to performing the actual transform. The FFT window is characterized by an array of coefficients. The terms of this window are multiplied term-by-term with the values in the data arrays. The purpose of this operation is to reduce end-of-block truncation effects when FFT analysis is to be performed on a non-periodic data sequence. (The underlying theory of Discrete Fourier Transforms assumes that input data represent one period of a waveform having period N.) The window operation has the effect of a local smoothing of the FFT output spectrum. There are other side effects, however, including large changes in dominant frequency components and loss of much of the information from the beginning and end of the input data block.

There are two ways to specify a window. This parameter may be one of the pre-defined window types, specified by the following codes defined in the `CDAPCC.H` file:

- `WINDOW_RECTANGULAR`

- WINDOW_HANNING
- WINDOW_HAMMING
- WINDOW_BARTLETT
- WINDOW_BLACKMAN

A pre-defined window option will establish storage for window coefficients automatically. This is the most convenient way to apply a window operation. To make better use of storage in advanced applications where several tasks perform large FFT operations using similar window operations, it is worthwhile to establish a user-defined window vector.

WINDOW_RECTANGULAR is equivalent to no window operation, and may also be specified by a parameter value of zero. It means that data blocks are not modified prior to performing FFT computations. The other window types are the most common non-parametric window types described in the DSP literature.

Alternatively, the window parameter can specify a user-defined vector. In this case, the parameter must be a pointer to an array containing the N coefficients of the window operator. The values in the array must be 32-bit signed-long, positive values, scaled so that the range from 0 to +1 is covered by the full range of representable integers. In other words, each value can be considered a binary fraction with the binary point immediately after the leading zero (sign) bit. The storage for the user-defined array can be dynamically allocated by the custom command, for example using the **ralloc** function. The coefficients may also be defined by a VECTOR in a DAPL command file. Defining a VECTOR has the special advantage that multiple tasks can share the coefficient set. The VECTOR must be a signed long (32-bit) type, and the **vector_start** function must be used to obtain the pointer to the shared coefficient data.

The C language cannot accept a function parameter that is either an integer code or a pointer to 32-bit data; a parameter must have a single type. A compromise is reached by casting the window option, whether pointer or constant, to an unsigned long type before calling the **fft_init** function.

Windowing operations can be applied to real-valued or complex input data, for all computational methods, and either transform direction. Window operations are typically applied to real-valued time-domain data and forward direction transforms. The user should ascertain whether a window operation is appropriate before using one in other situations.

FFT Precision Options

There is more than one solution method available for computing an FFT. The computation technique is selected by the `solution` parameter.

When the `FFTSOLN_FAST` option is selected, the solution method uses faster instructions and algorithms at the expense of reduced precision, allowing more accumulated error during the FFT computation. When `FFTSOLN_ACCURATE` is selected, the solution method uses somewhat slower instructions and algorithms which retain more significant bits and round more carefully, at the expense of speed. The `FFTSOLN_FAST` option is preferred, for example, when looking for a particularly prominent frequency peak in noisy data. The `FFTSOLN_ACCURATE` version is preferred, for example, when studying low-level noise components.

When option value 0 is specified, the solution technique defaults to the `FFTSOLN_FAST` option.

FFT Direction Options

An FFT may be a forward-direction transform or a reverse-direction (inverse) transform, as specified by the value of the `direction` parameter, `FFTDIR_FORWARD` or `FFTDIR_REVERSE`. These two transforms form an inverse pair. That is, applying a forward transform and then a reverse transform yields (within computational accuracy) the original data. Applying a reverse transform and then a forward transform also yields (within computational accuracy) the original data. Even though the two transforms are mathematically very similar, they have different properties computationally. The forward transform is usually considered the transformation of time-domain data into frequency domain, and the reverse transform is usually considered the transformation of frequency domain data back into the time domain.

One of the two transforms must scale by a factor 1/N**,** in order to make the final scaling of all the terms come out right. This scaling factor may be applied either during the forward direction or the reverse direction transform. The 1/N is most commonly associated with the forward transform in the DSP literature, but this convention is not universal. In the FFT transforms provided by the Developer's Toolkit for DAPL, the 1/N factor is applied to the forward rather than the reverse transform.

This is not an arbitrary choice. As an FFT computation progresses, intermediate terms tend to grow and can overflow as terms are summed. To counter this tendency, it is advantageous to continuously scale the computations as the transform proceeds. At the end of the computation, a well-scaled transform results, with a net scaling factor of

1/N. This preserves the most significant information while avoiding overflow. For most FFT computations, the desired information is present in the peaks, and the lesser values are considered noise. The scaled forward FFT contains well-scaled information about peaks.

Not all applications have these same requirements. For example, in an application that measures harmonic distortion, the high peak value of a sine wave test signal is of no relevance. The important characteristics are the subtle low-amplitude peaks at multiples of the test frequency. For such an application, scaling the transform is a disadvantage because it suppresses the desired low-level information. A transform without the 1/N scaling is computationally a better choice to avoid loss of information.

A reverse transform can be used in place of a forward transform, to take advantage of the different scaling strategy, as long as the different properties of the two transforms are taken into account.

The first difference is that the weighting coefficients used in a reverse transform are the complex conjugates of the weighting coefficients used in a forward transform. When applied to a sequence of complex values, a reverse transform delivers transform results in reverse order. A special case of this, applying a reverse transform to a sequence of real values, produces results which are complex conjugates of the desired forward transform. In some cases the difference is of no importance—for example, conjugated data has no effect on the results of a power computation. Knowing what to expect, it is easy to adjust the data when necessary.

The second difference is that the scaling of the reverse transform can quickly send even relatively small peaks to saturation. For example, with a reverse transform of length 1024, any peak of magnitude 32 and above is effectively multiplied by 1024, causing saturation. Once saturated, it is not possible to distinguish small peaks from large ones.

The third differences is noise. The FFT computations are performed in fixed point arithmetic, so inevitably roundoff errors will accumulate. A rule of thumb is that for a length N transform where $N = 2^M$, the last M/2 bits contain noise. This is usually not a problem, however, because statistically meaningful peaks will stand out from the noise. For example, given a 1024-point transform and a very clean input signal, frequency peaks as small as 1/8 of the least-significant bit of the sampling resolution could be detectable. (Do plenty of experiments.)

The fourth difference is accuracy. Extra precision is needed to preserve all of the low-level information needed by the reverse transform. The `FFTSOLN_FAST` option does not preserve enough low-level information for most inverse FFT applications. Thus,

the FFTSOLN_ACCURATE solution method is usually necessary. There is of course a small penalty in execution time for this extra precision.

Post-FFT Processing Options

The post parameter specifies the processing steps to be applied after an FFT transform is completed. The symbols for selecting post-transform processing options are defined in the CDAPCC.H file.

Most operations are applied primarily to forward transforms with real-valued input data. The Developer's Toolkit for DAPL allows any of the post-processing options to be applied to any kind of transform, whether or not the operation has a meaningful physical interpretation, so use with care. For example, applying the FFTPOST_POWER option after a forward transform of real data yields information about power spectral density. Applying the FFTPOST_POWER option to a reverse transform of frequency spectrum data yields information about instantaneous complex power in a time-domain signal.

The available options include the following:

FFTPOST_DEFER
- Apply no post-transform processing and return no data. The input data provided to the FFT is returned without change. The FFT results may be accessed and post-processed in a separate operation at a later time. This option must be specified when it is necessary to preserve the original input data.

FFTPOST_REAL
- Extract only the real terms from the transform result, ignoring the imaginary terms.

FFTPOST_CPLX
- Extract both real and imaginary terms from the transform result, storing the complex values according to the data format specified for complex numbers. (See the discussion of the options parameter.)

FFTPOST_POWER
- Convert the transform results to power by squaring and summing real and imaginary parts. For a forward transform, this can be interpreted as power spectral density. The computed terms have 32-bit LONG precision, but the accuracy depends on the solution option (see the FFT_FAST and FFT_ACCURATE options below).
- The behavior is slightly different for real input data and complex input data. When the FFT input is complex, the power computations are always term-by-term. However, when the FFT input is real-valued, the power terms at the two ends of

the spectrum are identical and not distinguishable due to the symmetry properties of a transform. If the number of output terms is N/2 (see the FFT_HALFOUT option), the power from terms at the low and high ends of the spectrum are combined, in effect doubling the power terms. If the number of returned terms is N, the terms at the two ends of the spectrum are not combined, and an even symmetry can be observed in the data.
- Only real-valued outputs are generated. The storage specified by the realbuf parameter of the **fft_init** function is used to store the power values. Be sure that this area is sufficiently large to contain the long data type. The storage specified by the imagbuf parameter of the **fft_init** function is not affected. When input is real-valued, the imagbuf parameter can be set to NULL.

FFTPOST_NORMPOWER
- Apply power computations, almost the same as POWER, but treating the transformed values as normalized fractions, with the full output range covering the interval -1 to 1. As a practical matter, the result of this option is that each of the post-processed output values is larger by a factor of two. Sometimes, the resulting value is not representable, and is replaced by a ‘saturated’ maximum representable value. Otherwise, everything else is the same as for the FFTPOST_POWER option.

FFTPOST_MAGNITUDE
- Apply the same computations as FFTPOST_POWER, but then apply a square root operation. The result can be interpreted as the magnitude of a frequency component in the frequency domain, or as an instantaneous complex magnitude in the time domain. The output values have 16 bits precision. The storage specified by the realbuf parameter of the **fft_init** function is used to store magnitude values. The storage specified by the imagbuf parameter of the **fft_init** function is not affected.

FFTPOST_MAG_PHASE
- Apply the same computations as FFTPOST_MAGNITUDE, and also compute the phase angle (the arctangent of the ratio of imaginary part to real part).
- Both magnitude and phase values are returned in 16 bit precision. Because there are two output components, the output values are treated as if they were complex numbers. (See the processing options below). The storage specified by the realbuf parameter of the **fft_init** function is used to store magnitude values. The storage specified by the imagbuf parameter of the **fft_init** function is used to store phase values. Phase angles show an odd symmetry rather than an even symmetry when the FFT derives from real data.

Other Options

Other processing options are specified by a set of Boolean flag bits which make up the `options` parameter. Flags are merged using a bitwise OR operation, and presented to the **`fft_init`** function as a single parameter.

The option flags are used to select input and output data types. To use defaults, the options parameter may be set to zero. As a general practice, however, is it suggested that all options be declared explicitly, so that the custom command programmer doesn't have to remember which options are in effect.

At most one option flag may be specified from each of the following groups.

`FFT_REALIN`
`FFT_CPLXIN`

- These specify the type of input data provided to the FFT. Either real or complex data may be used with any solution precision, solution direction, or post-processing option.
- The impact of this option on speed is quite dramatic. For real-valued data, an alternative FFT algorithm is applied, saving roughly 40% of the computation time.
- The `imagbuf` parameter may be `NULL` if input data is real-valued and the post-processing options (such as `MAGNITUDE`) generate only real output terms.
- The default is `FFT_CPLXIN`.

`FFT_SEPARATED`
`FFT_PAIRWISE`

- The `FFT_SEPARATED` or `FFT_PAIRWISE` options select the storage organization for complex numbers. These options have an effect when there is complex-valued data on either input or output. The FFT will treat complex numbers consistently on input and output, either as pairs of values stored together, real part first and then imaginary part, or as separate terms stored in isolated buffers. Complex number arithmetic is simplified when the terms are stored together, but pipe operations may require separated terms.
- With `FFT_SEPARATED`, separate buffer areas are used for the real and imaginary terms of complex-valued inputs and outputs, and a separate `imagbuf` storage area must be provided for the imaginary parts. With `FFT_PAIRWISE`, complex terms are stored together, and the `imagbuf` parameter of the **`fft_init`** function should be `NULL`.
- The default is `FFT_SEPARATED`.

`FFT_HALFOUT`
`FFT_FULLOUT`

- Specifying FFT_HALFOUT suppresses output of the last N/2 terms of an FFT, and has some additional impacts when FFT_REALIN is in effect.
- FFT_HALFOUT is most commonly used in conjunction with the FFT_REALIN option. The FFT_HALFOUT option may be useful on occasions when the input data stream is complex, but it is known that the high frequency terms are not meaningful to the application.
- Applying an FFT to real input terms produces transformed real output terms with even symmetry, and imaginary output terms with odd symmetry. In other words, there is no additional information to be learned from the last N/2 terms of the transform. The FFT_HALFOUT option suppresses the unnecessary terms.
- There is another effect associated with this option. When FFT_REALIN is in effect, the symmetric transform artificially splits the power spectrum into two parts. When the FFT_HALFOUT option is used in conjunction with FFT_REALIN, power computations recombine the effects of high-end and low-end terms. This affects the FFTPOST_POWER, FFTPOST_NORMPOWER, FFTPOST_MAGNITUDE, and FFTPOST_MAG_PHASE processing options.
- The default option is FFT_FULLOUT.

Example of option flags:

To explicitly select the FFT options which are the default options, use the following:

```
unsigned defaultoptions;
defaultoptions = FFT_CPLXIN | FFT_SEPARATED | FFT_FULLOUT;
```

Typical FFT Options

As examples of typical FFT configurations, the following listing describes the option sets for the eight ‘modes’ supported by the FFT command provided by the DAPL system. The FFT32 command ‘modes’ are similar except that the FFTSOLN_ACCURATE solution option is used instead of the FFTSOLN_FAST option.

```
MODE 0:  Forward transform of real-valued data
  real and imaginary data buffers specified
  typically uses window operation
  FFTDIR_FORWARD,
  FFTSOLN_FAST,
  FFTPOST_CPLX,
  FFT_REALIN | FFT_FULLOUT | FFT_SEPARATED
```

```
MODE 1:  Forward transform of complex-valued data
  real and imaginary data buffers specified
  typically does not use window operation
  FFTDIR_FORWARD,
  FFTSOLN_FAST,
  FFTPOST_CPLX,
  FFT_CPLXIN | FFT_FULLOUT | FFT_SEPARATED

MODE 2:  Reverse transform of complex data retaining reals
  real and imaginary data buffers specified
  typically does not use window operation
  FFTDIR_REVERSE,
  FFTSOLN_FAST,
  FFTPOST_REAL,
  FFT_CPLXIN | FFT_FULLOUT | FFT_SEPARATED

MODE 3:  Reverse transform of complex data retaining reals
  real and imaginary data buffers specified
  typically does not use window operation
  FFTDIR_REVERSE,
  FFTSOLN_FAST,
  FFTPOST_CPLX,
  FFT_CPLXIN | FFT_FULLOUT | FFT_SEPARATED

MODE 4:  Forward transform of reals, power post-process
  real buffer specified
  typically uses window operation
  FFTDIR_FORWARD,
  FFTSOLN_FAST,
  FFTPOST_POWER,
  FFT_REALIN | FFT_HALFOUT

MODE 5:  Forward transform of reals, magnitude post-process
  real buffer specified
  typically uses window operation
  FFTDIR_FORWARD,
  FFTSOLN_FAST,
  FFTPOST_MAGNITUDE,
  FFT_REALIN | FFT_HALFOUT
```

```
MODE 6:  Forward transform of reals, mag/phase post-process
  real and imaginary buffer specified
  typically uses window operation
  FFTDIR_FORWARD,
  FFTSOLN_FAST,
  FFTPOST_MAG_PHASE,
  FFT_REALIN | FFT_HALFOUT

MODE 7:  Forward transform of reals, norm-power post-process
  real buffer specified
  typically uses window operation
  FFTDIR_FORWARD,
  FFTSOLN_FAST,
  FFTPOST_NORMPOWER,
  FFT_REALIN | FFT_HALFOUT
```

Deferred Post-FFT Processing

The raw transform result of an FFT operation is preserved until the next FFT operation is requested using the same `FFTB`. Before then, alternative post-transform processing may be applied. The results may be placed into the FFT input storage buffer area or into a different buffer area. A typical application for this option is to preserve the input data and send the FFT data to separate storage, so that both data sets can be processed further.

Use the **`fft_postop`** function to request post-FFT processing without computing a new transform. The **`fft_postop`** function has the following form:

```
fft_postop( fft, realbuf, imagbuf, post, options );
```

Note that the parameters are very much like the **`fft_init`** function parameters. The `fft` parameter provides access to the `FFTB` containing the preserved FFT result. The `realbuf` and `imagbuf` parameter specify locations for output data, which may or may not be distinctive from the storage areas originally used by the FFT. The `realbuf` and `imagbuf` parameters are used for data output exactly as the corresponding `realbuf` and `imagbuf` areas are used by the **`fft_request`** function.

The input options in the `options` parameter are ignored, but alternate output options may be specified. For example, the input to the original FFT may have been in the form of complex data pairs, but the new options can request real and imaginary parts returned separately.

In the following example, the original FFT operation returns the real and imaginary parts of a transform, and the follow-up operation returns the magnitude.

```
short int  databufr[256], databufi[256], databufm[256];

fft = fft_init( 256, databufr, databufi,
    WINDOW_RECTANGULAR, FFTDIR_FORWARD, FFTSOLN_FAST,
    FFTPOST_CPLX, defaultoptions);
fft_request(fft);

fft = fft_postop( fft, databufm, NULL,
    FFTPOST_MAGNITUDE, defaultoptions );
```

FFT Processing With More Than One Buffer

Most FFT processing involves a sequence of operations on a single data stream, but sometimes similar FFT transforms must be applied to data from a number of separate data channels. For applications with multiple data channels, the function **`fft_chngbuf`** allows setting up a single `FFTB` structure for use with number of different data buffers. A separate `FFTB` structure for each data stream is an option, but can consume a large region of memory if there are many data streams.

A call to the **`fft_chngbuf`** function has the form:

```
fft_chngbuf( pFFTB, realbuf, imagbuf);
```

The first parameter specifies the `FFTB` to be modified. The `realbuf` and `imagbuf` parameters are pointers to new real data and imaginary data storage areas respectively. If a null pointer is passed, the corresponding buffer pointer is not changed in the `FFTB` structure. It is important that the modified pointers always point to a memory area of sufficient length.

Example FFT Application

The following code uses a Fast Fourier Transform in a custom command. This custom command accepts three DAPL parameters: an input pipe, the size of the fast Fourier transform, and an output pipe. The input to the transform is real-valued data from a pipe. The results placed into an output pipe are the N points of the transform's magnitude. Note that this is different from the 'mode 5' transform of the built-in DAPL `FFT` command, which reports only N/2 output terms. Speed is considered most important in this application, so the fast solution is selected, with a slight accuracy penalty. The input data is not periodic, so a window is applied.

```
//  FFT2 (p1, n, p2)
//    - computes magnitude of a forward FFT transform
//    - data arrives in pipe p1
//    - size of transform n, expressed as number-of-values
//    - output placed into pipe p2
//    - output is magnitude values
//    - output data to pipe 'p2'

#define   FOREVER  1

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE *in_pipe;      /* Input pipe, real data */
    PIPE *out_pipe;     /* Output pipe, magnitude data */
    int  n;             /* Size of FFT blocks */

  // Storage for processing
    PBUF *inbuf, *outbuf;
    FFTB *fft;
    short int  *databuf;

  // Access parameters
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W,T_CONST_W, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];
    n = *(const short int *) argv[2];
    out_pipe = (PIPE *) argv[3];

  // Perform initializations
    /* Prepare pipes to share a buffer with the FFT*/
    pipe_open (out_pipe, P_WRITE);
    pipe_open (in_pipe, P_READ);
    inbuf = pbuf_open(in_pipe, n);
    outbuf = pbuf_open(out_pipe, 0);
    databuf = (short int *) pbuf_get_data_ptr(inbuf);
    pbuf_set_data_ptr(outbuf,databuf);
    pbuf_set_min_cnt(inbuf,n);
    pbuf_set_max_cnt(inbuf,n);
    pbuf_set_min_cnt(outbuf,n);
    pbuf_set_max_cnt(outbuf,n);
```

```
    /* Set up FFT */
    fft = fft_init( n, databuf, NULL,
        WINDOW_HANNING,        /* Use Hanning window */
        FFTDIR_FORWARD,        /* Use forward transform */
        FFTSOLN_FAST,          /* Accuracy not critical */
        FFTPOST_MAGNITUDE,     /* Compute magnitudes */
        FFT_REALIN|FFT_FULLOUT);  /* n reals in, n out */
    if (fft == NULL )
      param_error();

  // Begin continuous processing
    while ( FOREVER )
    {
        pbuf_get(inbuf);
        fft_request(fft);
        pbuf_set_cnt(outbuf,n);
        pbuf_put(outbuf);
    }
}
```

Using Finite Impulse Response Digital Filters

The Developer's Toolkit for DAPL provides a set of functions for 16-bit finite impulse response (FIR) digital filtering using a shift register filter structure. A shift register is a region of memory which records a sequence of sample values. The filter calculates an output value by multiplying the sequence of samples in the shift register, term by term, with a corresponding sequence of coefficients from a pre-defined vector. The pairwise products are then summed to yield a calculated result. For subsequent calculations, the oldest data are discarded from the shift register, and new data are introduced to replace them. The process repeats. The Developer's Toolkit for DAPL functions take care of shift register management and numerical computations. The client custom command must provide the data and define the filter characteristics.

FIR filtering is performed by means of the following sequence of functions:

- **`fir_init`** `define characteristics of a FIR filter`
- **`fir_request`** `apply filter to data`

FIR Filter Initialization

The **`fir_init`** function defines the properties of a FIR filter and its shift register in an information structure of type `FIRB`. This structure maintains information about sampled data, filter coefficients, processing options, and numerical operations. The

fir_init function returns a pointer to the allocated FIRB structure. The pointer is used by all subsequent filter operations.

The parameter list of the **fir_init** function has the form:

```
fir_init( coeffs, length, scale, decimate );
```

The coefficients in vector coeffs determine the filter's output properties. The length parameter defines the length of the coeffs vector, which in turn fixes the length of the filter shift register. The values contained in the vector determine the filter's frequency and transient response.

FIR filter design technique described in any DSP textbook can be used to derive the coefficients. Alternatively, the FGEN utility from Microstar Laboratories can be used to design the coefficient vector and analyze filter performance. The coefficients may be placed into an array in the custom command, or in a VECTOR in a DAPL command file. The vector computed during the design process is encoded as an array of signed 16-bit fixed-point fractions with 15 bits after the implied binary point, reserving the high-order bit for the sign. The coefficients can also be thought of as ordinary integer values in the range -32768 to +32767 with an extra scale factor of 1/32768 to be applied later.

The number of bits required at intermediate stages of filter calculations can become quite large. To control the growth in the number of bits, there is a scaling constraint upon the values of the coefficients.

For the case of small filters, the sum of the absolute values of the coefficients should produce a fixed-point value less than 2.0, in the binary fraction notation. Equivalently, if the coefficients are thought of as ordinary signed integers, the sum of the absolute values of the vector coefficients must not exceed 65535. If the filter vector has this property, a scale parameter value of 1 is appropriate. Equivalently, the scale parameter may be set to zero to indicate "no scaling is applied."

Filters for which the signed sum of the coefficient vector terms is 32768 times the scale parameter value have the property that the gain of the filter at zero frequency is 1.0 exactly. Most lowpass filters are designed to have this property, so that they do not alter the magnitudes of low frequency components.

For some filter designs, particularly long filters, scaling the filter terms as described above forces many coefficients to be very small, leading to a loss of precision and degraded performance. When this is the case, the coefficient values may be multiplied by a convenient power of two. This allows additional bits of precision in the filter representation. The scaling multiplier, in addition to being a power of two, should be

less than the filter length, and must be chosen so that the filter coefficient with largest absolute value is representable in a 16-bit format. The scaling multiplier must then be specified as the `scale` parameter to the **`fir_init`** function. Note that the FGEN utility can be instructed to compute an appropriate scaling factor automatically.

For example, the following filter characteristic is not properly scaled:

```
short int vfilt [11] =  {7088, 13511, 19441, 22800,
  14355, 0, -14355, -22800, -19441, -13511, -7088};
```

The sum of the absolute values of coefficients is 154390, which is greater than 65535. Since this is a relatively short filter, it may be reasonable to scale the coefficient values by the ratio 65534/154390 to obtain the following scaled filter characteristic:

```
int vfilt [11] =  {3009,5735,8252,9678,6093,0,-6093,-9678,
    -8252,-5735,-3009};
```

Now the sum of the absolute values of the coefficients is 65534, which conforms to the scaling constraint. Alternatively, 154390/4 is 38597, which is less than 65535, so the original coefficients can be used with a scaling factor of 4.

Mathematically, the operation applied by a FIR filter is a discrete convolution. This operation can be interpreted as term-by-term multiplication between a discrete-time sequence and another time-reversed discrete-time sequence. From this point of view, the terms in the filter coefficient vector may be interpreted as the time-reversed sequence of output values that result when an impulse (an isolated maximum input sample surrounded by all zeroes) is applied to the filter. This fact is not relevant to symmetric filters, as designed by the FGEN utility, because symmetric filters are the same in forward and reverse order.

The last parameter of the **`fir_init`** function is called the “decimation factor.” FIR filters are particularly well suited for lowpass filters. For example, to prevent aliasing of high frequency noise into low frequencies prior to performing an FFT analysis, it is very common to sample data at a high rate and apply digital filtering to eliminate the high frequency components. After this lowpass filtering, fewer samples are necessary to accurately represent the cleaned signal, so the sample rate can be reduced by taking one sample then skipping a constant number of samples in a cyclic manner. The length of this cycle is specified by the `decimate` parameter. If decimation is not required, this parameter should be 1, or alternatively 0, to indicate “no decimation factor.”

FIR Filter Computation

After completing the filter initialization and entering the run-time loop, the **fir_request** function is used to initiate computations. The parameter list of the **fir_request** function has the form:

```
fir_request( fir, data, count );
```

The `fir` parameter is the pointer returned by the **fir_init** function. The `data` parameter is a pointer to an array of new data to be added to the filter shift register. For example, if data is obtained from a pipe using a `get_bpipe` function, the `data` parameter may point directly to the data buffer in the pipe's `PBUF` structure. The `count` parameter specifies the number of new data samples to add to the filter shift register.

Filtered results are computed in-place and are available when the function returns. The **fir_request** function reports the number of computed values that resulted. For example, if the filter is length 40 but only 38 values have been supplied so far, the **fir_request** function will return a value of zero.

A number of initial samples are required to fill the shift register before processing can begin. For example, consider a symmetric filter of length 41. The first 40 samples, samples 0 through 39, are required to prepare the shift register. The arrival of the 41st sample, sample 40, fills the shift register and allows the first computation to proceed. This calculates a filtered value corresponding to the center location of the filter, the twenty-first sample, at sample location 20. In other words, the filter does not produce outputs corresponding to the first 20 input samples, 0 through 19. This delay is called "linear phase" or "group delay" in the linear filtering literature, but its practical effect is shifting (delaying) the output data stream by 1/2 the filter length. If this delay is important, for example, when synchronizing the filtered signal to the original signal for comparison or triggering operations, a custom command must compensate. It may inject extra values into the filter (for example, send the first sample value to the filter an extra 20 times), or replicate extra output values (for example, sending the first filter output to the command output pipe an extra 20 times).

Once the shift register is full, one result can be computed. One result is generated for each additional sample (when there is no decimation).

The amount of output data is reduced if a decimation factor greater than 1 is specified for the filter. Decimation has the effect of bypassing some of the computations. Before each computation, a number of samples equal to the decimation factor is removed from the shift register and this same number of new samples must be added. In the

event that a new data block does not have enough samples to refill the shift register, no computed result can be returned until more data become available.

Latency of a filtering command depends on the filter design and on the manner that data is collected and sent for processing. Collecting samples into longer blocks requires fewer service calls and allows more efficient processing, but results are delayed until the entire block is processed. Lowest latency is attained by passing each datum to the filter immediately when received.

The inherent delay of the filter has an impact on latency. For the previous example of the symmetric filter, 20 extra samples (samples 21 through 40) were required before the filtered result at sample 20 could be computed. This 20-sample delay directly affects the latency of the filtering process.

Additional FIR Operations

Two additional functions provide supplementary control over FIR filter operations. These are specialized functions not needed for most filtering applications.

The **`fir_change`** function may be used to change the properties of the filter without disturbing the status of the filter shift register. This could be useful, for example, to allows a user application to select from a number of smoothing (lowpass) filter characteristics by swapping filter coefficient sets.

Changing the length of the filter or the decimation factor can change data buffering requirements, leading to inefficiency, or in the worst case, insufficient storage to continue filter operation. To avoid storage problems, initialize the filter using the longest filter vector and largest decimation factor that the application will use, then apply **`fir_change`** to select the actual characteristics to be used before starting the filtering run-time loop. This guarantees that the memory allocations for the filter are adequate to cover the worst case. Extra memory will not degrade filter performance for smaller filters. Keep in mind that changing the filter length also affects the delay inherent in the filter, and can affect data synchronization.

Changing filter characteristics should be considered a relatively expensive operation, roughly equal in complexity to performing a filter computation. It should be done with great care. The **`fir_change`** function may perform extra computations to examine the new filter characteristic and select numerical techniques to apply. The extra computation could have an effect on latency.

The other specialized function is **`fir_advance`** . This function is useful in applications that must reduce data rates. For example, an application may need to perform an FFT analysis where there is a very high frequency component. In order to

preserve the high frequency information, samples must be captured at a high sampling rate, but this rate may be much too fast for a PC application to display all of the results. The **fir_advance** function has the effect of advancing the FIR filter shift register, discarding the specified number of old samples, without performing any filter computations. This guarantees that old, unneeded data are purged from the filter shift register when filtering operations resume.

One of two situations will result after using **fir_advance**. The first situation is that some of the data currently in the shift register are needed to resume computations. In this case, the application should continue to provide data to the FIR filter in the normal manner until the shift register fills, at which point computations resume automatically. The second situation is that none of the old data present in the shift register will be required again. In this case, the FIR filter is left in an "empty" state, and it must be refilled completely. It also may be necessary to purge additional samples from the data source after the shift register is empty. The return value from the **fir_advance** function reports the number of items that must be purged from the data source after calling **fir_advance**. If the return value is zero, removing data from the data source is not necessary.

For example, in the following sequence, filtering without decimation, 32 filtered values are computed and then the next 96 values are skipped.

```
/* Process 32 filtered values */
fir_apply(fir, coeff_array, 32);
/* Skip the next 96 values */
more_to_skip = fir_advance(fir, 96);
if ( more_to_skip)
  pipe_rem( inpipe, more_to_skip);
```

A Data Smoothing Application

In this example application, a data stream is obtained by sampling a continuous process. The measurements are contaminated by occasional 'noise spikes' which interfere with quality control statistics to be computed from the measurements. A statistical study demonstrated that a local smoothing operation is effective in reducing the impact of the noise spikes. The selected filter is a seven-term interpolating filter that, in effect, performs a local quadratic least-squares fit to the data, then replaces the center term with the center value of the curve fit.

The least squares fitting process results in a linear filter formula defined by the following equation.

$$X_0 = (-2 X_{-3} + 3 X_{-2} + 6 X_{-1} + 7 X_0 + 6 X_1 + 3 X_2 + -2 X_3) / 21$$

The linear formulation means that the filtering operation has an alternate interpretation in terms of lowpass filtering, and FIR filtering features can be used to implement this filter.

For properly scaling the filter, the coefficients need to sum to something less than 2.0 in the binary fraction notation. Using 32768 as a normalizing multiplier, the coefficients take on the following representation in the custom command:

```
int ls7filt[7] =  { -3121, 4681, 9362, 10924, 9362, 4681,
    -3121 };
```

These coefficients sum to 32768, which means that the zero frequency gain of the filter is exactly 1. The absolute values sum to 45252. This means that for some frequencies, it is possible that a very large amplitude signal could cause overflow, but because the coefficients are properly scaled, the overflow will be correctly saturated to the appropriate negative or positive limit. The application might be able to limit the input signal to the range -23000 to 23000, which would eliminate the possibility of overflow. Tests with actual data might also demonstrate that overflow is not a problem for the special mix of frequencies present.

The following implements the filter.

```
// LSFILT (p1, p2)
//   - reads data from pipe p1
//   - applies 7-point least-squares smoothing
//   - places results into pipe p2

#define   FOREVER  1

// FIR filter characteristic vector
static short int  ls7vect[7] =
{  -6241, 9362, 18724, 21845, 18724, 9362, -6241 } ;


int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE *in_pipe;
    PIPE *out_pipe;

  // Storage for processing
    FIRB *fir;
    GENERIC_SCALAR  gsample;

  // Access parameters
    argv = param_process (plib, &argc, 2, 2,
        T_PIPE_W, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];
    out_pipe = (PIPE *) argv[2];

  // Perform initializations
    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);

    fir = fir_init (
        &(ls7vect[0]), /* coefficients */
        7, /* length of filter */
        2, /* scale factor */
        0  /* no decimation */ );
    if (fir == NULL)
        param_error();

    /*
    ** Compensate for the 3-sample delay of a 7-term
```

```
    ** symmetric filter.
    */
    for (int i=0; i<3; ++i)
        pipe_put(out_pipe,0L);

  // Begin continuous processing
    while ( FOREVER )
    {
        /* Apply filter to "array of length 1" in gsample */
        pipe_value_get(in_pipe,&gsample);
        if (fir_request(fir,&(gsample._i16),1))
            pipe_value_put(out_pipe,&gsample);
    }
    return 0;
}
```

10. Real-Time Control

This chapter describes general considerations for using a Data Acquisition Processor as a component of a real-time system. The Data Acquisition Processor family is designed both for data acquisition and real-time control applications. Data acquisition systems place primary emphasis on fast and dependable data capture, with processing and transmission of acquired data as a secondary priority. In contrast, real-time control systems must place balanced priority on acquiring data, interpreting the data, and reporting outputs within given time constraints.

Latency

A real-time system is often required to monitor a continuous input quantity. It may be required to respond to input events on many input channels, with different data rates on each channel, while different processing is going on for each channel. Because of these many activities, sometimes the CPU cannot be assigned to process an event immediately. The time between the arrival of the input data and the delivery of the system response is called "latency."

To meet timing requirements, most real-time systems employ "interrupt-driven processing," in which a computing resource is applied upon demand, and then released for other processing. Interrupt-driven control is possible in the native processor of an 80x86-based PC, but latency covers the complete system response, not just processing of one single interrupt. A PC must contend with monitor, keyboard, disk, and real-time clock services, which compete with the control task for CPU resources. A Data Acquisition Processor, on the other hand, dedicates its full resources to acquisition and control. The kernel services of the DAPL systems are interrupt-driven and highly optimized. Furthermore, the Data Acquisition Processor provides supplemental processing hardware that can sustain accurate sampling even when the CPU resource is momentarily dedicated to other processing.

Processing speed and latency are different measures of system performance. Processing speed, also known as throughput, is determined by the average amount of CPU resource that must be applied to produce each computed result. Processing speed is optimized by collecting a large number of data samples, then processing them all at once in a highly-efficient processing loop. On the other hand, latency is introduced while waiting for samples to be collected for processing. Every real-time application must make a design trade-off between processing speed and response latency. This

chapter compares the tradeoff between latency and throughput when processing multiple channels.

Multitasking

Another common characteristic of real-time control systems is asynchronous events. Real-time software systems that attempt to anticipate every possible combination and sequence of inputs and outputs can become hopelessly difficult. One strategy for coping with this complexity is to factor the control process into a number of separate processes that (in concept) run in parallel, as independent tasks, with modules interacting through carefully controlled interfaces. DAPL provide exactly these services. Each processing command or downloaded custom command is implemented as a separate task. DAPL pipes serve as the interfaces that synchronize data exchange between tasks.

There are costs associated with the multitasking strategy. The software system must perform a certain amount of computation to maintain information about the identities of the various tasks, to select tasks for processing, and to save information about the state of each task before suspending it and assigning the CPU resource to another task. The task-switching computation is small, but it can become significant as more tasks are added and as task switching occurs more frequently. If DAPL tried to perform a task switch each time a data sample arrived, all of the CPU resource could be consumed by the task switching, with no CPU resource remaining to process the data.

To minimize the cost of task-switching and reserve CPU resources for processing operations, DAPL uses a simple task management scheme. Every processing task is given an opportunity to process the data available to it. The task will be suspended while it waits for data to arrive or when it voluntarily releases control by calling the **`task_switch`** function. To prevent any one task from consuming too many resources, DAPL enforces a limit on the amount of CPU time that an individual task can consume at any one opportunity.

The DAPL operating system provides the `SCHEDULING`, `QUANTUM` and `BUFFERING` options for adjusting tradeoffs between processing speed and latency. . The `SCHEDULING` option may be set to `ADAPTIVE` or `FIXED`. The `ADAPTIVE` scheduling mode selectively schedules tasks in an effort to balance the flow of data among all tasks. If there is a relatively balanced data flow, and real-time events occur regularly, this strategy this tends to yield efficient processing. However, there is no analytical guarantee of when any given task will be scheduled to execute. Latency could be very large for a task that handles relatively infrequent real-time events, because this task has very low data flow and is scheduled less often. The `FIXED` scheduling option guarantees that all tasks are scheduled equally often. It greatly reduces the uncertainty

of response to critical real-time events, but tends to use more CPU capacity for task switching overhead.

The `QUANTUM` option sets a limit on the interval of time that an individual task can run uninterrupted. If a task requires more than this amount of time, it is forced to temporarily release the CPU, allowing other tasks to run. When there is a mix of real-time and computational tasks, usually the computational processing should not delay real-time response. In such cases, the `QUANTUM` option should be set to a relatively small number, so that the computational tasks do not hold the CPU too long. On the other hand, the real-time system could have a computation that is time-critical. For greatest efficiency and lowest latency in this critical task, the task should run to completion. In this case, the `QUANTUM` option should be set to a relatively large number. Note that tasks that have nothing to do will release the CPU voluntarily, so there is ordinarily no time penalty for having a larger `QUANTUM` value. However, most analytical methods for guaranteeing real-time performance depend on bounding the amount of time that tasks can run, and larger bounds reduce the effectiveness of analytical methods.

The `BUFFERING` option specifies the amount of storage to be used for data buffering in pipes. Most real-time systems process data quickly without backlog, so setting `BUFFERING` to `OFF` is typical for real-time systems. Some systems will accumulate blocks of data, but then must process the data as efficiently as possible once the block is filled. Because longer blocks are processed more efficiently, such applications should probably select the `MEDIUM` or `LARGE` buffering options.

Under DAPL, each task will either complete all operations on the available data, or will be interrupted at intervals to allow other tasks to run. In most real-time systems, the desirable case is the one in which all available input data is processed in one scheduling quantum, because results become available quickly. It is the other case, however, which guarantees a predictable response time. In the worst case, a data sample is captured just after the task to process it finishes execution. Because the task did not receive that input sample, processing of that sample is delayed until all other tasks are given a chance to run. At that point, the first task will receive the input value and complete its processing. Assuming that the computations for one value can be completed in one task-scheduling interval, and assuming that N tasks are operating, the response will be available in at most N scheduling intervals.

There are a number of important conclusions. For a given configuration of tasks, response to an input is guaranteed within a fixed time interval. Since the calculation of that interval assumes that all tasks use the maximum amount of CPU at each opportunity, which is almost never the case in practice, statistical measures of response time are typically much better than the worst-case measures.

Strategies for Improving Real-Time Response

Response time of a multitasking real-time system under DAPL depends on the scheduling, the number of tasks that must process each value, and the total number of tasks that must be scheduled. Strategies for improving real-time response result from adjusting these factors.

The first strategy is taking advantage of the `SCHEDULING` and `QUANTUM` options to control the scheduling quantum and strategy. Most real-time applications will select a `FIXED` scheduling strategy and relatively small scheduling quantum.

The second strategy is reducing the number of tasks. The `TASKSTAT` command provided by DAPL will show the number of tasks present in your configuration. If there are several processing tasks, it may be possible to achieve a faster real-time response by building a custom command that combines the function of those tasks.

The third strategy is control of the CPU resource in custom commands. If your custom command cannot continue to perform computations, it is important for it to call **`task_switch`** to release the CPU to other tasks, so that the other tasks are not delayed. In some cases, however, a small delay may be acceptable. For example, if data values are processed in pairs, but only one data sample has arrived, it may be better to wait in a loop for a few cycles until the second sample arrives. Or, if other tasks must wait for the first task to obtain data, it is probably better for the first task to wait in an active loop, since scheduling the other tasks will serve no useful purpose.

Latency When Using Floating Point

There are some special considerations for real-time response when using hardware-supported floating point. The floating point unit is functionally a separate processor. It runs in parallel with the general purpose Integer Processing Unit (IPU), beginning a floating point operation when the IPU detects one in the instruction stream. Operation of the two units continues in parallel until the IPU detects another floating point instruction. At that point, if the FPU has not finished its previous operation, the integer processing unit must wait. In most cases, the delays are just a few machine cycles, but some FPU instructions take hundreds of machine cycles to complete.

Special floating point instructions are used to store and reload the state of the floating point unit after task switching has occurred. Each computation performed by the FPU alters the internal state of the FPU. If the DAPL scheduler switches from one task which is using the FPU to a second task that also needs the FPU, the state of the computations for the first task must be saved and the state of the second task's computations must be loaded. The storing and restoring are performed automatically

by DAPL, but only when needed. FPU state storing and recovery do not occur at all if fewer than two tasks use the FPU. FPU state storing and recovery are infrequent if tasks perform floating point computations at different times.

The worst case for real-time control occurs when the first task executes a floating point instruction immediately before a task switch is due. Once the task begins the storing and restoring operation, it must perform both operations before the task switching can occur. If the next task needs to execute a floating point instruction as it resumes execution, another storing and restoring operation occurs. The combination of these operations in the two tasks can introduce additional response latencies of up to 25 microseconds depending on processor and memory speed.

Single Tasking

Multitasking is an integral part of the DAPL system that cannot be switched on and off. However, the DAPL system scheduler avoids unnecessary operations. So an advanced technique for obtaining maximum efficiency is to package the entire application in one processing command. This is called an *advanced technique* because of the programming skill that will be required. A multi-tasking solution packages operations that are closely related into a task, leaving the problem of making sure that the tasks execute in a timely manner to the operating system. In a single-task solution, one command locks all processing into a rigidly structured execution sequence. If this is an optimal sequence, it will deliver better results than a multi-tasking solution. If it is a sub-optimal sequence, it will deliver poorer results, despite all of the development effort invested.

Monitoring Application Example

The `RTALARM` custom command reads from an input pipe. If the values of three consecutive samples exceed a pre-defined control limit, an alarm is raised and latched by setting a bit on the digital output port. Individual values are fetched for minimal latency. If values are not available, other tasks are allowed to run while the `RTALARM` task waits for data to arrive.

```
// RTALARM(p1, vlim, p2)
//   Fast real-time alarm for data values over the
//   control limit.
//   - Read data from input channel pipe 'p1'.
//   - Three consecutive readings must be over the
//       limit for alarm.
//   - Alarm latches bit 8 of the digital output port.

#define   OUTPUT_BIT     8
#define   CONTROL_LIMIT  10000
#define   DEBOUNCE       3

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void  **argv;
    int  argc;
    PIPE  * in_pipe;

  // Storage for processing
    short int  val;
    int  consec = 0;

  // Access parameters
    argv = param_process (plib, &argc, 1, 1, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];

  // Perform initializations - clear digital port alarm bit
    pipe_open (in_pipe, P_READ);
    digital_set_bit(OUTPUT_BIT,0);

  // Begin continuous processing
    while (1)
    {
        val = (int) pipe_get(in_pipe);
        if  ( val > CONTROL_LIMIT )
        {
            ++consec;
            if (consec >= DEBOUNCE)
                digital_set_bit(OUTPUT_BIT,1);
        }
        else  consec = 0;
    }
```

```
    return 0;
}
```

The following DAPL commands route input samples to the `RTALARM` custom command at 50 microsecond intervals.

```
reset
idef a 1
  set ipipe0 s0
  time 50
end
pdef b
  rtalarm(ipipe0);
end
start a,b
```

After the DAPL interpreter performs the `START` command, the `RTALARM` command initializes and begins continuous operation.

Customized PID Controllers

Using Developer's Toolkit for DAPL services, you can configure a control system to meet special requirements. If more functionality is needed, you can easily extend the basic controller features, adding functions for data monitoring, nonlinear output characteristics, or managing a group of control loops. If maximum speed is needed, you can build a simple configuration, trimmed to the bare essentials.

The essential properties of a PID controller are as follows:

- It produces a control output for each sample of the system output which it receives.
- The control output increases as the deviation of system output from the setpoint increases, and reduces as the system output approaches the setpoint. This is "proportional" or P-correction.
- The controller adjusts the control output to correct for errors that persist over time. This is "integral" or I-correction.
- The controller adjusts its output to oppose excessively rapid changes in the system output. This is "derivative" or D-correction.
- The controller's output is a weighted sum of the P-, I-, and D-corrections.

One implementation of PID control is the `PID1` command provided in DAPL. The `PID1` command is controls a single PID loop. Though quite general, the standard `PID1` command has some limitations. Since each control loop is managed by a separate `PID` task, there is a correspondingly large multitasking overhead when the

number of loops is large. Higher overhead means that less CPU capacity is available for managing the control loops, and that the processing rate is limited.

A PID control command is structured similarly to any other processing commend. It requires some additional initializations to set up two special data structures which are required for PID control.

Structures for PID Control

The `PID` data structure is a DAPL system structure required to maintain internal state information for a PID control loop. The **`pid_open`** function must be called once for each PID control loop, to allocate and initialize the corresponding `PID` structure. The returned pointer must be saved for use by other PID functions. The **`pid_open`** function also performs some PID initializations which require an estimate of the output of the controlled system at start-up. In some cases, you will have a good estimate for this value, for example, when the system always starts with its output at zero. In other cases, you will not have this information, and must read a sample from the input pipe at command start-up.

One or more `PIDCOEF` data structures are needed to organize PID control parameters in the custom command's local memory, and to install the parameter values using the **`pid_tune`** function. The **`pid_tune`** function must be called for each PID control loop. The fields in the `PIDCOEF` structure are:

`setpoint`	desired level of system output
`p1`	proportional correction gain, multiplier
`p2`	proportional correction gain, divisor
`i1`	integral correction gain, multiplier
`i2`	integral correction gain, divisor
`d1`	derivative correction gain, multiplier
`d2`	derivative correction gain, divisor
`clamp_lo`	output low limit clamp
`clamp_hi`	output high limit clamp

See the description of **`pid_tune`** in the command reference section for more information about the effects of the control parameters.

Most systems begin in a state of minimum energy, often called a "zero state," "resting state," or "cold start." This is the state given to the `PID` structure when it is initialized by the **`pid_open`** function. In most cases, this is the correct assumption. In other cases, it might be a terrible assumption. For example, a system might need to be manually brought to 90% of its operating speed prior to being switched over to automatic control. To make the transition as smooth as possible, the **`pid_preset`**

function can be used. The **pid_preset** function takes the known control input applied to the system, and the known feedback measurement of system output, and adjusts the internal state of the PID structure to be consistent with these conditions. Then, when automatic PID operation begins, there will be a smooth transition to the final setpoint.

The Control Loop

After the structures have been initialized, and the control parameters have been installed, the real-time update loop can begin. The following illustrates the general form of the real-time loop.

```
Loop forever
  If new control parameters are available
    Modify parameters
  Endif
  For each control loop
    If a new setpoint parameter is available
      Modify the setpoint parameter
    Endif
    Perform output computations
    Perform other control functions
    Send outputs
  End for
End loop forever
```

The real-time update loop runs continuously until it is stopped by DAPL. The loop is structured as a nest of two loops, with efficient updates in the inner loop, and relatively infrequent operations which require more computation in the outer loop.

The **pid_tune** function is used to adjust PID parameters during real-time operation. Some computation is required to do this, so the adjustments should be done in the outer loop. Frequent coefficient adjustments could limit the rate at which the application can run. Some strategies to minimize the impact of parameter changes on latency and overall speed include:

- Use fixed parameter values.
- Avoid installing parameters when there have been no parameter changes.
- Perform several inner loop passes before installing new parameters.
- If numerous parameter changes must be installed, try to spread the installation over time, rather than installing everything at once.

When updating PID parameters, all of the values in the PIDCOEF structure must be valid. If you wish to change parameters, you must save the contents of the PIDCOEF

structure, modify the fields which are to change, and pass the modified structure to the **pid_tune** function.

Most PID applications use a fixed setpoint, or at least a setpoint which is adjusted infrequently. The setpoint is established along with all of the other PID parameters when **pid_tune** is called. Other applications, which require frequent or continuous updates of the setpoint, can use the **pid_set_setpoint** function to dynamically change the setpoint without affecting the other parameters. This function is much faster than installing the full set of PID parameters, but unnecessary calls still should be avoided.

The function **pid_compute** is used in the inner loop to compute PID output corrections and update the internal state of the controller. This function is called once for each pass of the inner control loop. The parameters are the current value of the controlled system's output and the PID structure to be updated. The returned value is the computed PID control output value. This value can be modified as needed before being sent. The final result is written to a DAC using the **dac_out** function. Asynchronous DAC output is used, to avoid the data buffering and long response latencies introduced by sustained synchronized output.

Low-latency PID Response

The section show an example for a single high-performance PID loop, using a minimal configuration for lowest response latency.

The SPID2 command has the following requirements:

- It operates on a single PID control loop.
- It uses the output sign conventions of the DAPL PID command.
- Its control parameters are fixed when the application is compiled.
- It reads each datum individually.
- It sends control output directly to the DAC channel.
- It starts from a zero initial state.
- It runs continuously after startup.

The SPID2 parameters are defined as follows:

```
SPID2 (<in_pipe>, <dac_out>)
```

<in_pipe>	word pipe, feedback from system outputs
<dac_out>	word constant, DAC address for control output

This command does not make control parameter adjustments and it does not perform any supplementary control functions. The parameters are all pre-defined, built into the

custom command in a static `PIDCOEF` structure. Once the `PID` structure is initialized and the pipes are opened, the custom command reads each feedback sample, computes a response, and updates the analog output.

The following is a listing for the `SPID2` command:

```
// SPID2(pq, vdac)
// Fixed, single-loop PID control command with
// minimum latency
//    - tuning parameters fixed at compile-time
//    - reads system output feedback from 'p1'
//    - control output sent to DAC 'vdac'
//

#define  FOREVER        1
#define  OKAY           0

#define  INITIAL_STATE  0
static   PID     * PID_block ;
static   PIDCOEF   coeffs =
   {
    10000,   /* setpoint   */
    1000,    /* p1;        */
    100,     /* p2;        */
    4,       /* i1;        */
    100,     /* i2;        */
    200,     /* d1;        */
    100,     /* d2;        */
    0,       /* clamp_lo;  */
    24000    /* clamp_hi;  */
   } ;

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE  * in_pipe;
    short int  dac_id;
    GENERIC_SCALAR  pipe_value;

  // Access parameters
    argv =  param_process(plib, &argc, 2, 2,
          T_PIPE_W, T_CONST_W );
    in_pipe =   (PIPE *) argv[1];
    dac_id  =  *(short int *) argv[2];

  // Perform initializations
    pipe_open(in_pipe,P_READ);
    PID_block = pid_open( INITIAL_STATE );
```

```
      pid_tune(PID_block, &coeffs);

    // Begin continuous processing
      while (FOREVER)
      {
          pipe_value_get(in_pipe,&pipe_value);
          dac_out ( dac_id ,
             pid_compute(PID_block,pipe_value._i16) );
      }
      return 0;
  }
```

Efficient Control of Multiple Loops

In contrast to the SPID2 command, which controls a single PID loop, the BPID2 command updates a group of PID control loops. In each pass through the real-time loop, each PID in a "block" of PID controls is updated once. The response latency is approximately equal to the time to collect all of the samples. The delay allows all PID computations to be performed together, very efficiently, by a single task. It also allows the update computations to proceed in parallel with acquisition of the next data block. The latency and processing speed are much better than would be achieved by an equivalent number of independent DAPL PID1 tasks.

The BPID2 command has the following requirements:

- It operates on a block of PID control loops, configurable at task startup.
- It uses the output sign conventions of the DAPL PID command.
- It operates with fixed control parameters and setpoint.
- It uses blocked pipe operations to efficiently read system feedback signals.
- It sends control outputs directly to DAC channels.
- It runs continuously after startup.
- It runs in parallel with other DAPL tasks.

The BPID2 parameters are defined as follows:

```
BPID2 (<in_pipe>, <parameter_pipe>, <blocksize>
  <dac_vector> )
```

<in_pipe>	word pipe, feedback from system outputs
<parameter_pipe>	word pipe, source of parameter data
<blocksize>	word constant, the number of PID loops
<dac_vector>	word vector, port numbers of output DACs

The command does not perform any supplementary control functions. It uses fixed setpoint and parameter values, a pipe as a source of data for initializing PID parameters, and a set of samples from the controlled systems for initializing the `PID` structures.

The following describes the initialization logic.

```
INITIALIZE PID PARAMETERS
  Fetch one block of samples from the input pipe.
  For each PID loop
    Allocate and initialize PID structure
  End for
  While PID coefficients remain in the parameter pipe
    Fetch a group of PID parameters from parameter pipe
    Apply PID parameters to the specified loop
  End while
End INITIALIZE PID PARAMETERS.
```

The `BPID2` command obtains all configurable parameters from a single data pipe, using normal Developer's Toolkit for DAPL pipe functions. Parameters for each PID loop are read from the pipe together, as a unit. A special "tag" number precedes them, identifying the PID loop to which the parameters apply. A composite data structure, consisting of the special "tag" and the `PIDCOEF` data, is used to access the parameter data directly from the input buffer.

The `BPID2` command does not use the parameter input pipe after initialization is completed. Other commands which allow parameter changes would use the **`pipe_num`** function in the real-time loop at appropriate intervals to see whether new parameter groups have appeared.

The following listing shows the source code for the `BPID2` command.

```
// BPID2 (p1, p2, n, vdac)
// Blocked-PID real-time control command
//   - reads control parameter data from 'p2'
//   - reads system output feedback from 'p1'
//   - controls 'n' PID loops
//   - sends control outputs to DACs specified by 'vdac'

#define  iMaxPIDLoops    64
#define  FOREVER          1
#define  OKAY             0

static PID   * PID_blocks [iMaxPIDLoops] ;

struct tagged_PIDCOEF  {
    int   tag;
    PIDCOEF  coeffs;
    };

void   real_time_updates( PBUF * inpipe, VECTOR * outDACs,
          int size );


// BPID2 command main routine.

int __stdcall  ENTRY (PIB **plib)
{
  // Storage for parameters
    void **argv;
    int argc;
    PIPE    * in_pipe;
    PIPE    * param_pipe;
    VECTOR  * DACs;
    int     blocksize;

  // Storage for processing
    int     channel;
    PBUF    * coef_buf, * in_buf;
    short int   * samples;
    struct  tagged_PIDCOEF  * coeff_set;
    PID     * current_PID;

  // Access parameters
    argv =  param_process(plib, &argc, 4, 4,
          T_PIPE_W, T_PIPE_W, T_CONST_W, T_VECTOR_W );
```

```
  in_pipe =     (PIPE *) argv[1];
  param_pipe =  (PIPE *) argv[2];
  blocksize = * (short int *) argv[3];
  DACs =        (VECTOR *) argv[4];

// Perform initializations
  /* Check blocksize */
  if ( blocksize < 1  || blocksize > iMaxPIDLoops ||
       vector_length(DACs) != blocksize )
              param_error();

  /* Prepare for fetching feedback data blocks. */
  pipe_open(in_pipe,P_READ);
  in_buf = pbuf_open(in_pipe,blocksize);
  pbuf_set_min_cnt(in_buf,pbuf_get_max_cnt(in_buf));

  /* Prepare pipe for reading one tagged_PIDCOEFF */
  pipe_open(param_pipe,P_READ);
  coef_buf = pbuf_open(param_pipe,
      sizeof(struct tagged_PIDCOEF)/2);
  pbuf_set_min_cnt(coef_buf,pbuf_get_max_cnt(coef_buf));
  coeff_set = (struct  tagged_PIDCOEF *)
      pbuf_get_data_ptr(coef_buf);

  /* Fetch one input sample block to initialize */
  pbuf_get(in_buf);
  samples = (short int *) pbuf_get_data_ptr(in_buf);
  for (channel=0; channel<blocksize; ++channel)
  {
      current_PID = pid_open( samples[channel] );
      PID_blocks[channel] = current_PID;
  }

  /* Read parameter data and apply to the PID loops. */
  while ( pipe_num(param_pipe) )
  {
      pbuf_get(coef_buf);
      channel = coeff_set->tag;
      if ( channel < 0  ||  channel >= blocksize )
          continue;
      if ( pid_tune( PID_blocks[channel],
            &(coeff_set->coeffs) ) != OKAY )
          param_error();
  }
```

```
  // Begin continuous processing. Does not return.
    real_time_updates( in_buf, DACs, blocksize );
    return 0;
}  //  End of BPID2 main function

// Real-time update loop for PID control.
//
void   real_time_updates ( PBUF * in_buf, VECTOR * DACs,
    int blocksize)
{
    short int        * samples;
    short int const  * outputs;
    int  channel;

    samples = (short int *) pbuf_get_data_ptr(in_buf);
    outputs = (short int *) vector_start(DACs);

    while (FOREVER)
    {
        pbuf_get(in_buf);
        for  (channel=0; channel<blocksize; ++channel)
        {
            dac_out ( outputs[channel] ,
                pid_compute( PID_blocks[channel],
                     samples[channel] ) );
        }
    }  /* End real-time update loop */
} // End of real-time function
```

11. Tips and Techniques

This chapter provides some tips, ideas, and advanced techniques for custom module development.

Names: Module, DAPL and C++

The DAPL system imposes new constraints on name visibility. To help distinguish these, the various namespaces are briefly summarized here.

Names can have the following kinds of visibility in a C++ programming environment.

- code block visibility. Elements have automatic storage class. The names go out of scope at the end of the block. Formal parameters for a function can be considered this visibility type, as parameter names go out of scope at the end of a defining code body.
- class member visibility. This includes the `struct` as a special case. Header files declare the member names and specify `private`, `public`, and `protected` properties restricting access privileges.
- C++ `namespace` visibility. A `namespace` allows name visibility restrictions that can apply to multiple sections in multiple modules. Declarations of names and namespaces are usually shared through inclusion of one or more header files.
- Items outside of class member definitions, code blocks and `namespace` sections, and declared to be `static`, are visible within one source code compile-unit.
- Names not restricted by `class` or `namespace`, and not limited in scope by a static property, are visible globally. Global scope is equivalent to an entire downloadable DAPL module. Sharing of names is usually through inclusion of one or more header files.

Names in the C++ environment are not visible to the DAPL environment unless these names appear in an explicit named list of exported names. The list is collected automatically by the compiler tools. Interaction between the environments requires reserving the listed names in each.

Names in the DAPL system are determined by the compile-time configuration and by data content of downloadable modules. The module name is provided by the compiler. The command names are obtained from a module's data through the **`ModuleInstall`** activation protocol, executed when the module is loaded. The module names and the command names are recorded in the DAPL system, equivalent to configuration-defined symbols for processing procedures, data pipes, or shared variables.

Because the DAPL system namespace is isolated from the module namespace, task parameters are shared at runtime through distribution of handles, not through visible names. Values received via the handles can be assigned to any appropriate name in the module namespace.

For example, suppose that the pointer variable `VAR1` is declared in one of the source code units:

```
static short int volatile *VAR1;
```

The name `VAR1`, visible with compile-unit scope, establishes a pointer that can point to a 16-bit word of storage. The DAPL system command

```
VARIABLE VAR1=4
```

defines a DAPL variable named `VAR1`, which references a word of storage in the DAPL system environment. Although the module code and the DAPL system are using the same name, this does not imply that the `VAR1` name in the C++ environment automatically points to the `VAR1` storage in the DAPL environment. To establish a connection, the DAPL variable must be passed as a parameter via a DAPL task configuration. The command code must use the parameter handle to initialize access to the `VAR1` pointer:

```
VAR1 = (short int volatile *) argv[1];
```

After this statement is executed, the command's `VAR1` in the C namespace *does* refer to the same storage location as the `VAR1` variable defined in the DAPL system environment. For example, the following C statements change the value of the DAPL variable `VAR1` in the shared DAPL environment from `4` to `7`:

```
var1 = (VAR *) argv[1];
*var1 = 7;
```

Debugging Custom Commands

The DAPL operating environment is more demanding than the environment of a typical PC application. The DAPL operating system often has several tasks executing shared command code, using different data streams. Furthermore, execution of a custom command often is subject to severe timing constraints; a custom command must process data efficiently and respond rapidly to external events.

Traditional debugging tools are not well suited to debugging downloaded commands. DAPL multitasking executes hundreds of different code fragments every second; and

this activity is very difficult to trace. A debugger also tends to slow execution; this has undesirable effects in a real time operating system and can introduce spurious timing errors that in themselves are sufficient to fault the execution of the system. In short, there is little hope of getting any help from debugging tools. Your best strategy is to avoid bugs. Those who favor a chop-and-try development style should be prepared for a hard ride. Good module design and careful code review should eliminate bugs before the code is run for the first time.

System routines are not able to check their input parameters for validity. Incorrect pointers are a common source of problems. There is some protection from pointer errors, but it is not comprehensive. For example, the compiler will most likely place two static arrays in the same address space, so an indexing error in one array could cause damage in the other. A `PIPE` handle where a `PBUF` handle was expected might seem to produce correct results, but the damage might not be detected until much later with something that seems completely unrelated.

During custom command debugging, it often is useful to send intermediate results to the PC using the function **`printf`**. For example, in a preliminary test run, add `printf` function calls to display the values of extracted parameters, then terminate the task with **`exit`**. Log or display the text output. Improperly initialized values will usually be obvious. Getting this far indicates correct connections to the DAPL system, and this eliminates a major element of uncertainty. Once this test is passed, remove the temporary displays.

During real-time operation, some commands must respond too rapidly to allow execution of a **`printf`** function. In these cases, an indication of progress can be sent to the 16-bit digital output port of the Data Acquisition Processor, using the function **`digital_out`**. For example, if the output codes indicate locations in the run-time loop and the values on the port “freeze” this indicates the code location where the processing deadlock occurs.

Examining Task Scheduling

The Data Acquisition Processor constantly gathers statistics about how much CPU time various tasks use. This information is useful for analyzing CPU utilization.

The DAPL command `TASKSTAT` has two forms, `TASKSTAT CLEAR` and `TASKSTAT STATUS`. `TASKSTAT CLEAR` resets all the CPU time statistics to zero. Issue this command after starting command processing. `TASKSTAT STATUS` shows statistics about the amount of CPU time used by each task, from the time of the last `TASKSTAT CLEAR` command.

A TASKSTAT STATUS command might report something like the following:

Task	CPU Time Used (in ms)
DAPL	394
OVR_CHK	0
UND_CHK	0
INF_TSK	0
CFG_TSK	0
MEM_TSK	407
ALARM	2413
PID1	3645
system idle/overhead	4869
Average task cycle latency (in us):	210

The first six tasks are system tasks that always are defined. The ones that are never scheduled show zero CPU time used. CPU utilization for system activities such as communication and configuration is covered by *DAPL*. The ALARM and PID1 tasks can be seen to consume portions of the processing time. CPU time that is consumed by task switching and searching for tasks to schedule is covered by the system idle/overhead category.

The functions **pipe_get**, **pbuf_get**, **task_pause**, and **trigger_wait** suspend task processing when no data are available. If a task does not seem to use much CPU time, check whether it is waiting for data or waiting for some other event to occur rather than completing its computations. If a task seems to be using a disproportionately large amount of CPU time, check whether it is wasting time looking for something to do when it should be releasing the CPU for other processing.

Using Assembly Language in Custom Commands

If certain critical processing must be done with extreme efficiency, these portions of the custom command can be coded in assembly language.

One way is to inject assembly directly into C code, using the _asm directive. This is not necessarily as efficient as it would first seem. Some compiler optimizations are disabled when inline assembly code is present. Also the compiler might generate an excessive amount of register protection code as safety, even when this has no beneficial effect in practice. Have the compiler generate an assembly listing and check whether inline assembly code is effective or a bad idea.

The linker allows command code to call functions declared extern "C" and separately implemented in an Assembler module. See your compiler manuals for

information about mixed language programming. This is more work to set up, but provides complete control over code efficiency.

Note: Assembly language custom command programming is recommended only for advanced programmers. The following guidelines must be followed when writing assembly language functions for command modules.

- Modules run in a flat environment. The segment registers DS, ES, and SS are configured for the module runtime environment and must not be changed.
- The `FS` and `GS` registers must never be altered for any reason, because these are dedicated to system tasking and hardware access.
- If the string-operation direction flag is set, this flag should be cleared before leaving the assembly language routine.
- Recommended module configuration directives:

```
.486
OPTION   SEGMENT:USE32
.MODEL   flat, stdcall
```

- You can use the `.CODE` and `.DATA` directives to identify executable and data segments, respectively.
- `STDCALL` conventions are suggested but not mandatory.
- Extended features of the `PROC` directive can be used to generate the parameter access prologue and stack cleanup epilogue code automatically, for example:

```
quick_swap  PROC  NEAR STDCALL PUBLIC  \
                  USES eax ebx ecx edi, \
                  source:DWORD, dest:DWORD
```

- Assembly code must not generate software interrupts.
- Assembly code must not attempt to access BIOS features, since the code does not run in a PC environment.
- A `CLI` instruction does not guarantee that interrupts will be masked, since some Data Acquisition Processors use the nonmaskable interrupt (`NMI`). Interrupts can be masked and unmasked with the functions `sys_mask_interrupts` and `sys_unmask_interrupts`. Masking interrupts is extremely dangerous. Interrupts must not remain masked for more than a few CPU clock cycles or operating system failures will occur. If you think that your application requires interrupt masking, contact Microstar Laboratories for information specific to your software and hardware configuration.

Building Modules with Multiple Commands

Downloadable modules can contain multiple commands. This makes it possible for commands to share common code and data elements. For example the *DAPL IFM* module (Infinite impulse response Filtering Module) on the Microstar Laboratories DAPtools Standard CD-ROM provides a family of digital filters. How many filter computation functions do you think are used by the various filtering families? You are correct if you guessed that there is only one.

The following listing shows the command registration section for a module that contains two commands.

```
#include "DTD.H"
#include "DTDMOD.H"
#define TRUE  1
#define FALSE 0

int __stdcall MYCOPY_entry(PIB **plib);
int __stdcall MYLCOPY_entry(PIB **plib);

int __stdcall ModuleInstall(void *hModule)
{
  if ( CommandInstall(hModule, "MYCOPY",
           MYCOPY_entry,  NULL)   &&
       CommandInstall(hModule, "MYLCOPY",
           MYLCOPY_entry, NULL) )
    return TRUE;
  return FALSE;
}
```

The convenient naming macros cannot be used here, because each command must have a different name. Instead of one command implementation function, two functions are declared. Instead of one **`CommandInstall`** function call, there are two calls. The results from the two calls to **`CommandInstall`** are combined with a logical *AND* operator to test the overall success of the install process.

The bodies of the `MYCOPY_entry` and `MYLCOPY_entry` commands have no unusual features and are omitted from the listing above. The complete listing is provided in the `DTD32\EXAMPLES\MULTIPLE.CPP` file. That would be a good place to start when building a new multiple-command module.

12. Data Acquisition Runtime Library

This chapter describes the system routines provided by the Developer's Toolkit for DAPL. When the functions are available only for specific versions of the DAPL system or specific DAP models, the description will contain a section titled *"Restrictions"* that describes the limitations.

Service Overview

Pipe Operations

pipe_get	get a fixed-point value from a pipe
pipe_value_get	get a scalar value from a pipe
pipe_num	determine whether a pipe contains data
pipe_num_complete	return the number of data in a pipe
pipe_open	open a pipe
pipe_purge	remove all data from a pipe
pipe_put	put a fixed-point value into a pipe
pipe_value_put	put a scalar value into a pipe
pipe_rem	remove a fixed number of data values from a pipe
pipe_width	return the width of a pipe

Pipe Buffer (PBUF) Operations

pbuf_open	allocate and open a pipe buffer control block
pbuf_get	get a block of data from a pipe and report count
pbuf_put	put a block of data into a pipe
pbuf_put_set_cnt	set data count and put a block of data into a pipe
pbuf_get_cnt	determine the current data count
pbuf_get_data_ptr	fetch the pipe buffer storage pointer
pbuf_get_max_cnt	fetch the maximum data count
pbuf_get_min_cnt	fetch the minimum data count
pbuf_set_cnt	set the current data count
pbuf_set_data_ptr	set the pipe buffer storage pointer
pbuf_set_max_cnt	set the maximum data count
pbuf_set_min_cnt	set the minimum data count

Data Access

`param_error`	generate error message and terminate task
`param_error_msg`	generate task error message and terminate task
`param_process`	locate task parameters and check types
`param_type`	test a task parameter type
`free`	release dynamically allocated storage
`malloc`	dynamically allocate storage
`ralloc`	dynamically allocate task storage
`rfree`	release dynamically allocated storage

Vectors

`vector_length`	determine the length of a DAPL vector
`vector_start`	obtain a pointer to DAPL vector data
`vector_type`	return the type of data contained by the DAPL vector
`vector_width`	return the size of one data element in the DAPL vector

Task Control

`exit`	terminate a task
`task_pause`	pause for a specified time
`task_switch`	release the CPU

Text Formatting

`atof`	convert an ASCII string to a float
`printf`	format and print a string
`fprintf`	format and print a string
`sprintf`	format a string
`sscanf`	parse a string

Asynchronous Device Output

`dac_out`	send a value to a digital-to-analog converter
`digital_out`	send a value to a digital output port
`digital_set_bit`	set a single bit of a digital output port
`digital_toggle_bit`	toggle the state of a single bit of a digital output port

Triggers

trigger_get	return next available trigger assertion
trigger_get_immediate	return assertion or status immediately
trigger_get_opmode	return a trigger's operating mode
trigger_get_property	return a trigger's property value
trigger_get_status	return a trigger's current status count
trigger_num	determine if an assertion is present
trigger_open	initialize a trigger
trigger_put	place an assertion into a trigger
trigger_set_status	set a trigger’s status field
trigger_updt_put	increment a trigger’s status then assert
trigger_updt_status	increment a trigger’s status field
trigger_wait	wait for a trigger assertion

FFT

fft_chngbuf	modify FFT data pointers
fft_init	define an FFT
fft_postop	apply post-processing to FFT result
fft_request	initiate FFT processing

Digital Filters

fir_advance	bypass selected FIR filter computations
fir_change	modify FIR characteristics
fir_init	define a FIR filter
fir_request	initiate FIR filter processing

PID Feedback Control

pid_open	open and initialize a PID
pid_preset	establish a pre-determined PID operating state
pid_set_setpoint	adjust PID setpoint
pid_tune	set PID coefficients
pid_compute	compute new PID state and output

General Math

icosine	return the integer cosine of an integer value
icoswave	build an array of integer cosine values
icplxwave	build an array of integer sinusoid complex values
isine	return the integer sine of an integer value
isinewave	build an array of integer sine values
isqrt	return the integer square root of an integer value

Requests to Command Interpreter

sys_exec_command	send a command to the DAPL system interpreter
sys_get_info	return system information
sys_get_time	return the current time
sys_get_version	return the DAPL version number

Compiler Runtime Functions

The general rule for compiler runtime library functions: if a DAPL downloadable module calls these functions and builds correctly it will work correctly at runtime. However, do not expect any runtime library functions to work *a-priori*. Beyond the fact that there are no assurances of suitability for any purpose in the original PC environment, most functions in the compiler run-time libraries depend upon unavailable features such as file system, memory management, screen displays, exception handlers and so forth. *Microstar Laboratories cannot guarantee compatibility or correct operation of any functions in the compiler run-time libraries.* Unresolved name references, function parameter inconsistencies, and various syntax errors will clearly indicate when a problem exists, though it might not be clear which function or functions originated the problems. If you wish to try using compiler run-time library code, check the functions individually, and if they seem to work, use them at your discretion.

The Standard C mathematical functions are supported by the DAPL system. The implementations are very efficient and are shared by all command modules and the DAPL system kernel. Do not include the compiler's `math.h` or `errno.h` files. The equivalents of these are provided by the DTD.H and associated files.

Math functions that are supported but not covered by the C standard include the following:

```
acosh       asinh       atanh
cosh        sinh        tanh
```

atof

Convert an ASCII string to a double precision floating point value.

```
double atof (
   const char *string                    // Pointer to numeric text
   );
```

Parameters

string

A sequence of characters that can be interpreted as a numeric value.

Return Values

The function returns a double precision floating point value. If the input string has an incorrect form, the function **atof** returns the value `0.0`. The return value is undefined in case of floating point range errors.

Description

The function **atof** converts a number represented by a character string to a double precision floating point value. The input string is a sequence of characters that can be interpreted as a floating point numeric value. The string must have the following form:

```
[sign] [digits]. [digits] [exponent]
```

Leading or trailing space and/or tab characters are ignored. `sign` is an optional plus `(+)` or minus `(-)`. `digits` are one or more decimal digits. At least one digit must be present. The optional `exponent` consists of an introductory letter `e,` or `E` and an optionally signed decimal number.

dac_out

Send a value to a digital-to-analog converter asynchronously.

```
void dac_out (
   int dac_number,
   int data
   );
```

Parameters

dac_number

A number specifying the digital-to-analog converter channel. This number is 0 or 1 for the analog output ports on the Data Acquisition Processor. Larger numbers can be used when analog expansion hardware is connected to the Data Acquisition Processor.

data

A 16-bit number representing the desired output voltage.

Return Values

There is no return value.

Description

The function **dac_out** sends a value to a digital-to-analog converter. See the chapter "Voltages and Integers" in the DAPL manual for an explanation of how 16-bit numbers convert to analog output voltages.

If external analog output expansion hardware is connected to the Data Acquisition Processor, DAC channel numbers greater than one may be specified. DAC output expansion is enabled using the DAPL OUTPORT command.

The function **dac_out** updates the digital-to-analog converters immediately when it executes. This immediate response makes **dac_out** useful in low latency applications. However, it also means that update times depend on the execution scheduling for the custom command task. Task scheduling depends on the activities of all other tasks in the multi-tasking DAPL operating system so DAC updates produced by this function do not typically appear at regular intervals over time. For precise timing between DAC updates, it is recommended that a custom command write DAC data to an output channel pipe. An output procedure then can read the channel data and update the DAC synchronously.

digital_out

Send 16 data bits to a digital output port.

```
void digital_out (
   int port_number,
   int data
   );
```

Parameters

port_number
A number specifying the digital output port. This number is 0 for the digital output port on the Data Acquisition Processor. Larger numbers can be used when digital expansion hardware is connected to the Data Acquisition Processor.

data
16 bits of data.

Return Values

There is no return value.

Description

The function **digital_out** sends sixteen bits of data to the specified digital output port.

If external digital output expansion hardware is connected to the Data Acquisition Processor, digital port numbers greater than zero may be specified by the digital output functions. Digital output expansion is enabled using the DAPL OUTPORT command.

The function **digital_out** updates the digital output port immediately when it executes. This immediate response makes **digital_out** useful in low latency applications. However, it also means that update times depend on the execution scheduling for the custom command task. Task scheduling depends on the activities of all other tasks in the multi-tasking DAPL operating system so DAC updates produced by this function do not typically appear at regular intervals over time. For precise timing of digital output port updates, it is recommended that a custom command write digital output data to an output channel pipe. An output procedure then can read the channel data and update the digital output port synchronously.

digital_set_bit

Set a single bit of a digital output port.

```
int digital_set_bit (
    int bit_number,
    int data
    );
```

Parameters

bit_number

Bit identifier number. The value is in the range 0 to 15 for digital port B0, in the range 16 to 31 for digital expansion port B1, etc.

data

This value must be 0 or 1.

Return Values

The function **digital_set_bit** returns the previous state of bit *bit_number*.

Description

The function **digital_set_bit** sets the state of bit *bit_number* of the digital output port to the value of *data,* which is 0 or 1.

Bit number 0 is the least significant bit of the digital output port. If external digital output expansion hardware is present, the value of *bit_number* can exceed 15. Digital output expansion is enabled using the DAPL OUTPORT command.

Note: The **digital_out** function should be called to initialize the bit values on the digital port before calling this function. The value returned by **digital_set_bit** is undefined on power-up and after a RESTART command.

See Also

digital_out

digital_toggle_bit

Toggle the state of a single bit of a digital output port.

```
int digital_toggle_bit (
    int bit_number
    );
```

Parameters

bit_number

Bit identifier number. The value is in the range 0 to 15 for digital port B0, in the range 16 to 31 for digital expansion port B1, etc.

Return Values

The function returns the previous state of bit *bit_number*. The return value is 0 or 1.

Description

The function **digital_toggle_bit** toggles the state of bit *bit_number* of the digital output port. If the current state of the digital output bit is one, the digital output is set to zero. If the current state of the digital output bit is zero, the digital output is set to one. The function returns the state of the bit as it was prior to the toggle operation.

Bit 0 is the least significant bit of the digital output port. Digital output expansion is enabled using the DAPL `OUTPORT` command.

Note: The **digital_out** function must be called to initialize the bit values on the digital port before calling this function.

See Also

digital_out

exit

Terminate a task.

```
void exit (
   int exit_code
   );
```

Parameters

exit_code

A number that is reported to the DAPL system upon task termination

Return Values

There is no return value.

Description

The function **exit** causes a task to terminate.

The *exit code* parameter value indicates to the DAPL system a reason for task termination. The usual values are 0 to indicate no errors and 1 to indicate an abnormal condition. Only the low-order eight bits are meaningful to the DAPL system, so error code values should be in the range 0 to 255.

After a task calls **exit**, the DAPL system does not schedule the task for execution, but the task continues to appear on lists of active tasks produced by the DAPL command TASKSTAT. The task does not release temporary storage or local variables. Storage deallocation is not performed until a DAPL command STOP is executed.

fft_chngbuf

Modify FFT data pointers.

```
void  fft_chngbuf (
   FFTB * fft,                    // FFT control block handle
   short int * real,              // Pointer to storage
   short int * imag               // Pointer to storage
   );
```

Parameters

fft

Pointer variable containing a handle for the FFT control block to be modified.

real

Pointer to data storage for real-valued terms.

imag

Pointer to data storage for imaginary-valued terms.

Return Values

There is no return value.

Description

The function **fft_chngbuf** changes the real and imaginary data pointers previously installed in an FFTB. The control block is identified by the handle *fft*. This function allows a single FFTB to refer to data blocks from multiple data streams. The change takes effect with the next operation that uses the specified FFTB.

See Also

fft_init

fft_init

Define an FFT.

```
FFTB *fft_init (
   int size,
   short int *realbuf,                  // Pointer to storage
   short int *imagbuf,                  // Pointer to storage
   unsigned long window,                // Enumeration pointer
   int direction,                       // Enumeration
   int solver,                          // Enumeration
   int post,                            // Enumeration
   int options                          // Bit mask
   );
```

Parameters

size

The length of the FFT and required data areas. It specifies the number of complex input items N, where $N = 2^M$ for integer M in the range 2 to 14. This range may be restricted for particular Data Acquisition Processor models and certain DAPL versions.

realbuf

Pointer to a data storage area for real-valued terms.

imagbuf

Pointer to a data storage area for imaginary-valued terms. The *imagbuf* pointer can be null if imaginary data storage is not needed for either input data or output data.

window

Either a window operator predefined enumeration, or a pointer to an array of length *size* containing the 32-bit values defining a window operator. The predefined enumeration codes include the following:

```
WINDOW_RECTANGULAR
WINDOW_HANNING
WINDOW_HAMMING
WINDOW_BARTLETT
WINDOW_BLACKMAN
```

Optionally, this parameter can specify a pointer to an array of long values explicitly defining a window. Cast the pointer to an `unsigned long` type.

direction
One of the following codes:

```
FFTDIR_FORWARD
FFTDIR_REVERSE
```

solver
One of the following codes:

```
FFTSOLN_FAST
FFTSOLN_ACCURATE
```

post
One of the following codes:

```
FFTPOST_DEFER
FFTPOST_REAL
FFTPOST_CPLX
FFTPOST_POWER
FFTPOST_NORMPOWER
FFTPOST_MAGNITUDE
FFTPOST_MAG_PHASE
```

options
"Flag" bits that are combined using bitwise OR operations to select additional processing options. One option from each of the three groups may be selected:

```
FFT_REALIN
FFT_CPLXIN

FFT_SEPARATED
FFT_PAIRWISE

FFT_HALFOUT
FFT_FULLOUT
```

Return Values

The function returns a pointer to a `FFTB` configuration block, which is used by all other FFT functions.

Description

The function **fft_init** allocates an FFT control block structure and initializes it with the options that define the characteristics of the FFT and its related operations. The actual operations are performed separately.

The *realbuf* and *imagbuf* parameters specify pointers to data storage areas for real-valued and imaginary-valued terms respectively. The *imagbuf* pointer can be NULL if imaginary data storage is not needed for either input data or output data. The **fft_request** function will fetch input data using these pointers. Depending on processing options, it also uses the same storage for returning results.

The storage must be allocated by the custom command, and must cover all input and output requirements. The **ralloc** function can be used to obtain storage blocks. The number of items to reserve is sometimes but not always equal to the number specified by the *size* parameter. Some examples:

- *Complex input data.* When the input data is complex and stored in multiplexed fashion using the FFT_PAIRWISE option, both real and imaginary terms are provided by one data source, the *realbuf* array. The *realbuf* array requires *2 * size* terms.
- *Half-length output data.* With processing options FFT_HALF and FFT_CPLX, the number of real input terms equals *size*, but after transforming, *1/2 * size* terms each are used for storing the real and imaginary results.
- *Power output post-processing.* Using real input data and the post-processing options FFTPOST_POWER and FFT_FULLOUT, the number of terms returned is *size*, but the data type is long int rather than int. The *realbuf* array must allow for *2 * size* terms rather than *size* terms in its memory allocation.

The *window* parameter specifies either a pre-defined enumeration code for a window operator or a pointer to an array of length *size* containing window operator terms. The DAPL system can distinguish pointer values from enumeration codes, so the meaning of the parameter is unambiguous. Unfortunately, C syntax does not allow parameter type overloading, so a choice must be made between an unsigned long int or a pointer type. The function **fft_init** requires the unsigned long type. If a custom window vector is used, type cast the array storage pointer to an unsigned long type to satisfy the compiler.

The *direction* parameter specifies a forward transform, typically used for transforming from time-domain data to frequency-domain, or a reverse transform, typically for transforming from frequency-domain data to time-domain.

The *solver* parameter allows a selection of computational methods, one optimized for speed and with noisy data, the other optimized for accuracy with clean, precise data.

The *post* and *options* parameters provide additional control over the representation of the input data and the output results.

See Chapter 9 for more information about the meaning and application of the various configuration options.

See Also

`fft_request`, `ralloc`

fft_postop

Apply post-processing to an FFT result.

```
int fft_postop (
   FFTB *fft,                          // FFT control block handle
   short int *realbuf,                 // Pointer to storage
   short int *imagbuf,                 // Pointer to storage
   int post,
   int options
   );
```

Parameters

fft

Pointer variable containing a handle for the FFT control block to be used.

realbuf

Pointer to a data storage area for real-valued terms.

imagbuf

Pointer to a data storage area for imaginary-valued terms.

post

One of the following codes:

```
FFTPOST_REAL
FFTPOST_CPLX
FFTPOST_POWER
FFTPOST_NORMPOWER
FFTPOST_MAGNITUDE
FFTPOST_MAG_PHASE
```

options

"Flag" bits that are combined using bitwise OR operations to select additional processing options. One option from each of the three groups may be selected:

```
FFT_SEPARATED
FFT_PAIRWISE

FFT_HALFOUT
FFT_FULLOUT
```

Return Values

The function returns a nonzero error code if a parameter error is detected, or a 0 code if the operation is completed.

Description

The function **fft_postop** performs post-transform processing on an FFT result after FFT computations are completed but before a subsequent FFT is performed using the same FFTB configuration block. This operation allows additional processing, beyond that which is done by the original FFT operation. It also allows separation of input and output processing, so that input data are not replaced by output data.

When the FFTPOST_DEFER option is selected in the call to the **fft_init** function, the **fft_request** function does not return any data so a call to the **fft_postop** function is required to make computation results available to the command.

The parameters are very similar to the processing options of the **fft_init** function.

The *realbuf* and *imagbuf* fields must specify pointers to data storage areas for real-valued and imaginary-valued output terms. The custom command must allocate sufficient storage to cover all output requirements.

The *post* and *options* parameters provide additional control over the representation of the input data and the output results.

See Chapter 9 for more information about the various configuration options.

See Also

fft_init, **fft_request**

fft_request

Initiate FFT processing.

```
void  fft_request (
   FFTB * fft                          // FFT control block handle
   );
```

Parameters

fft

Pointer variable containing a handle for the FFT control block to be used.

Return Values

There is no return value. The results of the FFT computation are returned in the FFT control block.

Description

The function **fft_request** initiates FFT computation, using the configuration previously established by the **fft_init** function. The custom command is required to place the input data for the FFT operation into the storage arrays prior to making this function call.

See Also

fft_init

fir_advance

Bypass selected FIR filter computations.

```
int fir_advance (
   FIRB *fir,                          // FIR filter control block handle
   int count
   );
```

Parameters

fir

Pointer variable containing a handle for the FIR filter control block to be adjusted.

count

A value specifying the number of items to be removed from the data source.

Return Values

The function returns the number of additional items that must be removed from the data source.

Description

The function **fir_advance** is an optional function to advance data through a FIR filter internal shift register, bypassing selected filtering operations. A normal filtering operation removes old data from the filter, adds new data to replace them, and then performs filter computations. The **fir_advance** function removes old data, without replacing with new data, and without performing any filter computations.

The function **fir_advance** reports the number of additional items that must be removed from the data source. If just a few items are bypassed, the filter shift register is not emptied, the function returns the value zero, and filtering resumes automatically when enough new data are provided by function **fir_request** to refill the shift register. If the *count* is larger than the number of items present in the shift register, **fir_advance** reports the number of additional items that must be skipped by the calling program before refilling the filter shift register.

The most common application of function **fir_advance** is data skipping, for example, capturing data at a high sampling rate to preserve high frequency

information, but eliminating large blocks to avoid excessive data volume. Another application is specialized decimating filters.

See Also

`fir_request`

fir_change

Modify FIR characteristics.

```
int fir_change (
   FIRB *fir,                    // FIR filter control block handle
   short int * coeffs,           // Pointer to coefficient array
   int length,
   int scale,
   int decimate
   );
```

Parameters

fir
Pointer variable containing a handle for the FIR filter control block to be modified.

coeffs
An array containing the coefficients that determine the computational characteristics of the filter.

length
A value specifying the number of terms in the *coeffs* array, up to 1024.

scale
A value specifying an optional non-negative scaling constant.

decimate
A non-negative number.

Return Values

If the function succeeds and the change is installed successfully, the return value is 0. If the space previously allocated for the filter is not sufficient, or if any of the new filter characteristics are invalid, a nonzero error code is returned.

Description

The function **fir_change** changes filter characteristics after initialization by the **fir_init** function. The parameters of this function correspond to the parameters of the **fir_init** function, with the addition of the first parameter *fir*, which specifies the filter to be modified. This function does not allocate a new FIRB structures.

This function should be used with care, because it can affect efficiency, output continuity, phase and latency. For example, if the filter is made longer, the internal shift register previously filled is suddenly not filled. The filter will cease generating output values until a number of new samples are provided. Similarly, reducing the filter length can leave the filter somewhat overfilled, causing an unexpected burst of output results the next time a filtering operation is requested. The filter reserves extra space for computational efficiency when it is initialized, but efficiency may drop if that extra space is consumed by a longer filter structure.

Changing coefficient values in *coeffs* data storage after initialization can interfere with the evaluation of the filter. The only guaranteed way to "tune" coefficients safely is to compute them in separate array storage, and then switch to the new array with a call to **fir_change**.

All parameter values must be specified. If some of the parameters are unchanged, specify the old values.

See Also

fir_init

fir_init

Define a FIR filter.

```
FIRB *fir_init (
   short int * coeffs,                    // Pointer to coefficient array
   int length,
   int scale,
   int decimate
   );
```

Parameters

coeffs
An array containing the coefficients that determine the computational characteristics of the filter.

length
A value specifying the number of terms in the *coeffs* array, up to 1024.

scale
A value specifying an optional non-negative scaling constant.

decimate
A non-negative number.

Return Values

The function returns a pointer containing a handle value required by all subsequent filter operations.

Description

The function **fir_init** allocates a FIR digital filter control block structure and initializes it with the options which define the characteristics of the filter. The actual operations are performed separately.

The coefficients which determine the computational characteristics of the filter are provided to the function **fir_init** in the array *coeffs*. The *length* parameter specifies the number of terms in the *coeffs* array, up to 1024. The length of the filter equals the length of this vector.

The *scale* parameter specifies an optional non-negative scaling constant. The scaling is applied after other filter computations, dividing the intermediate filter

result by the specified amount to produce the final filter result. The scale factor must be an exact power of 2, and must be smaller than the *length* parameter. The final scaling operation is bypassed if the *scale* parameter has a value 1 or 0.

The *decimate* parameter is a non-negative number. If the *decimate* parameter is greater than 1, one filter value is computed and then *decimate-1* values are skipped, so that *decimate* values are consumed for each filter output value generated. A *decimate* value of 1 or 0 indicates that no decimation is to be applied, and each input value will generate one corresponding output value.

The returned value is a handle required by all subsequent filter operations. If this returned pointer is a NULL pointer, there is a parameter error, and the **fir_init** function was unable to configure a filter as specified.

See Chapter 9 for more information about the meaning and application of the various configuration options.

See Also

fir_change, **fir_request**

fir_request

Initiate FIR filter processing.

```
int  fir_request (
   FIRB * fir,                          // FIR filter control block handle
   short int * data,                    // Data to be filtered
   int count
   );
```

Parameters

fir
Pointer variable containing a handle for the FIR filter control block to be used.

data
An array containing the data to which the filter is applied. Result values will replace the original data in this array.

count
The number of filter input values in the data array.

Return Values

The function returns a status code. If filter computations are in progress, the returned code is -1. If the amount of data provided in the `data` array is not sufficient to fill the internal filter shift register, and computations cannot proceed, a 0 is returned. The function can return a positive value indicating the number of results generated.

Description

The function **`fir_request`** initiates digital filter computations, using the configuration previously established by the **`fir_init`** function. The filter operation is applied to data provided in the *`data`* array. The *`count`* parameter specifies how many items are provided to the filter. Result values replace the original data in the *`data`* array.

See Also

`fir_init`

fprintf

Format and print a string.

```
int fprintf (
   PIPE *output,                    // Pipe handle
   char *format_string,             // Pointer to conversion string
   ...                              // Additional parameters
   );
```

Parameters

output

Pointer variable containing a handle for the pipe to be examined.

format_string

A string of ASCII characters controlling the conversions.

...

A varying number of characters appearing after the mandatory parameters.

Return Values

The function returns the number of characters sent.

Description

The function **fprintf** formats characters and values into a string and sends the string to the byte output pipe specified in *output*. This function is similar to the fprintf function defined in Standard C, except that the *output* destination is a byte pipe rather than a STREAM. The full set of Standard C conversion codes is supported in *format_string*, except for long long and long double conversions and data types.

Note: To bound stack usage, there is a 132-character limit on the length of the final formatted string. Be particularly careful not to format a very large floating point number using the %f format conversion code.

free

Release dynamically allocated storage.

```
void  free (
   void   * storage
   );
```

Parameters

storage

The pointer to storage previously allocated by function **malloc**.

Return Values

None.

Description

The function **free** releases storage allocations previously obtained using the function **malloc**. Do not call this function for memory obtained by other means.

The C++ `delete` and `delete[]` operations use this function implicitly . However, **free** can be used explicitly for managing dynamically-constructed data structures of various types.

See Also

malloc, **ralloc**, **rfree**

icosine

Return the integer cosine of an integer value.

```
int icosine (
    short int ang
    );
```

Parameters

ang

An input angle. The angle is interpreted in radians, as a 16-bit signed fractional multiple of PI. The integer values -32768 to +32768 represent angles of (-32768 * PI) / 32768 to (+32767 * PI) / 32768. For example, an input value of 16384 represents an angle of PI/2 radians and an input value of -16384 represents an angle of -PI/2 radians.

Return Values

The function returns the trigonometric cosine of angle *ang*. The result is in undimensioned units, as a 16-bit signed fraction of 1.0. For example, a result value of 16384 represents a cosine value of 1/2.

Description

The function **icosine** returns the trigonometric cosine of angle *ang* in a fixed-point representation. Ideally the values -1.0 through +1.0 would be represented by the fixed point range -32768 to +32768, but due to a non-symmetry of the processor hardware, the value of +32768 cannot be reached. For most purposes, it is sufficient to treat the value +32767 as the representation for cosine value 1.0.

The integer cosine computation performed by **icosine** is considerably faster than the floating point cosine computation performed by the `cos` function in the Standard C library. For many applications the fixed point approximation is sufficient.

See Also

icoswave

icoswave

Build an array of integer cosine values.

```
int icoswave (
   long ltb,
   long lcyc,
   long lw,                          // Enumeration
   long iscale,
   void *storage                     // Pointer to data storage array
   );
```

Parameters

ltb
The number of entries actually constructed in the table.

lcyc
A value specifying the number of samples necessary to cover one complete cycle (two PI radians) of the wave.

lw
A code indicating the data type to place into storage.

iscale
A value specifying a signed scaling multiplier.

storage
A pointer to the storage location where values are to be stored.

Return Values

The function returns a Boolean error flag. The returned value is 0 if the data array is constructed successfully, or nonzero otherwise.

Description

The function **icoswave** is a utility for constructing trigonometric waveform tables. Applications include specialized transforms and signal generation.

A table with *ltb* values is constructed in the storage location specified by pointer *storage*. The *lcyc* parameter specifies the number of samples necessary to cover one complete cycle (two PI radians) of the wave. The *ltb* parameter specifies the number of entries actually constructed in the table. The *ltb* value may be smaller or

greater than the *lcyc* parameter. For example, 1/4 cycle of a cosine wave of 2000 points, including the two endpoints bounding this interval, can be specified by setting the *lcyc* parameter to (2000/4)+1.

The values are stored starting at the location specified by pointer *storage*. The type of the data stored there depends on the value of the *lw* parameter. If *lw* is `eWaveWord`, data of type `short int` is placed into the array storage. If *lw* is `eWaveLong`, data of type `long` is placed into the array storage.

This function does not dynamically allocate memory for the waveform data. This allows great flexibility, but it also means that care must be taken to allocate sufficient storage and correctly specify the *storage* pointer. For example, in the 1/4 wave example above, storage for the 501 short integer values can be requested at task initialization time using the **ralloc** function:

```
short int *waveptr;
waveptr = ralloc((500+1)*sizeof(short int));
```

The values may be scaled by a signed multiplier given by the *iscale* parameter. For 16-bit data, the multiplier can range from -32767 to +32767; and for 32-bit data the multiplier can range from -2147483647 to +2147483647. The multiplier can be interpreted as a bound on the range of the waveform, or as a binary fraction multiplier in the range -1 to +1. An *iscale* parameter value of zero means that the waveform covers the maximum range, with no scaling applied to the data.

The waveform values are represented as a fixed-point binary fraction. The most significant bit is the sign bit, and the remaining bits are a binary fraction, with the first bit after the binary point immediately following the sign bit.

The **icoswave** function returns an error code. An error condition will be indicated if any of the following constraints are violated:

- The data type code is neither `eWaveWord` nor `eWaveLong`.
- The *lcyc* parameter is greater than 65536.
- The total amount of storage required for the table is greater than 32768 bytes.

Note: The greatest accuracy is obtained when the cycle length specified by parameter *lcyc* is equal to a power of 2 and the waveform is not scaled.

See Also

ralloc

icplxwave

Build an array of integer sinusoid complex values.

```
int icplxwave (
   long ltb,
   long lcyc,
   long lw,                          // Enumeration
   long iscale,
   void *storage                     // Pointer to data storage array
   );
```

Parameters

ltb
The number of entries actually constructed in the table.

lcyc
A value specifying the number of samples necessary to cover one complete cycle (two PI radians) of the wave.

lw
A code indicating the data type to place into storage. Specify `eWaveWord` or `eWaveLong`.

iscale
A value specifying a signed scaling multiplier.

storage
A pointer to the storage location where values are to be stored.

Return Values

The function returns an array of sinusoid values. The values are stored pairwise, cosine term first followed by the sine term.

Description

The function **`icplxwave`** is a utility for constructing trigonometric waveform tables. Applications include specialized transforms and signal modulation. This function is like a combination of the **`icoswave`** function and the **`isinewave`** function, except that the returned values are stored pairwise, cosine term first followed by the sine term, rather than in separate areas.

Because both cosine and sine terms are stored in the data array, the amount of storage allocated for the data array is twice as much as required for the **icoswave** function. In other respects, the parameters are the same as for the **icoswave** function.

See Also

icoswave, **isinewave**

isine

Return the integer sine of an integer value.

```
int isine (
    short int ang
    );
```

Parameters

ang

An input angle, interpreted in radians, as a 16-bit signed fractional multiple of PI. The integer values -32768 to +32768 represent angles of (-32768 * PI) / 32768 to (+32767 * PI) / 32768. For example, an input value of 16384 represents an angle of PI/2 radians and an input value of -16384 represents an angle of -PI/2 radians.

Return Values

The function returns the trigonometric sine of angle *ang*. The result is in undimensioned units, as a 16-bit signed fraction of 1.0. For example, a result value of 16384 represents a sine value of 1/2.

Description

The function **isine** returns the trigonometric sine of angle *ang* in a fixed-point representation. Ideally the values -1.0 through +1.0 would be represented by the fixed point range -32768 to +32768, but due to a non-symmetry of the processor hardware, the value of +32768 cannot be reached. For most purposes, it is sufficient to treat the value +32767 as the representation for cosine value 1.0.

The integer sine computation performed by **isine** is considerably faster that the floating point sine computation performed by the `sin` function in the Standard C library. For many applications the fixed point approximation is sufficient.

See Also

isinewave

isinewave

Build an array of integer sine values.

```
int isinewave (
   long ltb,
   long lcyc,
   long lw,                          // Enumeration
   long iscale,
   void *storage                     // Pointer to data storage array
   );
```

Parameters

ltb
The number of entries actually constructed in the table.

lcyc
A value specifying the number of samples necessary to cover one complete cycle (two PI radians) of the wave.

lw
A code indicating the data type to place into storage. Specify `eWaveWord` or `eWaveLong`.

iscale
A value specifying a signed scaling multiplier.

storage
A pointer to the storage location where values are to be stored.

Return Values

The function returns a Boolean error flag. The returned value is 0 if the data array is constructed successfully, or nonzero otherwise.

Description

The function **`isinewave`** is a utility for constructing trigonometric waveform tables. Applications include specialized transforms and signal generation. This function and its parameters are identical to the **`icoswave`** function, except that the values of the sine function rather than the cosine function are returned in the data array.

See Also

icoswave

isqrt

Return the integer square root of an integer value.

```
long int isqrt (
    long int x
    );
```

Parameters

x

A long integer.

Return Values

The function returns the integer part of the real-valued square root of the long integer parameter *x*. If the input value is negative, **isqrt** returns zero.

Description

The integer square root computation performed by **isqrt** is considerably faster that the floating-point square-root computation performed by the function `sqrt`.

malloc

Dynamically allocate bulk storage.

```
void * malloc (
   unsigned  size
   );
```

Parameters

size

The size, in bytes, of the storage to be allocated to a task.

Return Values

The function returns a pointer to the block of allocated storage, or a NULL pointer if insufficient memory is available

Description

The function **malloc** allocates storage to a task and returns a pointer to this storage. De-allocation is performed automatically when a STOP command is issued or when a free function is called. The storage size is guaranteed to be at least *size* bytes. The storage can exist in a pooled storage segment for efficiency, so it is possible that more than *size* bytes are physically addressable. However, it is essential to access only the amount of storage allocated to avoid corrupting task and system data.

The function **malloc** is preferred over the function **ralloc** for the purposes of constructing C++ objects. It is independent of task and diagnostic features. However, without these features and without an ability to raise an exception to report failure conditions, it is necessary to test the returned pointer value for a NULL value to verify the success of constructor operation. In contrast, the **ralloc** function will issue a diagnostic message and terminate the task in the event of an allocation failure.

See Also

ralloc

param_error

Generate an error message and then terminate task.

```
void param_error (
  );
```

Parameters

This function requires no parameters.

Return Values

There is no return value.

Description

The function **param_error** prints a diagnostic message in the following form and then callsfunction **exit**:

```
<commandname>: parameter error
```

If the DAPL ERRORQ option is on, the error message is suppressed and ERRORQ is set to a nonzero value.

See Also

param_error_msg, **exit**

param_error_msg

Generate a task error message and then terminate task.

```
void param_error_msg (
   enum ParamErrors pecode,          // Enumeration
   int ip
   );
```

Parameters

pecode

A code indicating the type of parameter error to be described in the error message.

The value is one of the following:

pe_GeneralError	No
pe_LengthInconsistent	Vector or array size mismatch
pe_SizeInconsistent	Precision error
pe_TypeInconsistent	Inconsistent data types
pe_ValueInconsistent	Inconsistent parameter value
pe_ValueOutOfRange	Range limit exceeded
pe_ValueNotAllowed	Invalid parameter value
pe_OptionNotAllowed	Invalid optional parameter
pe_ParamMissing	Invalid number of parameters
pe_ExtraParam	Invalid number of parameters
pe_ParamType	Invalid parameter type

ip

The value of this parameter indicates which parameter is incorrect, counting parameters from left to right starting with 1.

Return Values

There is no return value.

Description

The function **param_error_msg** prints an error message in the following form and then calls terminates the task:

```
Error 1236: <cmdname> - parameter <ip> - <descriptive text>
```

The function **param_error_msg** should be used rather than function **param_error** when diagnostic information is necessary to identify the error.

Code numbers for *pecode* are defined in the file DTDCNSTS.H and are included automatically when the DTD.H file is included in each source code module. The DAPL operating system supplies the cmdname, and also provides the descriptive text based on the value of the parameter *pecode*. The parameter number ip indicates the position in the parameter list where the error was detected. This number corresponds to the index that would be used to access the parameter from the argv pointer list generated by the **param_process** function.

If the DAPL ERRORQ option is on, the error message is suppressed and ERRORQ is set to a nonzero value.

See Also

param_error, **exit**

param_process

Locate task parameters and check types.

```
void **param_process (
   PIB **plib,                          // Parameter block handle
   int *argc,                           // Pointer to integer
   int min_arg,
   int max_arg,
   ...                                  // Additional parameters
   );
```

Parameters

plib
Pointer variable containing a handle for the PIB to be examined.

argc
Pointer to a variable reserved for the number of actual parameters.

min_arg
The minimum number of task parameters.

max_arg
The maximum number of task parameters.

...
A varying number of data type names appear after the mandatory parameters.

Return Values

There is no return value.

Description

The function **param_process** generates an argument vector from a task's parameter-list information block (PLIB). The function places the number of actual parameters in *argc* and returns a pointer to an array *argv* of task arguments. The parameters then can be referenced by indexing *argv*:

```
argv[0]  - the name of the custom command
argv[1]  - parameter 1
argv[2]  - parameter 2
argv[3]  - parameter 3
       .
       .
       .
```

Note that this method of referencing task parameters is very similar to the manner in which Standard C references command line parameters. The differences are that Standard C command line parameters are always strings, while task parameters can be other data types; and Standard C includes `argv[0]` in its parameter count while DAPL does not.

The function **`param_process`** also checks that the numbers and types of the parameters passed to a task are correct. The number of actual parameters specified by a task definition using this command must be between *`min_arg`* and *`max_arg`*. The types of the parameters must match the parameter types that follow *`max_arg`*. The number of parameters after *`max_arg`* must equal the value of *`max_arg`*.

The parameter type codes are provided in the file `DTDCNSTS.H` which is included in each source module automatically when the `DTD.H` file is included. The supported type codes include the following:

`T_VAR_W`	`T_VAR_L`	`T_CONST_W`	`T_CONST_L`	`T_TRIGGER`
`T_PIPE_B`	`T_PIPE_W`	`T_PIPE_L`	`T_RFLAG`	`T_STR`
`T_VECTOR_W`	`T_VECTOR_L`	`T_VAR_F`	`T_PIPE_F`	`T_VAR_D`
`T_PIPE_D`	`T_CONST_F`	`T_VECTOR_F`	`T_CONST_D`	`T_VECTOR_D`

The type codes are also described in the chapter *Using Developer's Toolkit Functions*. If a task allows several types for a parameter, the C *bitwise-or* operation can be used to combine the type codes.

If **`param_process`** finds a parameter list error, the function prints an error message and halts the task. If an error occurs when the DAPL `ERRORQ` option is on, the error message is suppressed and `ERRORQ` is set to a nonzero value.

param_type

Test a task parameter type.

```
unsigned long   param_type (
   PIB **plib,                          // Parameter block handle
   int pnum
   );
```

Parameters

plib

Pointer variable containing a handle for the PIB to be examined.

pnum

Task parameter number.

Return Values

The function returns one of parameter type codes used with the **param_process** function. The code specifies the type of task parameter number *pnum*.

Description

The function **param_type** returns the type of task parameter number *pnum*. Parameters are numbered starting with parameter one. The returned value is a parameter type code, one of the codes used with the **param_process** function.

This function is typically used for supplementary parameter type checking after the **param_process** function has limited the possibilities. For example, if function **param_process** allows T_PIPE_W or T_PIPE_L for a parameter, the **param_type** function can be used to distinguish between these types.

See Also

param_process

pbuf_get

Get a block of data from a pipe.

```
unsigned int  pbuf_get (
   PBUF  *inbuf                          // Pipe buffer handle
   );
```

Parameters

inbuf

Pointer variable containing a handle for the pipe buffer control block to be used.

Return Values

Returns the number of values that were fetched from the pipe into the pipe buffer storage.

Description

The function **pbuf_get** reads a block of data from a pipe into the data array of pipe buffer *inbuf.* The pipe buffer control block is associated with a data source pipe by the function **pbuf_open**..

The values pbuf_max_cnt and pbuf_min_cnt of the PBUF must satisfy the following restrictions:

```
0 < pbuf_max_cnt <= MAX_BUF
0 <= pbuf_min_cnt <= pbuf_max_cnt
```

MAX_BUF is the maximum data array size; this is selected when a pipe buffer control block is allocated by **pbuf_open**.

The function **pbuf_get** automatically sets pbuf_cnt to the number of data values read into the data array. The value can be accessed at a later time using function **pbuf_get_cnt**, but in most cases it is more convenient to use the function return value.

The pbuf_max_cnt and pbuf_min_cnt are used by **pbuf_get** to determine how many values should be read into the data array. The function **pbuf_get** transfers a maximum of pbuf_max_cnt values from the input pipe to the data array. If the input pipe contains less than pbuf_min_cnt values, **pbuf_get** suspends the task until sufficient data values are available in the input pipe.

If `pbuf_min_cnt` is zero, the function **`pbuf_get`** returns to the caller regardless of whether any data were available in the associated pipe. This feature is useful to avoid suspending execution of the task when no data are available. When using this feature, be especially careful to check for zero available items by inspecting the return value or calling the **`pbuf_get_cnt`** function.

The function **`pbuf_get`** overwrites old data in the data array.

See Also

`pbuf_get_cnt`, **`pbuf_open`**

pbuf_get_cnt

Determine the current count of a pipe buffer.

```
unsigned int pbuf_get_cnt (
   PBUF  *pb                                  // Pipe buffer handle
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be examined.

Return Values

The function returns the current count field of a pipe buffer control block.

Description

The function **pbuf_get_cnt** obtains the current count field of a pipe buffer control block. The current count contains the number of valid data values in the pipe buffer's data array. This function is typically called after calling the function **pbuf_get** to determine the number of values that have been obtained from the associated pipe.

See Also

pbuf_get

pbuf_get_data_ptr

Get a pointer to the data array of a pipe buffer.

```
void *pbuf_get_data_ptr (
   PBUF  *pb                              // Pipe buffer handle
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be examined.

Return Values

There is no return value.

Description

The function **pbuf_get_data_ptr** returns a pointer to the storage array area of a pipe buffer control block. The returned pointer should be cast to an appropriate pointer type depending on the type of data.

This function is commonly used to obtain direct access to data read into the pipe buffer storage array by the function **pbuf_get**.

See Also

pbuf_get

pbuf_get_max_cnt

Determine the maximum pipe buffer count.

```
unsigned int pbuf_get_max_cnt (
   PBUF  *pb                                  // Pipe buffer handle
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be examined.

Return Values

The function returns the maximum number of items that can be read into the pipe buffer control block's data array by the function **pbuf_get**.

Description

The routine **pbuf_get_max_cnt** reports the maximum number of items that can be read into the pipe buffer's data array by the function **pbuf_get**. The maximum count field is initialized to the size of the pipe buffer's data array by **pbuf_open**.

The function **pbuf_get_max_cnt** is typically used to obtain information about a buffer control block that has been initialized previously, so that it is not necessary to maintain separate information about storage sizes of various PBUF structures.

See Also

pbuf_get, **pbuf_open**

pbuf_get_min_cnt

Determine the minimum pipe buffer count.

```
unsigned int pbuf_get_min_cnt (
   PBUF  *pb                              // Pipe buffer handle
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be examined.

Return Values

The function returns the minimum number of items that can be read into the pipe buffer control block's data array by the function **pbuf_get**.

Description

The routine **pbuf_get_min_cnt** reports the minimum number of items that can be read into the pipe buffer's data array by the function **pbuf_get**. The minimum count field is initialized to 1 by **pbuf_open**.

The function **pbuf_get_min_cnt** is typically used to obtain information about a buffer that has been initialized previously, so that it is not necessary to maintain separate information about storage sizes of various PBUF structures.

See Also

pbuf_get, **pbuf_open**, **pbuf_get_max_cnt**

pbuf_open

Open a pipe buffer.

```
PBUF *pbuf_open (
   PIPE  *pipe,                          // Pipe handle
   unsigned int bufsize
   );
```

Parameters

pipe

Pointer variable containing a handle for the associated pipe.

bufsize

The maximum number of data values that the array can hold.

Return Values

The function returns a pointer containing a handle to a PBUF control structure.

Description

The function **pbuf_open** allocates a pipe buffer control block and a data array for pipe *pipe*. The size of the data array is determined by *bufsize*, which specifies the maximum number of data values the array can hold. Therefore, the size of the data array, in bytes, is given by

```
bufsize * pipe_width(pipe)
```

The function **pbuf_open** also initializes three properties:

```
pbuf_cnt = 0
pbuf_min_cnt = 1
pbuf_max_cnt = bufsize
```

These initializations mean that the data array initially contains no data, at least one datum should be placed into the buffer when fetching data from a pipe, and no more than *bufsize* items can be placed into the data array storage area at any time.

If a buffer size of zero is passed to **pbuf_open**, the data management portion of a pipe buffer control block is allocated, but storage for the data array is not allocated.

Separate operations must be performed to obtain storage and complete initialization before the PBUF is used to transfer data into or out of a pipe. The functions **pbuf_set_data_ptr**, **pbuf_set_max_cnt**, and **pbuf_set_min_cnt** must be called to complete the initialization.

Note: Pipe *pipe* must be opened using **pipe_open** before **pbuf_open** is called.

See Also

pbuf_set_data_ptr, **pbuf_set_max_cnt**, **pbuf_set_min_cnt**, **pipe_open**

pbuf_put

Write a block of data to a pipe.

```
void pbuf_put (
   PBUF  *outbuf                          // Pipe buffer handle
   );
```

Parameters

outbuf

Pointer variable containing a handle for the pipe buffer control block used.

Return Values

There is no return value.

Description

The function **pbuf_put** writes a block of data from the data array of *outbuf* to a pipe.

The pipe buffer contains a field that points to the pipe to be written. This field is initialized by the function **pbuf_open**.

The function **pbuf_put** requires that the pbuf_cnt field be set to the number of data values to transfer. In most cases, it is more convenient to combine the operation of setting the count and performing the transfer using the function **pbuf_put_set_cnt** instead of **pbuf_put**. Before returning, **pbuf_put** sets the pbuf_cnt field to zero.

If **pbuf_put** cannot add the required number of values to the pipe because the capacity of the pipe has been reached, the calling task either goes to sleep until the pipe has room or throws away any excess data and returns immediately. The choice is determined by the pipe's WAIT /NOWAIT property. This property can be set when the pipe is defined using the DAPL command PIPES.

See Also

pbuf_open, pbuf_put_set_cnt

pbuf_put_set_cnt

Write a block of data to a pipe, where the amount of data in the block is specified.

```
void pbuf_put_set_cnt (
   PBUF  *outbuf                          // Pipe buffer handle
   unsigned int  *count

   );
```

Parameters

outbuf

Pointer variable containing a handle for the pipe buffer control block used.

Return Values

There is no return value.

Description

The function **pbuf_put_set_cnt** writes a data block of specified size from the data array of *outbuf* to a pipe.

This function is in effect a combination of the function **pbuf_set_cnt** and the function **pbuf_put.** This combined operation is often very convenient to use in practice. When the data is in-place in the buffer and the data count is readily available, using this function saves the overhead of an additional service call. See the descriptions of these two functions for additional details.

See Also

pbuf_open,pbuf_put, pbuf_set_cnt

pbuf_set_cnt

Set the current count field of a pipe buffer.

```
void pbuf_set_cnt (
   PBUF  *pb,                          // Pipe buffer handle
   unsigned int count
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be modified.

count

The number of data in the data storage array of the PBUF.

Return Values

The function has no return values.

Description

The function **pbuf_set_cnt** places a number into the current count field of a pipe buffer control block. This function is typically called after copying data into the PBUF storage area, but before calling the function **pbuf_put**, to inform the system of the number of items available for transfer to the associated pipe.

See Also

pbuf_put

pbuf_set_data_ptr

Assign a data storage array to a pipe buffer handle.

```
void  pbuf_set_data_ptr (
   PBUF *pb,                          // Pipe buffer handle
   void *data                         // Pointer to data storage array
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer to be examined.

data

Pointer to a data storage array.

Return Values

There is no return value.

Description

pbuf_set_data_ptr assigns a data storage array to a pipe buffer. The *data* array assigned by **pbuf_set_data_ptr** can be type `int`, `long`, or `float`, and should be consistent with the type of data in the pipe.

This function is commonly used to share data buffer storage between a `PBUF` for reading data from one pipe and another `PBUF` for writing data to a second pipe. Typically, the function **pbuf_get_data_ptr** is used to obtain the address of the storage area for the first pipe buffer, then the function **pbuf_set_data_ptr** assigns that pointer value to the second pipe buffer. When a storage area is assigned, it is also necessary to adjust the internal buffer size fields by calling the **pbuf_set_min_cnt** and **pbuf_set_max_cnt** functions.

See Also

pbuf_set_max_cnt, **pbuf_set_min_cnt**, **pbuf_get_data_ptr**

pbuf_set_max_cnt

Set the maximum pipe buffer count.

```
void  pbuf_set_max_cnt (
   PBUF  *pb,                          // Pipe buffer handle
   unsigned int count
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be modified.

count

The maximum number of items that can be read into the pipe buffer control block's data array.

Return Values

The function specifies the maximum number of items *count* that can be read into the pipe buffer's data array by the function **pbuf_get**.

Description

The routine **pbuf_set_max_cnt** specifies the maximum number of items *count* that can be read into the pipe buffer's data array by the function **pbuf_get**. The maximum *count* field is initialized to the size of the pipe buffer's data array by **pbuf_open**.

The function **pbuf_set_max_cnt** is typically used to limit the number of items to be read for a specific purpose, even though a larger storage area is available for other purposes. The specified maximum must be greater than or equal to the minimum count specified for the PBUF, but must never exceed the amount of storage available in the PBUF data storage area.

See Also

pbuf_open, **pbuf_get**, **pbuf_set_min_cnt**

pbuf_set_min_cnt

Set the minimum pipe buffer count.

```
void  pbuf_set_min_cnt (
   PBUF  *pb,                          // Pipe buffer handle
   unsigned int count
   );
```

Parameters

pb

Pointer variable containing a handle for the pipe buffer control block to be modified.

count

The minimum number of items that can be read into the pipe buffer control block's data array.

Return Values

The function sets the minimum number of items *count* that can be read into the pipe buffer's data array by the function **pbuf_get**.

Description

The routine **pbuf_set_min_cnt** sets the minimum number of items *count* that can be read into the pipe buffer's data array by the function **pbuf_get**. The minimum count must not be negative and must never exceed the amount of storage available in the PBUF storage array or the limit set by the function **pbuf_set_max_cnt**.

The function **pbuf_set_min_cnt** is typically used to obtain data in fixed block sizes rather than whatever amounts happen to be available. Fetching fixed-size blocks also requires calling the routine **pbuf_set_max_cnt** to set the minimum count and the maximum count equal.

See Also

pbuf_get, **pbuf_set_max_cnt**

pid_compute

Compute new PID state and output.

```
int pid_compute (pid, val)
   PID *pid,                          // PID control block handle
   short int val
   );
```

Parameters

pid

Pointer variable containing a handle for the PID control block to be adjusted.

val

Value of the sample.

Return Values

The function returns the value of the PID control output.

Description

The function **pid_compute** performs the real-time computation of PID control output. This function is called once for each captured sample of the controlled system's output. The value of the sample is *val*. The internal state of the PID computation is maintained in the *pid* structure. The **pid_compute** function returns the value of the PID control output.

Note: For controlled systems that have inverting inputs, the negative of the value returned by **pid_compute** should be used as the final control output. See the **pid_tune** function description for more information.

Note: The **pid_tune** function must be called to establish values of the PID parameters before the **pid_compute** function is called.

See Also

pid_tune

pid_open

Open and initialize a PID control block.

```
PID *pid_open (
    short int val
    );
```

Parameters

val

A value used to initialize PID computations. This value is an estimate or sample of the controlled system output.

Return Values

The function returns a pointer containing a handle to a PID control structure.

Description

The function **pid_open** allocates a PID control structure and returns a handle for that structure. The estimated initial value *val* of the controlled system's output is used to initialize PID computations.

If a good estimate for *val* is not available, a sample of the output of the controlled system can be used as the initialization value. Some systems start from a "zero state," and for these systems, a constant zero value can be specified.

Note: **pid_open** must be called before the **pid_tune**, **pid_set_setpoint**, or **pid_compute** functions are called.

See Also

pid_compute, **pid_set_setpoint**, **pid_tune**

pid_preset

Establish a pre-determined PID operating state.

```
int pid_preset (
   PID  *pid,                          // PID control block handle
   short int sysval,
   short int ctrlval
   );
```

Parameters

pid

Pointer variable containing a handle for the PID control block to be adjusted.

sysval

A value specifying the feedback from the controlled system's output.

ctrlval

A value specifying the PID control output level required as input to the system to sustain the system output level at *sysval*.

Return Values

If the function succeeds, the return value is 0.

If the function fails, the return value is an non-zero error code.

Description

The function **pid_preset** establishes a pre-determined PID operating state.

The *sysval* parameter specifies the feedback from the controlled system's output. The *ctrlval* parameter specifies the PID control output level required as input to the system to sustain the system output level at *sysval*. The gain, setpoint, and limit settings are obtained from the PID structure specified by the *pid* parameter. An internal state for the PID controller is computed and stored into the PID structure.

This function is typically used when PID control action is not applied initially, but some other control strategy is applied, so that current input and output conditions for the system are known.

Suppose that the *sysval* and *ctrlval* parameters correspond to steady state operating conditions for the controlled system, and that the PID structure's setpoint

parameter is equal to *sysval*. Then, after successful completion of this function, the **pid_compute** function will produce the output value *ctrlval* when the system feedback value *sysval* is applied. In other words, the PID control is also at a steady state, consistent with the state of the controlled system.

The PID control setpoint may also be set to a value different from the *sysval* parameter. In this case, the PID operation starts at the specified state, but begins a smooth control transient to move the system from *sysval* to the new setpoint specified in the PID parameters.

This function returns the value 0 if computations are successful. It returns a nonzero error code if an internal PID operating state cannot be computed to produce the specified *ctrlval* level given the specified *sysval* input. The PID structure is not updated unless the computation is successful.

There are two possible causes for unsuccessful completion and a nonzero error code. The first is that the output limit clamp parameters prohibit the *ctrlval* level specified in the call to this function. The other possibility is that the integral-correction coefficient is zero. Unless there is an absolute guarantee that the *ctrlval* parameter is within the application's limits, and the integral coefficient is nonzero, the custom command should check the error code and report errors to the application on the host computer.

Note: PID parameters must be established using the **pid_tune** function before calling **pid_preset**. The setpoint may be separately adjusted by calling **pid_set_setpoint**.

See Also

pid_tune, **pid_set_setpoint**, **pid_compute**

pid_set_setpoint

Assign a PID setpoint.

```
void pid_set_setpoint (
   PID *pid,                          // PID control block handle
   short int val
   );
```

Parameters

pid

Pointer variable containing a handle for the PID control block to be adjusted.

val

Setpoint value for PID structure

Return Values

There is no return value.

Description

The function **pid_set_setpoint** assigns a new setpoint value *val* to the PID structure identified by handle *pid*.

Note: This function must be called after the function **pid_tune** is called. The **pid_tune** function will initialize all PID control parameters, including the setpoint. It is not necessary to call **pid_set_setpoint** unless the setpoint is changed after parameter initialization.

See Also

pid_tune

pid_tune

Set PID coefficients.

```
int pid_tune (
   PID  *pid,                    // PID control block handle
   PIDCOEF  *coef                // Pointer to coefficient sets
   );
```

Parameters

pid

Pointer variable containing a handle for the PID control block to be adjusted.

coef

Pointer to the PIDCOEF structure from which control parameters are obtained.

Return Values

If the function succeeds, the return value is zero.

If the function fails, an nonzero error code is returned. The error code is one of the following:

0	- successful installation of coefficients
2	- upper and lower output clamp values reversed
4	- the i1 integral correction multiplier is too large, outside the range (-8192, 8191)
8	- warning, the P, I, and D terms are all zero

Description

The function **pid_tune** installs the parameter values from the *coef* structure into PID control structure *pid*. If any parameter values are inconsistent or out of range, the new coefficients are not installed and a nonzero value is returned by **pid_tune**. A return value of zero indicates successful installation.

The *coef* parameter points to the PIDCOEF structure from which control parameters are obtained. All fields in the structure have signed integer type. The fields are:

```
coef->setpoint      - desired output of controlled system
coef->p1            - multiplier for proportional correction
coef->p2            - divisor for proportional correction
coef->i1            - multiplier for integral correction
coef->i2            - divisor for integral correction
coef->d1            - multiplier for derivative correction
coef->d2            - divisor for derivative correction
coef->clamp_lo      - lower output limit
coef->clamp_hi      - upper output limit
```

The PID control output is given by the following equations:

```
          p1              i1              d1
    P =   -- ,      I =   -- ,      D =   -- ,
          p2              i2              d2

    correction = P * e  +  I * int(e)  +  D * d(e) ,

              | clamp_lo      if   correction  <  clamp_lo
    output =  | clamp_hi      if   correction  >  clamp_hi
              | correction    in all other cases
where

      e = <setpoint> - input
 int(e) = integral of e
   d(e) = derivative of e
```

The terms `P, I,` and `D` in the correction formula are specified by pairs of integer parameters. This allows representation of fractional numbers. The denominator terms `p2, i2,` and `d2` can be set to a convenient arbitrary value, such as 1000, and then the numerator values `p1,i1,` and `d1` can be adjusted to produce the desired control effects. The exact values of the parameters are not important, as long as the ratios are correct. A zero in a denominator term is treated the same as a zero in the numerator. There are some constraints on the ranges of the combined fractional values:

```
-256.0  < P <  256.0
-16.0   < I <  16.0
-256.0  < D <  256.0
```

The sign conventions for coefficients in the `PIDCOEF` structure are that a positive gain works to correct a positive error, consequently, a positive error results in a

reduced control output. For most controlled systems, this will reduce the level of error `e`. For some systems that have inverting inputs, either invert the signs on all gain terms or negate the output value computed by the function **`pid_compute.`**

Note: The **`pid_open`** function must be called to set up the `PID` structure before **`pid_tune`** function is called.

See Also

`pid_open`, **`pid_compute`**

pipe_get

Get a fixed point value from a pipe.

```
long int pipe_get (
   PIPE  *input                          // Pipe handle
   );
```

Parameters

input

Pointer variable containing a handle for the pipe to be examined.

Return Values

The function returns one value from a pipe. If the pipe has `int` data rather than `long int` data, the returned value can be cast to an `int` type.

Description

The function **pipe_get** reads one value from a pipe. If the pipe is empty when **pipe_get** is called, the calling task goes to sleep until the pipe contains data. If this behavior is not desired, function **pipe_num** or **pipe_num_complete** should be used first to determine whether the pipe contains data.

This function is provided primarily for backward compatibility, because it cannot be extended to work with all data types. The function **pipe_value_get** is the recommended choice, because it is compatible with all pipe data types.

See Also

pipe_value_get, **pipe_num**, **pipe_num_complete**

pipe_num

Determine whether a pipe contains data.

```
unsigned int pipe_num (
   PIPE  *pipe                          // Pipe handle
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to be examined.

Return Values

The function returns a number indicating a lower bound for either the number of data values in a pipe or the number of locations available for writing to a pipe.

Description

When applied to a pipe *pipe* which is opened as an input pipe, the routine **pipe_num** returns a lower bound for the number of data values in a pipe. When applied to a pipe *pipe* which is opened as an output pipe, the routine **pipe_num** returns a lower bound on the number of locations available for writing into the pipe.

This function should be used with care, since polling a pipe for data can slow an application. Also, there are subtle differences in behavior for different types of pipes. The behavior is regular and predictable for user-defined pipes, but can be non-intuitive for input, output and communication pipes.

There is no guarantee of the accuracy of the number returned by this function when used with input channel pipes. For example, function **pipe_num** could report a value such as 4 when, in fact, thousands of samples are available. Furthermore, the reported value does not necessarily improve as more data are written into the pipe. The number returned by this function should be treated as a 'Boolean' value. If it is nonzero, the reported number of values can be fetched safely.

Similarly, there is no guarantee in the utility of the returned value for output pipes. It is a lower bound, not an upper bound. For example, when the output pipe is an output channel pipe being used by an active output procedure, a value of zero could be returned. This value is meaningless; it says that there is no information about how

much space is available. It definitely does not mean that the synchronous output pipe cannot accept data.

If an accurate count of the number of samples available to read from an input pipe is required, the function **pipe_num_complete** should be used instead.

See Also

pipe_num_complete

pipe_num_complete

Return an accurate estimate of the number of data in a pipe.

```
unsigned int pipe_num_complete (
   PIPE *pipe,                          // Pipe handle
   unsigned count
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to be examined.

count

A value specifying the maximum number of samples

Return Values

The function returns an accurate estimate of the current number of samples available in pipe *pipe*, up to a maximum of *count* samples.

Description

The function **pipe_num_complete** returns an accurate estimate of the current number of samples available in pipe *pipe*, up to a maximum of *count* samples.

Except for additional samples which may appear in the pipe between the time this function starts and the time that it ends, the number returned by this function is accurate. This function performs a thorough search of a pipe's data structure to obtain this estimate. For maximum speed, the *count* parameter should be as small as possible.

The function returns when the entire pipe structure has been processed or when *count* values have been found. A call to **pipe_num_complete** on an input channel pipe is usually slower than a call to **pipe_num**. For other pipe types, **pipe_num** and **pipe_num_complete** produce equivalent results.

See Also

pipe_num

pipe_open

Open a pipe.

```
void pipe_open (
   PIPE  *pipe,                          // Pipe handle
   int mode
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to be opened for input or output.

mode

P_READ if the pipe is used for input and P_WRITE if the pipe is used for output.

Return Values

There is no return value.

Description

The function **pipe_open** prepares a pipe for input or output; *mode* must be P_READ if the pipe is used for input and P_WRITE if the pipe is used for output.

pipe_purge

Remove all data from a pipe.

```
void pipe_purge (
   PIPE  *pipe                          // Pipe handle
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to be examined.

Return Values

There is no return value. The function removes all data values from a pipe.

Description

The function **pipe_purge** removes all data values from a pipe. This function is not recommended in newer applications, as it removes data for all tasks reading from the pipe. Newer applications should use the function **pipe_rem** to empty a pipe. For example:

```
while (count = pipe_num_complete(pipe,100))
    pipe_rem (pipe, count);
```

pipe_put

Put a data value into a pipe.

```
void pipe_put (
   PIPE  *pipe,                          // Pipe handle
   long int val
   );
```

Parameters

pipe
Pointer variable containing a handle for the pipe to receive the value.

val
Value to be added to a pipe.

Return Values

There is no return value.

Description

The function **pipe_put** adds a data value to a pipe. If the pipe is full, the task either goes to sleep until the pipe has room, or throws out the data and returns immediately. Whether the task goes to sleep or returns immediately is selected by the WAIT/NOWAIT property when the pipe is defined using a PIPES command in the DAPL system configuration.

pipe_rem

Remove a fixed number of data values from a pipe.

```
void pipe_rem (
   PIPE *pipe,                          // Pipe handle
   unsigned int  num
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe from which data is removed.

num

Number of data values to be removed from the pipe.

Return Values

There is no return value.

Description

The function **pipe_rem** removes *num* data values from a pipe. If the pipe contains less than *num* values, the calling task goes to sleep until all data values become available and then have been removed.

pipe_value_get

Get a value from a pipe.

```
void  pipe_value_get (
   PIPE  *input                          // Pipe handle
   GENERIC_SCALAR  *value                // Storage for value
   );
```

Parameters

input

Pointer variable containing a handle for the source pipe.

value

A generic storage location where the fetched value is returned.

Return Values

The function returns one value from a pipe. The value may have any supported data type.

Description

The function **pipe_value_get** reads one value from a pipe. If the pipe is empty when **pipe_value_get** is called, the calling task goes to sleep until the pipe contains data. If this behavior is not desired, function **pipe_num** or **pipe_num_complete** should be used first to determine whether the pipe contains data.

See Also

pipe_get, **pipe_num**, **pipe_num_complete**

pipe_value_put

Put a value into a pipe.

```
void  pipe_value_put (
   PIPE  *input                    // Pipe handle
   GENERIC_SCALAR  *value          // Storage for value
   );
```

Parameters

input

Pointer variable containing a handle for the receiving pipe .

value

A generic storage location where the value is obtained.

Return Values

There is no return value.

Description

The function **pipe_value_put** takes the specified value and copies that value to the specified data pipe. If the pipe is full, the task either goes to sleep until the pipe has room, or throws away the data and returns immediately. Whether the task goes to sleep or returns immediately is selected by the WAIT/NOWAIT option when the pipe is defined using a PIPES command in the DAPL system configuration.

This function works with data of any type.

See Also

pipe_value_get, **pipe_num**, **pipe_num_complete**

pipe_width

Return the size in bytes of data elements from a pipe.

```
int pipe_width (
   PIPE *pipe                          // Pipe handle
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to be examined.

Return Values

The size of a data element from the pipe, in bytes.

Description

The function **pipe_width** returns the size of a data element from a pipe, in bytes. The length of the pipe determines how many elements the pipe buffer can contain, and the width of the pipe determines how large each individual item can be. The width is one for a byte pipe, two for a word pipe, four for a long pipe or a float pipe, eight for a double pipe.

printf

Format and print a string.

```
int printf (
   char *format_string,          // Pointer to character string
   ...                           // Additional parameters
   );
```

Parameters

format_string

ASCII character string controlling the conversions performed by **printf**.

...

A variable number of optional parameters appearing after the mandatory conversion text parameter.

Return Values

The function **printf** returns the number of characters sent to output pipe $SYSOUT.

Description

The function **printf** formats characters and numeric values into a string and sends the string to the output pipe $SYSOUT. This function does the same things as the Standard C function, except that the output is sent to a DAPL pipe instead of a STDOUT stream.

The string *format_string* consists of printable ASCII characters controlling the conversions performed by **printf**. All ANSI Standard C conversion codes are supported except for the long long and long double conversions and types.

To control stack requirements, there is a 132-character limit on the length of the final formatted string. Be particularly careful not to format a very large floating point number using the %f format conversion code. Unlike the Standard C versions of this function, if the resulting text does not fit into the specified field size, the field is filled with asterisk characters in the manner of Basic or FORTRAN.

See Also

sprintf

ralloc

Dynamically allocate bulk storage.

```
char *ralloc (
   unsigned int  size
   );
```

Parameters

size

The size, in bytes, of the storage to be allocated to a task.

Return Values

The function returns a pointer to the block of allocated storage. If insufficient memory is available, **ralloc** displays an error message and the calling task is stopped.

Description

The function **ralloc** allocates storage to a task and returns a pointer to this storage. De-allocation is performed automatically when a STOP command is issued or when function **free** is called. The storage size is guaranteed to be at least *size* bytes. The storage can exist in a pooled storage area for efficiency, so it is possible that more than *size* bytes are physically addressable. However, it is essential to access only the amount of storage allocated to avoid corrupting task and system data.

The function **ralloc** is preferred rather than the function **malloc** when initially setting up data structures for task operation. If memory is unavailable to initialize the task cannot run and some cleanup must be performed in the DAPL system configuration. In this situation, function **ralloc** will display an error message and terminate the task rather than trying to continue in an impossible situation.

See Also

free, **malloc**, **rfree**

realloc

Dynamically resize allocated bulk storage.

```
void * realloc (
   void  *oldstore
   unsigned  size
   );
```

Parameters

oldstore
The original allocated storage.

size
The size, in bytes, for the modified storage allocation.

Return Values

The function returns a pointer to the revised block of allocated storage, or a NULL pointer if insufficient memory is available or no memory is requested.

Description

The function **realloc** can be considered a combined storage allocation and deallocation operation. It is completely compatible with the similarly-named function in Standard C.

A new allocation is established that has the specified size. If the specified size is zero, the operation acts like a deallocation operation for the old store allocation, and function **realloc** returns a `NULL` pointer. Otherwise, a storage area of sufficient length is allocated, and a pointer to this area is returned. If the allocation fails, the original allocation is unchanged and the returned pointer is `NULL`.

After establishing a new storage area, copy operations are performed to preserve the data from the original storage. If the new allocation is larger, all of the original data is retained and any additional storage locations are indeterminate. If the new allocation is smaller, only the first part of the original data is retained, as much as will fit in the new storage size.

After allocation and copy operations are completed, the original allocation is released.

There is no guarantee that the original and the new pointers are different, as the memory management system could map different physical memory or the same physical memory locations to the old pointer value or to a new pointer value. There is no guarantee that if the pointer is the same the old memory previously present is still present and not reassigned to another purpose.

Final deallocation is performed automatically when a `STOP` command is issued or when a free function is called. The storage can exist in a pooled storage area for efficiency, so it is possible that more than *`size`* bytes are physically addressable. However, it is essential to access only the amount of storage allocated to avoid corrupting task and system data.

See Also

`malloc`

rfree

Release dynamically allocated task storage.

```
void rfree (
   void * storage
   );
```

Parameters

storage

The pointer to storage previously allocated by function **ralloc**.

Return Values

None.

Description

The function **rfree** releases storage allocations previously obtained using the function **ralloc**. Do not call this function for memory obtained by any other means.

See Also

malloc, **ralloc**, **free**

sprintf

Format a string.

```
int sprintf (
   char  *str,                    // Pointer to character string
   char  *format_string,          // Pointer to character string
   ...                            // Additional parameters
   );
```

Parameters

str

Pointer to a data storage character string.

format_string

ASCII characters controlling the conversions performed by **sprintf**.

...

A varying number of optional parameters appearing after the mandatory parameters.

Return Values

The function **sprintf** returns the number of characters stored in *str*.

Description

The function **sprintf** formats characters and values into the string *str*. This function is the equivalent of the Standard C *sprintf* function. All ANSI Standard C conversion codes are supported except for long long and long double conversions and types.

Unlike the Standard C versions of this function, if the resulting text does not fit into the specified field size, the field is filled with asterisk characters in the manner of Basic or FORTRAN.

See Also

printf

sscanf

Scan a string, converting recognized values and assigning them to variables.

```
int sscanf (
   char *str,                    // Pointer to character string
   char *format_string,          // Pointer to character string
   ...                           // Additional parameters
   );
```

Parameters

str

Pointer to a character string.

format_string

Printable ASCII characters controlling the conversions performed by **sscanf**.

...

A varying number of pointer parameters appearing after the mandatory parameters.

Return Values

The function returns the number of items matched and assigned.

Description

The function **sscanf** scans text string *str* under control of *format_string*, converting values which it recognizes, and assigning them to variables using pointers provided by a varying-length parameter list. This function is compliant with the Standard C version of the *sscanf* function, except that the long long and long double conversion codes and the long long and long double pointer types are not supported.

Note: This function is dangerous in any implementation. Be very careful that data types correspond exactly to the types implied by the conversion codes, each conversion code has a corresponding pointer in the varying portion of the parameter list, and there is sufficient storage available to receive the formatted string.

sys_exec_command

Send a DAPL command to the DAPL system command interpreter.

```
void sys_exec_command (
   char * command                          // Pointer to a character string
   );
```

Parameters

command

A pointer to a DAPL command text in a character string. The string must be a null terminated ASCII string containing no control characters. Multiple commands are not allowed in the string.

Return Values

There is no return value. Errors might be diagnosed by the command interpreter.

Description

The function **sys_exec_command** sends a DAPL command string to the DAPL command interpreter. DAPL will interpret a command sent by **sys_exec_command** when there are no other commands pending in the default text input pipe. For example, suppose a downloaded DAPL file specifies several processing procedures and a sequence of START, PAUSE, and STOP commands to run those procedures. Then, no **sys_exec_command** messages sent by custom commands are executed until the last operation specified in the downloaded DAPL file is completed.

sys_get_info

Return DAPL system information.

```
long int sys_get_info (
   int info_code
   );
```

Parameters

info_code

A value representing the request code for system information.

Return Values

The function returns DAPL system information selected by the request code parameter in a `long` representation.

Description

The function **`sys_get_info`** returns DAPL system information. The return information is selected by the request code parameter. The information contained in the value returned by **`sys_get_info`** may be a word constant, a long constant, or a pointer. The return value should be cast to the appropriate data type.

The codes are defined in the file `DTDCNSTS.H` and are included automatically when the `DTD.H` file is included in each source code file. The following table summarizes the request codes and return types:

Request Code	Return Type

```
GI_DECIMAL              int
GI_TERMINAL             int
GI_OVERQ                int
GI_IBIPOLAR             int
GI_OBIPOLAR             int
GI_FLOAT_ERROR          int
GI_ROUNDING             int
GI_AINEXPAND            int
GI_IN_ACTIVE            int
GI_OUT_ACTIVE           int
GI_IN_CNT               unsigned int
GI_OUT_CNT              unsigned int
GI_ICHAN_CNT            int
GI_DEFAULT_BUF_SIZE     int
GI_SYSOUT               PIPE *
GI_SYSIN                PIPE *
GI_HMEMAVL              unsigned int
GI_HMEMSIZE             unsigned int
GI_TMEMAVL              unsigned int
GI_TMEMSIZE             unsigned int
GI_SERIAL               unsigned int
GI_OEM_ID               int
GI_FFTSIZE              int
GI_IBURST_ACTIVE        int
GI_OBURST_ACTIVE        int
GI_BUFFERING            int
GI_SCHEDULE_MODE        int
GI_QUANTUM              int
```

The following request codes return a nonzero value if ON, a zero value if OFF: `GI_DECIMAL`, `GI_TERMINAL`, `GI_OVERQ`, `GI_IBIPOLAR`, `GI_OBIPOLAR`, `GI_IN_ACTIVE`, `GI_OUT_ACTIVE`, `GI_FLOAT_ERROR`, `GI_ROUNDING`, `GI_AINEXPAND`, `GI_IBURST_ACTIVE`, `GI_OBURST_ACTIVE`, and `GI_OPTIMIZE`.

The `GI_IN_ACTIVE` and `GI_OUT_ACTIVE` request codes indicate whether an input or an output procedure currently is started. `GI_IN_ACTIVE` and `GI_OUT_ACTIVE` do not indicate whether the procedure is capturing or updating samples when operating in burst mode. That information can be obtained using the `GI_IBURST_ACTIVE` and `GI_OBURST_ACTIVE` request codes.

The `GI_IN_CNT` and `GI_OUT_CNT` request codes return the current sample count of an active input procedure and the current output count of an active output procedure.

The sample count is undefined when no input procedure is active. The output count is undefined when no output procedure is active.

The GI_ICHAN_CNT request code returns the number of channels in the currently active input procedure. A returned value of zero indicates that no input procedure is active.

The GI_DEFAULT_BUF_SIZE request code returns a suggested number of data elements for PBUF storage. This number is used by the DAPL system's processing commands, and it is a good choice for custom commands which accept buffered data from other processing commands, or which write buffered data to other processing commands.

The GI_SYSOUT and GI_SYSIN request codes return pointers to DAPL system pipes, $SYSOUT and $SYSIN.

The GI_HMEMAVL and GI_HMEMSIZE requests codes return the size of available system heap storage, in bytes, and the total size of the system heap area, in bytes. The GI_TMEMAVL and GI_TMEMSIZE request codes return the size of available system memory, in bytes, and the total size of the system memory area, in bytes. The system memory area includes both the heap storage area and the data buffer areas.

The GI_SERIAL request code returns the serial number of the Data Acquisition Processor. The GI_OEM_ID returns the optional OEM code number for the DAPL configuration.

The GI_FFTSIZE code reports the current size limit on an FFT. Use the DAPL command OPTION FFTSIZE to adjust the limit. In most cases it is best to adjust OPTION FFTSIZE when downloading rather than when running the custom command.

The GI_QUANTUM code reports the length of the task scheduling quantum in microseconds. The GI_SCHEDULE_MODE code returns one of the codes eSchedFixed or eSchedAdaptive. The eSchedFixed and eSchedAdaptive codes are defined in the file DTDCNSTS.H and are included automatically when the DTD.H file is included in each source code file.

Note: The file CDAPCC.H may contain other request codes -- these are reserved for future expansion or backward compatibility.

---

sys_get_time

Return the elapsed time since the Data Acquisition Processor was powered on.

```
unsigned long int sys_get_time (
  );
```

Parameters

This function requires no parameters.

Return Values

The function **sys_get_time** returns the number of milliseconds since the Data Acquisition Processor was powered on.

Description

The function **sys_get_time** reports the elapsed time in milliseconds since power-up of the Data Acquisition Processor. This elapsed time is derived from the hardware CPU clock and provides good long-term accuracy. Because of the 32-bit representation, the timing interval wraps back to 0 in approximately 50 days. See the Data Acquisition Processor Hardware manual for information about clock accuracy.

sys_get_version

Obtain descriptions of software and hardware versions for the DAPL environment where the task is running.

```
void  sys_get_version (
   DAP_VERSION  *version,
   );
```

Parameters

version

Pointer to a special structure where pointers to descriptive text can be placed.

Return Values

There is no return value. Results are accessed through pointers placed into the *version* structure.

Description

The **sys_get_version** function returns descriptions of the software and hardware versions for the Data Acquisition Processor and DAPL operating system running the task.

To request the information, the caller must declare and initialize a special DAP_VERSION structure. This data type is defined in the file DTDTYPES.H and included automatically when the DTD.H file is included in each source code module. The infosize field of this structure must be initialized to sizeof(DAP_VERSION).

When the function **sys_get_version** is called, the DAPL system completes the structure by storing pointers to text strings describing the DAP and DAPL systems:

- daplname — name of the operating system
- daplver — version of the operating system
- dapmodel — model name of the DAP
- daprev — revision number of the DAP

Each description has the form of a C-style terminated text string. For example:

```
{
  DAP_VERSION  dver;
  dver.infosize = sizeof(struct _dap_version);
  sys_get_version( &dver );
  printf( “DAP model is: ‘%s’ \n”,  dver.dapmodel )
}
```

This sequence allocates a temporary structure of auto storage class, initializes it, invokes the sys_get_version function to fill in the version information, displays the description of the DAP model, and then releases the temporary structure as it goes out-of-scope.

task_pause

Pause a task for a specified time.

```
void task_pause (
   int ms
   );
```

Parameters

ms

A value that represents the time in milliseconds that task execution is suspended.

Return Values

There is no return value.

Description

The function **task_pause** suspends execution of a task for *ms* milliseconds. After this time has elapsed, the Data Acquisition Processor continues execution of the task at the statement following the function **task_pause**.

Note: As a result of DAPL system multitasking, there can be several milliseconds of additional delay before a task continues after a call to **task_pause**, dependent on the processing configuration and option setting.

See Also

sys_get_time

task_switch

Temporarily suspend the task to allow other tasks to use the CPU.

```
void task_switch (
  );
```

Parameters

This function requires no parameters.

Return Values

There is no return value.

Description

The function **task_switch** temporarily suspends the task and allows other tasks to use the CPU. Execution resumes after some delay, at the next statement after the **task_switch** function. This function allows tasks to suspend their operation for reasons other than waiting for new data to arrive, usually to decrease latency of real-time response in a multi-tasking configuration. Most tasks can simply wait for data to arrive, and do not need to use this function to release the CPU.

trigger_get

Extract and return the next available trigger assertion.

```
unsigned long int  trigger_get (
   TRIGGER  *trig                          // Trigger handle
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

Return Values

The function returns the next available assertion from trigger *trig*.

Description

Function **trigger_get** extracts a trigger event from a trigger. Function **trigger_get** does not return until the requested assertion is available, and this can block execution of the calling task, leading to backlog conditions. In most situations, the **trigger_wait** or **trigger_get_immediate** functions should be used instead. However, **trigger_get** can be called safely after using the **trigger_num** function to verify that a trigger assertion is available, or when backlog situations cannot occur.

See Also

trigger_get_immediate, **trigger_wait**, **trigger_num**

trigger_get_immediate

Return the next available assertion or status report immediately.

```
unsigned long int  trigger_get_immediate (
   TRIGGER  *trig,                    // Trigger handle
   int *flag                          // Pointer to integer variable
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be examined.

flag

The address of an integer variable.

Return Values

The function fetches the next available trigger assertion, or if an assertion is not available, returns the status of the writer for the trigger. The meaning of the returned value is indicated by the *flag* variable:

- *flag* is zero (logical false) if no assertion is present in the trigger structure, and the returned value is a status,
- *flag* is nonzero (logical true) if an assertion is was extracted from the trigger and sent as the returned value.

Description

The function **trigger_get_immediate** fetches the next available assertion event from the specified trigger, or if an assertion is not available, returns the status of the writer for this trigger. Whether the returned value is an assertion or status number is indicated by the contents of the integer variable *flag*.

Unlike the **trigger_get** or **trigger_wait** functions, which will cause the task to wait until a trigger assertion occurs, the **trigger_get_immediate** function avoids suspending the calling task.

Function **trigger_get_immediate** allows a trigger reading task to determine the state of the trigger writer task. If a status number is returned, the returned value specifies a sample number up to which it is guaranteed that no assertion occurs.

See Also

trigger_num, **trigger_get_status**, **trigger_get**, **trigger_wait**

trigger_get_opmode

Return a trigger's operating mode.

```
unsigned int  trigger_get_opmode (
   TRIGGER  *trig                          // Trigger handle
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

Return Values

The **trigger_get_opmode** function returns a code indicating the operating mode of trigger *trig*.

Description

The function **trigger_get_opmode** examines the operating mode defined for trigger *trig* in the DAPL configuration. The mode can be examined but not changed. For example, a custom command could be intended for single-event processing, and should only be used in a configuration with a trigger in TRIG_MANUAL_MODE. The **trigger_get_opmode** function allows the custom command to verify the trigger configuration.

The returned code will be one of the following:

```
TRIG_NATIVE_MODE
TRIG_MANUAL_MODE
TRIG_AUTO_MODE
TRIG_NORMAL_MODE
TRIG_DEFERRED_MODE
```

These codes are defined by the DTDCNSTS.H file and are included automatically when the DTD.H file is included in each source code module.

See Also

trigger_get_property

trigger_get_property

Return a trigger's property value.

```
unsigned long int  trigger_get_property (
   TRIGGER  *trig,                        // Trigger handle
   unsigned int  prop
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be examined.

prop

A code selecting a trigger property.

Return Values

The function returns the numerical value of the specified trigger property *prop* for trigger *trig*.

Description

The function **trigger_get_property** returns a trigger's property value. The property `prop` is an identifier from the following list. These codes are defined by the `DTDCNSTS.H` file and are included automatically when the `DTD.H` file is included in each source code module.

```
TRIG_HOLDOFF_PROPERTY
TRIG_CYCLE_PROPERTY
TRIG_STARTUP_PROPERTY
TRIG_GATE_PROPERTY
```

The returned numbers are the holdoff interval length, the auto-mode cycle length, startup interval length, or the `GATE` arming, respectively. `HOLDOFF`, `CYCLE`, and `STARTUP` intervals return unsigned long integer values. The `GATE` property is `ARMED` if the returned value is nonzero, or `DISARMED` if the value is zero. Only the `GATE` property can change after the trigger is defined.

Note: A processing command cannot directly change a trigger's `GATE` property. The property can be changed indirectly by sending a number to a `TRIGARM` task through

a pipe, or by sending an `EDIT` command to the DAPL system using the function **sys_exec_command**.

See Also

trigger_get_opmode, **sys_exec_command**

trigger_get_status

Return a trigger's current status count.

```
unsigned long int  trigger_get_status (
   TRIGGER  *trig                          // Trigger handle
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

Return Values

The function returns the current status count for the calling task.

Description

The function **trigger_get_status** gets a trigger's current status count. The status is different for each task accessing the trigger. The writer status describes the progress of the writer task scanning its data pipe for triggering. A reader status describes the progress of the reader task as it takes or discards samples from its data pipe.

Using this function, it is not necessary for the custom command to maintain a separate status count variable. This information can be obtained from the trigger as needed.

Note: The status information returned by the **trigger_get_status** function is different from the status information returned by the **trigger_get_immediate** function, which reports information about the progress of the trigger writer task to a trigger reader task.

See Also

trigger_get_immediate

trigger_num

Determine if an assertion is present.

```
unsigned int trigger_num (
   TRIGGER  *trig                          // Trigger handle
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

Return Values

For a trigger reader task, the function returns a nonzero number of assertions if an assertion is available in the trigger, or a zero value if no assertion is present. For a writer task, **trigger_num** reports the number of locations available for storing additional assertions in the trigger structure.

Description

The function **trigger_num** operates in the manner of the **pipe_num** function, except it tests trigger *trig* rather than a data pipe.

See Also

trigger_get_immediate, **pipe_num**

trigger_open

Initialize a trigger.

```
void trigger_open (
   TRIGGER  *trig,                    // Trigger handle
   int  mode
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

mode

The parameter *mode* must be P_WRITE to open the trigger for signaling assertions, or P_READ to open the trigger for receiving assertions.

Return Values

There is no return value.

Description

The function **trigger_open** initializes trigger *trig*. All tasks which use a trigger must call this function prior to calling other triggering functions. The codes P_WRITE and P_READ are defined by the DTDCNSTS.H file and are included automatically when the DTD.H file is included in each source code module.

trigger_put

Place an assertion into a trigger.

```
void trigger_put (
   TRIGGER *trig,                    // Trigger handle
   unsigned long int  sc
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

sc

The sample number for an asserted event.

Return Values

There is no return value.

Description

The function **trigger_put** generates a trigger assertion, writing sample number *sc* into trigger *trig*. The name trigger_assert is an alias for the function **trigger_put**. The status of the trigger is updated automatically to be consistent with the asserted sample number.

Note: The sequence of sample numbers written to the trigger must be a strictly increasing sequence.

trigger_set_status

Set a trigger's status field.

```
void trigger_set_status (
   TRIGGER *trig,                    // Trigger handle
   unsigned long int  sc
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

sc

A sample number indicating the task's progress in checking for events.

Return Values

There is no return value.

Description

The function **trigger_set_status** is used to set the status number of trigger *trig* to specified value *sc*. This informs the DAPL system that any samples or events with a lesser or equal sample number are no longer needed by this task. This function is useful for triggering commands which generate events at predetermined times, for example, automatic sweep generation. It is also useful for commands which copy status information from one trigger to another.

In most applications, it is safer and easier to compute an incremental change and apply the **trigger_updt_status** function instead.

Use of the **trigger_set_status** function is demonstrated in the TSTAMP2.CPP custom command example.

Note: It is essential for every trigger signaling or receiving task to keep the status of the trigger structure current with the number of the sample most recently processed. Samples are numbered starting with sample 0. The function **trigger_set_status** always must set the trigger status to a value which is greater than or equal to the previous trigger status.

See Also

`trigger_updt_status`

trigger_updt_put

Increment the trigger's status and assert the trigger at the new value.

```
void trigger_updt_put (
   TRIGGER * trig,                    // Trigger handle
   unsigned long int incr
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

incr

The number of samples.

Return Values

There is no return value.

Description

The function **trigger_updt_put** is a combination of a trigger status adjustment followed by an assertion at the new sample number. First, **trigger_updt_put** computes a new status, adding *incr* samples to the old status number field. Then, it signals a new event by placing this sample number into trigger *trig*.

The same effect can be achieved by fetching the value of the trigger status, adding *incr* to that number, and then calling **trigger_put** to assert the trigger event and update the status.

This function is particularly useful when data samples are processed in blocks. While scanning a data stream, if an event is detected at the Nth sample in the block, call the **trigger_updt_put** function:

```
trigger_updt_put(trig,N);
```

Otherwise, call the **trigger_updt_status** function:

```
trigger_updt_status(trig,N);
```

See Also

trigger_put, **trigger_updt_status**

trigger_updt_status

Increment a trigger's status field.

```
void trigger_updt_status (
   TRIGGER * trig,                    // Trigger handle
   unsigned long int incr
   );
```

Parameters

trig

Pointer variable containing a handle for the trigger to be accessed.

incr

The number of samples to increment the trigger *trig* status.

Return Values

There is no return value.

Description

The function **trigger_updt_status** increments the status number field of trigger *trig* by *incr* samples. This informs the DAPL system that any events or data corresponding to these samples are no longer needed.

This function is particularly useful when data samples are processed individually. Call **trigger_updt_status** to adjust the status count by one after each sample is processed.

Note: It is essential for every trigger signaling or receiving task to keep the status of the trigger structure current with the number of the sample most recently processed. Samples are numbered starting with sample 0. When processing blocks of data, beware of using this function in combination with the **trigger_put** function, which also adjusts the trigger status count.

See Also

trigger_put

trigger_wait

Extract and return the value of a trigger assertion when it becomes available. Automatically discard unneeded data.

```
unsigned long int trigger_wait (
   TRIGGER *trig,                    // Trigger handle
   PIPE *pipe,                       // Pipe handle
   unsigned long int pre_count,
   unsigned int mult
   );
```

Parameters

trig
Pointer variable containing a handle for the trigger to be accessed.

pipe
A pipe containing data samples to be processed.

pre_count
The number of pre-trigger samples

mult
A trigger rate correction. Most applications set *mult* equal to one. *mult* can be set to some other value N to locate a group of N samples in a multiplexed data set, in the manner that the WAIT command provided by the DAPL operating system processes multiplexed input channel list data.

Return Values

The function returns the value of an assertion from trigger *trig*.

Description

The function **trigger_wait** extracts and returns the value of an assertion from trigger *trig* . The function is used by trigger reader tasks that respond to trigger events by taking a data block from the *pipe* data stream.

While waiting for a trigger assertion to appear in trigger *trig*, function **trigger_wait** automatically removes unneeded data from pipe *pipe*, and updates the trigger status to account for the samples removed.

When function **trigger_wait** returns, pipe *pipe* contains data beginning *pre_count* samples before the trigger assertion. The calling task can use **pipe_value_get** or **pbuf_get** to fetch the data associated with the signaled event.

See Also

pipe_value_get, **pbuf_get**

vector_length

Determine the length of a DAPL vector.

```
int vector_length (
   VECTOR  *vect                          // Vector handle
   );
```

Parameters

vect

Pointer variable containing a handle for the vector to be examined.

Return Values

The function returns the number of elements in a DAPL vector.

Description

The function **vector_length** is useful for determining an index bound for accessing items in a DAPL vector.

See Also

vector_width, **vector_start**

vector_start

Return a pointer to the first element in a vector defined in a DAPL configuration.

```
void *vector_start (
   VECTOR  *vect                          // Vector handle
   );
```

Parameters

vect

Pointer variable containing a handle for the vector to be examined.

Return Values

The function returns a pointer to the first element in a DAPL vector.

Description

The routine **vector_start** returns a pointer to the first element in a vector defined in a DAPL configuration. The returned pointer must be cast to the appropriate data type before attempting to access the vector data.

See Also

vector_length

vector_type

Return the type of data contained by a DAPL vector.

```
unsigned long vector_type (
   VECTOR  *vect                          // Vector handle
   );
```

Parameters

vect

Pointer variable containing a handle for the vector to be examined.

Return Values

The function returns a code which indicates they type of data contained by the vector. This returned code is one of the codes used to specify a vector data type during parameter processing.

Description

The routine **vector_type** accepts a handle to a DAPL vector of any data type, and returns a code indicating the type of data contained by the vector.

See Also

param_process

vector_width

Return the size in bytes of one data element in a DAPL vector.

```
int vector_width (
   VECTOR  *vect                          // Vector handle
   );
```

Parameters

vect

Pointer variable containing a handle for the vector to be examined.

Return Values

The function returns the size in bytes of one data element in the vector.

Description

The function **vector_width** reports the number of bytes for one element of a DAPL vector storage array, in the manner that a C *sizeof* operator would report the size of a C variable or *struct*. This is useful for determining storage utilization directly, rather than deriving storage size from task parameter information.

See Also

vector_length, **vector_type**

13. Appendix A. Compatibility with DTD Version 4

This appendix discusses changes and compatibility between version 4 of the Developer's Toolkit for DAPL and the new version 5.

Hardware Compatibility

The Developer's Toolkit for DAPL version 5 generates 32-bit code that is not compatible with a 16-bit processor. Features that apply only to 16-bit products are not supported in the Developer's Toolkit for DAPL version 5. The Developer's Toolkit for DAPL version 4 has been remarkably stable, and it can continue to be used to develop code for older 16-bit Data Acquisition Processor models if this is necessary.

Binary Code Compatibility

There is no binary compatibility between 32-bit modules and 16-bit custom commands. The binary modules are completely different. The BDOWNLOAD command and the DAPIO features used to download 16-bit custom commands will not work with 32-bit modules. You must use the 32-bit downloading facilities as described in the chapter *Compiling and Downloading*.

Compatibility with Previous DTD Versions

Most of the source code developed for 16-bit custom commands can be converted easily to the new form required for 32-bit custom modules. The majority of the DAPL system functions work the same as they did in the Developer's Toolkit for DAPL version 4, and the overall organization of the source code is very similar. However, there are there some fundamental reasons why 100% source code compatibility is not possible.

- The compilers are different
- The compiler output formats are different
- The instruction sets are different
- The data element sizes are different
- The runtime environment is different.

The CDAPBACK.H file from version 4 mapped many deprecated or retired functions from earlier Developer's Toolkit for DAPL versions into version 4 equivalents. Users with a strong commitment to preserving old source code can salvage the CDAPBACK.H from the Developer's Toolkit for DAPL version 4 and make

appropriate modifications. This should make it possible to continue using many of the archaic code conventions. Microstar Laboratories does not endorse this practice, so exercise good judgment.

Among the version 3 or earlier features no longer supported:

1. Historical function names for many of the fundamental services, including pipe, pipe buffer, system information, triggering, and PID operations.
2. Compatibility macros that mapped old naming schemes.
3. Features related to the 16-bit DSP operations on the historical DAP 2400e and DAP 2400a products. These services were replaced in all newer products.

Use of *int* data type

The 16-bit compilers implemented the int data type as 16-bit fixed point. With the 32-bit compilers, the int data is implemented as 32-bit fixed point. Compilers are free to do this, as the precision of the int data type is bounded below but not above. Be careful, this is not consistent with the behavior of all compilers, and may be unfamiliar.

The extra precision will usually make no difference to program flow, but it can make a big difference in the management of sampled data, which arrives in a compacted 16-bit format. Pointers and data structures must be consistent with this, or the compiler will generate incorrect addresses when fetching the data. As a general rule, any constants, structures or variables which relate to data samples and their values should be 16-bit short int or short unsigned to match the precision of the data.

The fft and fir filtering functions make extensive use of data buffering. The function prototypes have been updated to declare the buffer storage areas short int rather than int, but the calling code will need to be updated to make storage declarations compatible.

32-bit Variable Access

The functions `var32_get` and `var32_set` are no longer provided. Under the 16-bit Developer's Toolkit for DAPL version 4, there was a very small chance that a multitasking operation could interrupt a task between the time that the upper and lower 16 bits of a 32-bit value were stored or fetched. This could cause confusion with other tasks sharing that variable. However, in the 32-bit environment, these operations are done in a single 32-bit machine cycle, with no possibility of interruption, so no useful purpose is achieved.

If there is extensive use of DAPL variables, and it is undesirable for some reason to upgrade the code to remove the unsupported function, user-defined macros can be added to the DTD.H file:

```
#define  var32_set(pt,val)    (*pt=val)
#define  var32_get(pt)    (*pt)
```

Multitasking Control

The `sys_set_multitasking( )` function is no longer useful and is removed. Many versions ago, the DAPL operating system had a certain level of overhead associated with operation of the multitasking system. Since that time, the multitasking system has been improved so that there is no performance difference between using this feature and not using it -- but the restrictions necessary to use it are very complicated. Hence, the decision was made to remove it.

If there is existing code with a heavy commitment to using this function, a user-defined macro can be added to the DTD.H file:

```
#define   sys_set_multitasking(mode)    (1)
```

PID Gain

Back in antiquity, the DAPL `PID` command (which is no longer supported) did an unconventional thing. In a conventional control loop structure, a PID correction is computed by first taking the setpoint input minus the feedback. So, for example, if the feedback level is too low, the resulting difference is positive. This difference is multiplied by P, I and D gain terms to compute the control output. The problem was, the old `PID` command got the sign of the difference wrong. Consequently each P, I and D gain term needed to have its sign reversed to make the sign of the control output right.

This problem was corrected long ago in the DAPL system, with the introduction of the processing command `PID1`. In the relatively quiet Developer's Toolkit for DAPL, there was no opportunity to repair the old sign problem, and the PID function set continued with the reversed sign.

The sign problem is now corrected in Developer's Toolkit for DAPL version 5. Rather than risk that old code would be compiled and seem to run just fine, only to discover that a sign was changed, a decision was made to remove the function `pid_update`. The function **`pid_compute`** is its replacement. It has the same form and purpose as `pid_update`, except for the difference in sign.

To convert to the new function, it is necessary to do one of the following:

1. Locate the code that sets up the P, I and D gains for the custom PID control, and reverse the signs on each term.
2. In the controller custom command, apply a minus sign to each value computed by function `pid_compute`, and then continue to use the inverted gain settings as before.

Pipe PBUF Get and Put

The operations of setting an output sample count and then putting that number of data to a pipe almost always occur together. The complementary operations, getting a number of data from a pipe and then determining how many values are available, also occur in pairs. It is now possible to perform paired operations with one function call. Check the function description details and the tutorial sections for more information about how to use this feature to improve program efficiency.

Dynamic Allocations

Code that manages large dynamic areas as lists of independently-allocated “pages” will still work. If there are any problems, the **`ralloc`**`()` function can now allocate very large memory regions, so it might be easier to restructure the data as a single contiguous storage region. This function also has a companion **`rfree`**`()` function that can be used to release dynamic allocations.

The **`malloc`** and **`free`** functions that are employed by C++ constructors and destructors are also supported.

A New `sys_get_version` Function

The **`sys_get_version`** function has changed to work with new DAP systems and server software. Applications that use the old version of this function must modify both the function call and the processing of the returned values. In return for this inconvenience, the returned information should be much easier to interpret because there are no intermediate number codes to look up.

C++ Environment

The Developer's Toolkit for DAPL version 5 supports C++ compiler mode. To the extent that C++ is a “better C” (that argument will never be settled) most code will

work the same as it did before, and a “C” coding style as opposed to an object-oriented style is still perfectly acceptable.

The C++ exceptions facility cannot be used. Perhaps this limitation can be removed in future Developer's Toolkit for DAPL versions, but we must live with this for now. The most significant consequence of this is that constructors do not have any mechanism to indicate an internal failure. So, for example, an object could be returned that does not have a proper initialization of one of the derived or component classes. The choices are:

- validate the object after construction
- presume that construction is okay if the returned pointer value is not a NULL.

Index