Developer's Toolkit for DAPL Manual

Custom command developer's toolkit
for DAPL and DAPL 2000
operating systems

Version 4.02

Microstar Laboratories, Inc.

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

Part Number MSDTDM402-9807

Contents

1. Introduction

The Developer's Toolkit for DAPL contains the software tools required for creating custom commands for Microstar Laboratories Data Acquisition Processors. Custom commands are user-defined processing task commands that extend the DAPL or 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 designed primarily for advanced users.

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

A DAPL file downloaded to the Data Acquisition Processor defines processing procedures containing lists of task. Each task definition invokes a predefined or custom command. The task definition specifies a list of parameters, identifying the data sources and data destinations to be used by the custom command. A typical custom command consists of three sections: a section that analyzes the task parameters, an initialization section, and an endless processing loop.

When the Data Acquisition Processor receives a `START` command for a processing procedure containing a custom task, the Data Acquisition Processor activates the custom command. The custom 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 the 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 custom command parameter list may be communication pipes, input channel pipes, user-defined pipes, or other types. It makes no difference within the custom command; all pipes are treated uniformly.

This document explains how to create and use custom commands. Before studying this document, the reader should become familiar with the Data Acquisition Processor manuals, particularly the DAPL manual.

Compatibility

The Developer's Toolkit for DAPL supports DAPL versions 4.0 and above and DAPL 2000 versions 1.0 and above. Some DAPL features supported by the Developer's Toolkit for DAPL are not available in DAPL versions prior to version 4.3. The waveform, FFT, and FIR filter functions are available only with DAPL 2000.

For all supported compiler versions, the custom commands are compiled as DOS applications written in C.

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

- Microsoft C compiler version 6
- Microsoft C/C++ compiler version 7
- Microsoft Visual C++ version 1.0 and 1.5 Professional Edition
- Borland C++ compiler versions 4.0 and 4.5

The Developer's Toolkit for DAPL supplies two versions of its runtime library. The `SMALL` library supports all Data Acquisition Processor, DAPL, and supported compiler types. This library provides no floating point features, but generates the smallest and most efficient code modules. The `FP` library uses more code and system memory space, and provides access to the floating point emulator or hardware services integrated into the DAPL operating system.

Installation

Before installing the Developer's Toolkit for DAPL software, make backup copies of all Microstar Laboratories diskettes and store the original diskettes in a safe place.

After making backup copies, insert the Developer's Toolkit for DAPL diskette in drive `A:` and enter the DOS command:

```
MORE  <A:README.TXT
```

The PC will display any recent changes or corrections that are not included in this document.

The Developer's Toolkit for DAPL is installed by copying the contents of the diskette to a directory on your PC. The DOS `XCOPY` command can be used to perform this copy:

```
XCOPY A:*.* C:\DTDC /S
```

The Developer's Toolkit for DAPL diskette includes a file named `FILES.TXT`, which contains descriptions of all of the files on the diskette.

2. Data Acquisition Programming in C

The Developer's Toolkit for DAPL provides a C runtime environment supporting data acquisition and real time applications. Within this environment, the Developer's Toolkit for DAPL provides a library of functions for input and output of data and for control of the Data Acquisition Processor hardware. These functions are called system routines because they give access to services provided by the DAPL or DAPL 2000 operating system. In addition to the system routines, the Developer's Toolkit for DAPL library supplies replacement functions for some of the Microsoft or Borland C runtime library components. These substitutions are automatic and invisible, and they require no changes to the runtime libraries that come with the compilers.

A C custom command is a conventional C program that calls system routines to perform operations specific to the DAPL environment. The most commonly used routines provide access to DAPL pipes and the data they contain:

pipe_get	read a data value from a DAPL pipe
pipe_get_float	read a floating point value from a DAPL `FLOAT` pipe
pipe_put	write a data value to a DAPL pipe
pipe_put_float	write a floating point value to a DAPL `FLOAT` pipe
pipe_open	open a DAPL pipe for input or output
pipe_num	determine whether a pipe contains data
pipe_num_complete	return the number of data values in a pipe
pipe_width	return the width of a DAPL pipe
pipe_purge	remove all data values from a DAPL pipe
pipe_rem	remove data items from a DAPL pipe

Eleven functions provide access to pipe data in blocks called pipe buffers, rather than individual items:

`pbuf_open`	open a pipe buffer and allocate storage
`pbuf_get`	read a block of data from a DAPL pipe
`pbuf_put`	write a block of data to a DAPL pipe
`pbuf_get_data_ptr`	obtain a pointer to the data storage
`pbuf_set_data_ptr`	set the data storage pointer
`pbuf_get_cnt`	determine the number of items available in storage
`pbuf_set_cnt`	set the number of items placed into storage
`pbuf_get_min_cnt`	determine the minimum number of items to fetch
`pbuf_set_min_cnt`	set the minimum number of items to be fetched
`pbuf_get_max_cnt`	determine the maximum number of items to fetch
`pbuf_set_max_cnt`	set the maximum number of items to fetch

Four system routines control the Data Acquisition Processor output hardware:

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

Twelve system routines manipulate DAPL software 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

Two system routines provide timing functions:

`sys_get_time`	get the value of the real-time clock
`task_pause`	pause for a specified time

Seven system routines provide string formatting and text output functions:

send	print a string
fsend	print a string to a specific output pipe
printf	format and print a string
sprintf	format a string
fprintf	format a string and print the string to a specific output pipe
sscanf	parse a string using C-style conversions
sys_exec_command	send a direct command to the DAPL operating system

Six routines provide integer math operations and waveform construction:

icosine	return the integer cosine of an integer value
isine	return the integer sine of an integer value
isqrt	return the integer square root of an integer value
isinewave	construct a sampled sine function table
icoswave	construct a sampled cosine function table
icplxwave	construct a complex-valued function table

Five routines provide PID control services:

pid_open	open a PID control block
pid_preset	establish a pre-determined PID operating state
pid_set_setpoint	adjust the PID setpoint
pid_tune	set the values of PID control parameters
pid_update	compute the real-time PID control output

Six routines provide access to fast Fourier transform operations and supplemental processing:

fft_init	initialize an FFT control block
fft_request	request an FFT transform operation
fft_status	check completion of transform
fft_receive	obtain transform results
fft_postop	perform follow-up processing after transform
fft_chngbuf	change the FFT storage area

Six routines provide access to FIR digital filtering operations:

fir_init	initialize a FIR filter control block
fir_request	request filtering of a supplied data block
fir_status	check completion of filtering
fir_change	dynamically alter filter characteristics
fir_receive	obtain filtering results
fir_advance	skip filtering steps by adjusting shift register

Two routines provides multitasking control:

sys_set_multitasking	turn multitasking on or off
task_switch	force a task to release the CPU

The following routines perform miscellaneous functions:

atof	convert an ASCII string to a float
exit	terminate a task
memcpy	copy a memory region
param_error	issue an error message
param_error_msg	generate task error message and terminate task
param_process	extract parameters and perform parameter type checking
param_type	return the data type of a task parameter
ralloc	allocate temporary storage
vector_length	return the length of a DAPL vector
vector_start	return the address of the first element of a DAPL vector
vector_type	return the type of data contained by the DAPL vector
vector_width	return the size in bytes of one data element in the DAPL vector
sys_get_info	return system information
sys_get_version	return the DAPL version number

Five routines provide access to the digital signal processor on a DAP 2400a:

dsp_alloc	allocate a DSP request
dsp_request_init	initialize a DSP request
dsp_send_request	send a DSP request
dsp_done	return the status of an active DSP request
dsp_receive_result	receive results of a DSP request

Sample Custom Command

A typical custom command implements a task which reads data from one or more input pipes, processes the input data in some way, and writes data to one or more output pipes. The command may also have various constant or variable parameters which control the way the command processes data. The pipes and the task which uses the custom command are defined by a DAPL file downloaded to a Data Acquisition Processor. The DAPL file associates the pipes specified in the task parameter list with the corresponding parameters of the custom command. The pipes serve as a logical connection between tasks when the custom command is started.

The following is a typical example. It can serve as a starting place for developing new custom commands. This example, called `ZTRUNC`, reads data values from one pipe, limits values to a specified lower limit, and sends the modified data to another pipe:

```
/* ZTRUNC (p1, vlim, p2)
* - read data from pipe 'p1'
* - truncate numbers below the limit 'vlim'
* - output data to pipe 'p2'
*/
#include <cdapcc.h>

/* Function prototypes */
void main (PIB **);
void ztrunc (PIPE *, VAR *, PIPE *);

void main (PIB **plib)
{
    void **argv;
    int argc;
    /* Extract task parameters */
    argv = param_process (plib, &argc, 3, 3,
                          T_PIPE_W, T_VAR_W, T_PIPE_W);
    ztrunc ((PIPE *) argv[1], (VAR *) argv[2],
            (PIPE *) argv[3]);
}
```

```
void ztrunc (PIPE *in_pipe, VAR *low_limit, PIPE *out_pipe)
{
    int val;
    int limit;

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

    /* Perform the real-time processing */
    while (1) {
        val = (int) pipe_get (in_pipe);
        limit = *low_limit;
        if (val < limit) val = limit;
        pipe_put (out_pipe, (long)val);
    }
}
```

The following description briefly explains how this custom command works. All the Microstar Laboratories system calls are explained in detail in later chapters.

`CDAPCC.H` is the include file for the Developer's Toolkit for DAPL. `CDAPCC.H` defines all constants, macros, and types required for access to system routines. This file must be included in any custom command source file. In the above example, the identifiers `PIB`, `T_PIPE_W`, `T_VAR_W`, `P_READ`, `P_WRITE`, **`param_process`**, **`pipe_open`**, **`pipe_get`**, and **`pipe_put`** are defined in the file `CDAPCC.H`.

A custom task begins execution at `main`. DAPL passes `main` a pointer to the task's parameters. The function **`param_process`** checks that the task parameters are of the correct types and returns an array of parameter pointers. The two `T_PIPE_W` identifiers specify that the corresponding command parameters are pipes containing data of type `int`. The `T_VAR_W` identifier specifies that the remaining parameter is a `VARIABLE` of type `int`. The three parameter values `argv[1]`, `argv[2]` and `argv[3]` are extracted in sequence, and passed to the function called `ztrunc`.

The first and third parameters of the `ZTRUNC` command point to pipes. The file `CDAPCC.H` defines a `PIPE` data structure. The pipe structures are not manipulated directly, but rather are passed to and from system functions by means of pointers. The `main` function casts the two pipe arguments of `ZTRUNC` into the type `PIPE *` as it passes them to the auxiliary function `ztrunc`.

The `ZTRUNC` command also requires a lower limit parameter which is a DAPL `VARIABLE`. This variable is defined by a `VARIABLE` command in DAPL and declared

as type VAR in a custom command. The main function casts the second argument of ZTRUNC into the type VAR * as it passes it to the auxiliary function ztrunc.

The function ztrunc first performs some initialization steps. It opens in_pipe for reading and out_pipe for writing. No special setup is required for the variable limit parameter.

The while loop performs the actual data processing. The function **pipe_get** removes the next data value from the input pipe and places this value into val. Because the function **param_process** guarantees that the input pipe contains data of type int, we may cast the returned value to int. Next, function ztrunc fetches a copy of the most current value of the limit variable. The if statement replaces each value lower than the limit value with the limit value. The function **pipe_put** sends the data to the output pipe.

In order to use the ZTRUNC custom command, the custom command code must be compiled, linked, and downloaded from the PC to the Data Acquisition Processor. This process is explained in more detail later in this document. After downloading, the ZTRUNC command can be used in any processing procedure, as in the following example:

```
; This DAPL command list acquires data from
; single-ended input channel 0, replaces all
; negative data values with zero, and prints
; the truncated data.

RESET
PIPES P1
VARIABLE VLIM = 0
IDEFINE A 1
SET IPIPE0 S0
TIME 10000
END
PDEFINE B
ZTRUNC (IPIPE0, VLIM, P1)
FORMAT (P1)
END
```

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

The `ZTRUNC` custom command code can be modified easily to perform many data processing functions. Simply replace the if statement with different C code which modifies the value of `val`. No additional knowledge about system routines is necessary.

3. System Interface File

The file CDAPCC.H defines the interface between C custom commands and the resources of the Data Acquisition Processor. CDAPCC.H contains structure, function, macro, and constant definitions. The upper-case and lower-case characters in identifiers printed in this document match the identifiers as they appear in CDAPCC.H.

The file CDAPCC.H may contain functions or structures that are not documented in this document — these are reserved for future expansion.

The file CDAPCC.H itself contains only definitions common to all Developer's Toolkit for DAPL library versions. At compile time, CDAPCC.H includes additional declarations from the file CDAP4.H, for supporting DAPL version 4, or the file CDAP16.H, for supporting DAPL 2000. One additional file called CDAPBACK.H is also included. This file has macro and function prototype definitions for backward compatibility with notations used in previous versions of the Developer's Toolkit for DAPL. Notations defined in CDAPBACK.H should not be used for new development. If the notations described in this manual are used, and not the old notations, the CDAPBACK.H include directive can be changed to a comment.

Structures and Types

DAPL uses several data types that are not defined in Standard C. `CDAPCC.H` defines these data types.

The most important data types are the ones which provide access to pipes, vectors, and triggers. These types are:

```
PIPE *
VECTOR *
TRIGGER *
```

`CDAPCC.H` also defines several auxiliary structured types — pipe buffer, parameter information block, PID control block, FFT control block, and FIR filter control block:

```
PBUF *
PIB *
PID *
FFTB *
FIRB *
```

The individual fields of these structures are not accessed directly. Instead, structure pointers are passed to and from system routines, and functions provide access to values stored internally. This ensures compatibility with future versions of DAPL and the Developer's Toolkit for DAPL.

`CDAPCC.H` also defines data types of general utility for use by C custom commands. The first two are integer pointer types that provide access to DAPL word variables and long variables:

```
VAR *
LVAR *
```

The next two provide access to DAPL constant values:

```
CONSTANT *
LCONSTANT *
```

The last one is a structure that organizes the coefficient sets used by PID control functions. It is defined and used only within custom commands, and it is the only structure for which individual fields can be accessed.

```
PIDCOEF
```

Functions and Macros

The file `CDAPCC.H` defines prototypes for all system functions.

Developer's Toolkit for DAPL system routines are implemented either as functions or as macros. Functions call system routines of the Data Acquisition Processor. Macros are identifiers defined with the C preprocessor directive `#define` to represent values or expressions.

Some of the system functions defined in the file `CDAPCC.H` have names prefixed with an underscore. These names are part of the interface implementation, and should not be called directly.

Constants and Enumerations

Several constants are defined in the file `CDAPCC.H`. Use of these constants is described in Chapter 4.

DAPL Parameter Types:

```
T_VAR_W
T_VAR_L
T_CONST_W
T_CONST_L
T_TRIGGER
T_PIPE_B
T_PIPE_W
T_PIPE_L
T_PIPE_FL
T_RFLAG
T_STR
T_VECTOR_W
T_VECTOR_L
```

Pipe Input/Output Flags:

```
P_READ
P_WRITE
```

Region Flag Values:

```
R_INSIDE
R_OUTSIDE
```

Hardware Type Values:

```
H_800     5
H_1200E   6
H_1200A   6
H_2400E   7
H_2400A   7
H_801     8
H_3200E   9
H_3200A   9
H_1216E  10
H_1216A  10
H_2416E  11
H_2416A  11
H_3000A  12
H_3400A  13
H_3216A  14
```

Request codes for **sys_get_info**:

```
GI_DECIMAL
GI_TERMINAL
GI_OVERQ
GI_OBIPOLAR
GI_IBIPOLAR
GI_SYSOUT
GI_SYSIN
GI_IN_CNT
GI_OUT_CNT
GI_SERIAL
GI_ICHAN_CNT
GI_IN_ACTIVE
GI_DEFAULT_BUF_SIZE
GI_OUT_ACTIVE
```

```
GI_HMEMAVL
GI_HMEMSIZE
GI_TMEMAVL
GI_TMEMSIZE
GI_OEM_ID
GI_FLOAT_ERROR
GI_ROUNDING
GI_AINEXPAND
GI_FFTSIZE
GI_IBURST_ACTIVE
GI_OBURST_ACTIVE
GI_BUFFERING
GI_SCHEDULE_MODE
GI_QUANTUM
```

Selections for multitasking control:

```
eMultiOn
eMultiOff
eMultiOffSYSIN
```

Selections for scheduling control:

```
eSchedAdaptive
eSchedFixed
```

Selections for buffer size control:

```
eBuffersNone
eBuffersMedium
eBuffersLarge
```

Selections for DSP process configuration:

```
eWaveWord
eWaveLong
FFTDIR_FORWARD
FFTDIR_REVERSE
WINDOW_RECTANGULAR
WINDOW_HANNING
WINDOW_HAMMING
WINDOW_BARTLETT
WINDOW_BLACKMAN
FFTPOST_DEFER
FFTPOST_REAL
FFTPOST_CPLX
FFTPOST_POWER
FFTPOST_MAGNITUDE
FFTPOST_MAG_PHASE
FFTPOST_NORMPOWER
FFT_REALIN
FFT_CPLXIN
FFT_PAIRWISE
FFT_SEPARATED
FFT_HALFOUT
FFT_FULLOUT
```

The last part of file `CDAPPCC.H` provides C prototypes for the system functions in the Developer's Toolkit for DAPL.

4. Using the Data Acquisition Runtime Library

This chapter illustrates the Developer's Toolkit for DAPL system routines with a number of typical custom commands. Several useful programming techniques to facilitate development of custom commands are described.

Custom Task Parameters

When the Data Acquisition Processor begins executing a custom command task, the task's`main` function is called. This function must obtain access to the parameter information provided by DAPL.

The `main` function of a custom command receives a pointer to the task's parameters in a structure called a PIB (rather than the usual `argc` and `argv` command line parameters of a Standard C application). The Data Acquisition Processor can execute several instances of a custom command concurrently; each task receives a pointer to a unique PIB. Each pointer in the list points to an internal DAPL data structure containing complete information about the corresponding parameter.

The custom command needs to extract the required parameter values from these data structures. While doing this, it also needs to check the parameter list to see whether the required parameters are present and of the correct types. The function **`param_process`** provides these services; it checks parameter types and returns a list of pointers to parameter values.

The use of **`param_process`** is illustrated in the following example:

```
#include <cdapcc.h>
void main (PIB **plib)
{
void **argv;
int argc;
argv = param_process (plib, &argc, 2, 2,
                      T_PIPE_W, T_VAR_W);
.
.
.
```

The first parameter to **`param_process`** is the `plib` parameter which the DAPL operating system provides to the `main` function. The **`param_process`** function sets the second parameter variable to the number of parameters in the parameter list for the task in the DAPL file. The next two parameters respectively specify the minimum and maximum numbers of parameters that the custom task accepts. This allows a task to accept a variable number of parameters. The remaining parameters are flags that specify the correct parameter types for each of the task's parameters.

The function **param_process** returns a pointer list in the manner of the argv parameter of a Standard C main function, except that here, the pointers in the list are pointers to various data types, not just text strings. For example, pointers to numeric values or to data elements such as PIPE and VARIABLE can be in this list.

The example above specifies that the task's first parameter is a word pipe and the task's second parameter is a word variable. The function list pointer returned by **param_process** is assigned to the local argv variable. The task's parameters then can be referenced by indexing argv:

```
argv[0] - pointer to the name of the custom command
argv[1] - pointer to parameter 1
argv[2] - pointer to parameter 2
argv[3] - pointer to parameter 3
etc.
```

There are a number of techniques that can be used to extract the pointers from this list. One way is to declare pointer variables of the appropriate type and assign the values from the list, applying the appropriate type casts. Another way is to cast the returned list pointer to the Standard C Library type va_list and use the C va_arg macro to extract the 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 them as pointers of the appropriate type. Most of the examples in this manual use the auxiliary function technique.

Parameter Types

A custom command can use many types of DAPL parameters. The following table lists the allowed DAPL parameter types, the C data types which correspond to the DAPL parameters, and the type flags used by **param_process**.

DAPL Type	C 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_FL
trigger	TRIGGER *	T_TRIGGER
word vector	VECTOR *	T_VECTOR_W
long vector	VECTOR *	T_VECTOR_L
word variable	VAR *	T_VAR_W
long variable	LVAR *	T_VAR_L
word constant	CONSTANT *	T_CONST_W
long constant	LCONSTANT *	T_CONST_L
region flag	const int *	T_RFLAG
string	const char *	T_STR

PIPE, TRIGGER, and VECTOR are C types defined in the file CDAPCC.H. These structures are not manipulated directly, but pointers to them are passed to and from system routines.

The first symbol in a DAPL region is a DAPL region flag INSIDE or OUTSIDE. A region flag is a pointer to a constant that contains one of two values, R_INSIDE or R_OUTSIDE. These constants are defined in the file CDAPCC.H. A region flag is always followed by two word values, either variables or constants, which define the lower and upper limits of the region.

A DAPL string is a pointer to a character array. DAPL strings are defined using the DAPL STRING command. The contents of DAPL strings must not be modified by custom commands.

Parameter Type Checking

In addition to generating an argument vector, **param_process** checks the type of each task parameter. A custom task must verify that it receives a correct number of parameters of the correct types. If incorrect parameters are passed to a task, and if the parameter types are not checked, a system failure may result.

If **param_process** detects a parameter error in a task, it generates an error message and halts the task. When the ZTRUNC command, described in Chapter 2, performs parameter type checking, there may be an error in the parameter list specified in the DAPL command file. The ZTRUNC command expects two PIPE parameters and one VARIABLE parameter:

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

The parameter errors in this DAPL file cause **param_process** to generate the following error messages:

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

Note that each error message identifies the command detecting the error, and provides additional information for diagnosing the problem.

Advanced Parameter Checking

Some custom commands permit optional parameters or several different combinations of parameter types. An example of this is the `COPY` command built into DAPL. This command allows either word pipe or long pipe parameters. In addition, from two to thirty-three parameters can be specified.

Complex parameter lists can be checked with **`param_process`**. The function **`param_process`** allows the specification of a minimum and a maximum number of allowed parameters, as well as information about possible parameter types. When more than one parameter type is valid, use the C bitwise "or" operation to combine all the valid types. For example, a custom command may have the following parameter requirements:

- accept from two to four parameters
- the first parameter must be a vector
- the next parameter can be any length integer variable
- the next parameter can be a trigger or a word constant
- the last parameter must be a word pipe

The custom command's parameters can be checked as follows:

```
#include <cdapcc.h>
void main (PIB **plib)
{
void **argv;
int argc;
argv = param_process (plib, &argc, 2, 4,
T_VECTOR,
T_VAR_W | T_VAR_L,
T_TRIGGER | T_CONST_W,
T_PIPE_W);
```

After parameter processing, a command can perform any number of additional checks. In the above example, it might be necessary for the second parameter to be type `T_VAR_W` when the third parameter is type `T_TRIGGER`. The **`param_type`** function is useful for analyzing such cases. If the supplementary tests detect an error condition, the function **`param_error`** can be used to generate a generic error message, or the function **`param_error_msg`** can be used to generate a more specific message. When a custom command task calls **`param_error`**, the task terminates with the message:

```
*** ERROR 1217: <name> - parameter error
```

When a custom command task calls **param_error_msg**, the task terminates with the message:

```
*** ERROR 1236: <name> - parameter <nn> - <description>
```

For example, suppose that custom command XCOM executes the following:

```
if ( (param_type(plib,2) != T_VAR_W) &&
     (param_type(plib,3) == T_TRIGGER) )
{
    /* terminate task with message */
    param_error_msg(pe_TypeInconsistent,3);
}
```

The corresponding error message is:

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

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.

The following example illustrates the use of **param_type** in a custom command that accepts both word and long DAPL constants. A DAPL constant can be either a word constant or a long constant. Sometimes it is convenient for a custom command parameter to accept either constant type. A number represented by a 16-bit word is passed to the custom command as a word constant. A number represented by a 32-bit word is passed as a long constant.

The following code fragment from the example custom command CPRINT.C uses **param_type** to determine how to assign the value of a constant input parameter to a long integer variable:

```
#include <cdapcc.h>
void main (PIB **plib)
{
void **argv;
int argc;
long val;
argv = param_process (plib, &argc, 1, 1,
                      T_CONST_W | T_CONST_L);

if (param_type(plib,1) == T_CONST_W)
    val = (long int) *(int *) argv[1];
else
    val = *(long int *) argv[1];
  .
  .
  .
```

This example accepts a single parameter that can be a word constant or a long constant. The **param_type** function detects the parameter type. If the parameter is a pointer to a word constant, the parameter pointer is cast to a word constant pointer, de-referenced, and the word value is cast to a long integer. The long integer then is assigned to the variable `val`. If the parameter is a pointer to a long constant, the parameter pointer is cast to a pointer to a long integer. Then, the pointer is de-referenced and the long word value is assigned to `val`.

Variables and Constants

DAPL provides access to the special `VARIABLE` and `CONSTANT` values defined in a DAPL command file. A constant is defined in DAPL by an explicit number value in a parameter list or by a reference to a name defined by the `CONSTANT` command. Variables are defined in DAPL by the `VARIABLE` command.

Because a constant value may be used by several tasks, a DAPL constant must never be modified by a custom command. One way to enforce this requirement is to use the special integer type `CONSTANT` defined in the file `CDAPCC.H` rather than an ordinary `int` type. The `CONSTANT` type adds a `const` qualifier to the ordinary `int` type, warning the compiler not to allow this value to be modified. An alternative is to copy the value of the constant to an `auto` or `static` variable in the custom command.

DAPL allows more than one task to access its `VARIABLE` values. These special variables are declared in a custom command with the type `VAR` or `LVAR`. These types are defined in the file `CDAPCC.H`. They have a `volatile` type qualifier. This disables certain compiler optimizations that are invalid for variables shared by multiple tasks.

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 access the variable value through a pointer. 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 */
```

In any multitasking system, multiple tasks can sometimes attempt to simultaneously access a shared variable. This leads to problems if one task attempts to read the shared variable while another task has partially written a new value to that variable. The functions **`var32_get`** and **`var32_set`** avoid task preemption during the fetch and write operations respectively, avoiding this problem.

Vectors

Vectors defined by the DAPL command `VECTOR` can contain data of type `short int` or `long int`. Because a vector defined by DAPL is a special data type, DAPL provides special means for determining the properties of the data and addressing the data array. Two mechanisms are used: parameter type checking and special functions.

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 code `T_VECTOR_W` or `T_VECTOR_L`. After that, the parameter can be extracted and assigned to a `VECTOR` variable.

Note: 32-bit long vectors and `T_VECTOR_L` are available only with DAPL 2000.

The following code tests for a task parameter list with a single long integer vector, and extracts the vector parameter to a local variable:

```
VECTOR * vect;
argv = param_process(plib, &argc, 1, 1, T_VECTOR_L);
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

Note: **`vector_type`** and **`vector_start`** are available only with DAPL 2000.

The length and storage location information are available only using the special functions. The other information also 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 file are available to multiple tasks, and should not be altered. One way to protect against accidentally changing vector data is to declare the contents to be type `CONSTANT` or `LCONSTANT` rather than `int` or `long`. Again 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);
```

Auxiliary Functions

After calling **param_process**, a custom task receives a list of pointers to its parameters. According to the declaration for the list pointer (called `argv` in the examples), the parameters are pointers to type `void`. This means that the pointers must be cast into pointers of the appropriate types before the data can be accessed.

One useful technique for structuring a custom command and organizing parameters is calling an auxiliary function. The formal parameter list of the auxiliary function is used to assign a mnemonic name and a correct type to each of the custom task's parameters. Each of the pointers from the parameter list is extracted and cast into an argument to the auxiliary function. This technique is particularly useful for simpler custom commands having a minimum of parameter checking requirements.

The following custom command calls the auxiliary function `pval` after checking parameters.

```
#include <cdapcc.h>
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 ... */
}
```

Most of the examples shown in this document call an auxiliary function whose name is the same as the name of the custom command.

Initializations and Allocations

Initializations must be performed after a custom command has extracted and checked its parameters. Many 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 to a DAPL structure of some kind. The pointer value must be stored so that it can be used during later operations.

A custom command must open each pipe 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 file CDAPCC.H defines two flags for this purpose: P_READ for input and P_WRITE for 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 must perform two initializations for each pipe. First, the **pipe_open** operation described above must be performed. Then, a structure called a pipe buffer must be allocated and initialized. This second step is performed using the function **pbuf_open**, which returns a pointer to 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 default values, which will be satisfactory in most cases. More information about using the PBUF structure is provided in the section on blocked pipe operations.

The following shows a typical initialization for reading blocks of up to 200 items.

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

Tasks which issue or receive software trigger assertions must initialize the `TRIGGER` structure 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 5 for 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.)

Some of the more specialized initializations, such as PID control or DSP computations, are discussed in detail in later chapters.

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.

Some custom commands operate on a small amount 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 first case. However, most of fundamental principals discussed in this section apply in both cases. Be sure to have a good understanding of this section before covering the section on blocked pipe operations later in this chapter.

Two important system routines are **`pipe_get`** and **`pipe_put`**. The function **`pipe_get`** removes one data value from a DAPL pipe. If a pipe is empty when **`pipe_get`** is called, the calling task goes to sleep until the pipe contains data. The function **`pipe_get_float`** is the same as **`pipe_get`**, except that it obtains a value from a floating point pipe.

The function **`pipe_put`** places a data value into a pipe. The function **`pipe_put_float`** is equivalent to **`pipe_put`** except that it puts a value into a floating point pipe. If the pipe is full, the calling task either suspends operation until the associated pipe has room, or throws out the data and returns immediately, as selected by the `WAIT`/`NOWAIT` parameter in the DAPL `PIPE` command. The default behavior for a DAPL pipe is `WAIT`.

Previous examples illustrated the use of **`param_process`** and an auxiliary function named `pval`. The functions **`pipe_open`** and **`pipe_get`** can be used to create a custom command with the same behavior as the DAPL predefined `PVALUE` command:

```
/* PVAL (p, v)
*   - keeps variable 'v' updated to the most recent
*     value in pipe 'p'
*/
#include <cdapcc.h>
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)
{
pipe_open (p, P_READ);
while (1) {
    *v = (int)pipe_get(p);
}
}
```

Most custom commands can be implemented using only the four system routines: **param_process**, **pipe_open**, **pipe_get**, and **pipe_put**. However, more efficient processing is often possible using the blocked pipe operations described later in this chapter.

Application Examples Using Pipes

The following custom command implements a simplified version of the predefined `COPY` command. It reads integer data from an input pipe and puts copies of the data into two output pipes.

```
/* COPY2 (p1, p2, p3)
*   - places copies of data from pipe 'p1' into
*     pipes 'p2' and 'p3'
*/
#include <cdapcc.h>
void copy2 (PIPE *p1, PIPE *p2, PIPE *p3);

void main (PIB **plib)
{
void **argv;
int argc;
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]);
}
void copy2 (PIPE *p1, PIPE *p2, PIPE *p3)
{
long int d;
pipe_open (p1, P_READ);
pipe_open (p2, P_WRITE);
pipe_open (p3, P_WRITE);
while (1) {
    d = pipe_get (p1);
    pipe_put (p2, d);
    pipe_put (p3, d);
}
}
```

This is a complete custom command application. The compile and download procedure given in Chapter 2 can be used to load the compiled command into the Data Acquisition Processor for execution. Once the COPY2 custom command is downloaded, a DISPLAY command can be used from DAPview to verify that COPY2 is defined:

```
#display commands
  COPY2 stacksize=1000
#
```

Note: A RESET command does not erase custom command definitions. RESET can be used between DAPL applications without requiring redefinition of custom commands. The RESTART command removes custom command definitions. The ERASE command erases a selected custom command definition.

A useful way of testing custom command code is to define a custom task and present the task with test data using the FILL command. To test COPY2 using Microstar Laboratories DAPview program, enter the following DAPL commands:

```
#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' and put results into pipe
*    'p2'
*/
#include <cdapcc.h>
void rave(PIPE *in_pipe, const int n, PIPE *out_pipe);
void main (PIB **plib)
{
void **argv;
int argc;
argv = param_process (plib, &argc, 3, 3,
            T_PIPE_W, T_CONST_W, T_PIPE_W);
rave ((PIPE *) argv[1], *(const int *) argv[2],
      (PIPE *) argv[3]);
}
void rave (PIPE *in_pipe, const int n, PIPE *out_pipe)
{
int I;
int *d, *max_d;
long int sum = 0;
pipe_open (in_pipe, P_READ);
pipe_open (out_pipe, P_WRITE);
d = (int *) ralloc (2*n);
/* initialize the data array with n values */
for (i=0; i<n; i++) {
   *d = (int)pipe_get(in_pipe);
   sum += *d;
   d++;
}

max_d = d;

/* write new values over the oldest data */
while (1) {
pipe_put (out_pipe, sum / n);
if (d == max_d)
d -= n;
sum -= *d;
*d = (int)pipe_get (in_pipe);
sum += *d;
d += 1;
}
}
```

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 14 in this document.

The following custom 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 with text and sends the
*    string to the PC
*/
#include <cdapcc.h>
void print_data (PIPE *p);
void main (PIB **plib)
{
void **argv;
int argc;
argv = param_process (plib, &argc, 1, 1, T_PIPE_W);
print_data ((PIPE *) argv[1]);
}

void print_data (PIPE *p)
{
    pipe_open (p, P_READ);
    while (1)
    {        printf ("Data = %d \n", pipe_get(p));
    }
}
```

Two additional routines provide transmission of simple strings to output pipes. The function **send** sends a string to the output pipe $SYSOUT. The function **fsend** sends a string to a specified output pipe. These functions have the side effect of altering the contents of the string.

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 requires 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. A blocked get operation reads a specified 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 nonblocked 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.

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 correct 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... */
```

The function **pbuf_put** writes data from a task's pipe buffer into the associated output pipe. Before calling this function, the custom command code must 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 Developer's Toolkit for DAPL provides functions for accessing several important internal fields of the pipe buffer (PBUF) structure. Remember that the pipe buffer belongs to a single task, and it does not interfere with other tasks when the following are used properly:

- 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.
- The function **pbuf_get_cnt** reports the number of data values present in the pipe buffer storage array. This function is particularly useful after data has been fetched into the storage array by the function **pbuf_get**.
- The function **pbuf_set_cnt** specifies the number of data values that have been placed into the storage array. This is typically used before calling the function **pbuf_put**.
- 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, established by the **pbuf_open** function or by the **pbuf_set_max_cnt** and **pbuf_set_min_cnt** functions, are used

by **pbuf_get** to determine how many values should be read into the data array. 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** sets the current number of samples stored in the storage array, so that the next call to the function **pbuf_get_cnt** can report this number to the caller.

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 **pbuf_get_max_cnt** and **pbuf_get_min_cnt** fields 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.

Both the minimum and the maximum data count fields are initialized by **pbuf_open**, but may be reprogrammed after **pbuf_open** is called.

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

There are two advanced techniques that are sometimes useful with blocked pipe input and output. The first is setting the **pbuf_get_min_cnt** field to zero. When this is done, the **pbuf_get** function does not wait for data to arrive, but instead returns whether or not any data is present in the PBUF storage. (Be sure to test for the case that the data count is set to zero upon return.) This technique should be used only when a single custom command must coordinate among several internal processes, and cannot afford to delay while waiting for data to arrive. The second technique is allocating a PBUF with a zero-size storage area. 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_set_data_ptr** to assign a storage array, and adjusting the maximum and minimum data counts accordingly using the

pbuf_set_max_cnt and **pbuf_set_min_cnt** functions. In this way, it is possible, for an input pipe and an output pipe to share the same storage area.

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. It uses the buffer-sharing technique described above for the two output pipes.

```
/*  BCOPY2 (p1, p2, p3)
*      - places copies of data from pipe 'p1' into
*         pipes 'p2' and 'p3'
*/
#include <cdapcc.h>
void bcopy2 (PIPE *p1, PIPE *p2, PIPE *p3);
#define BUF_SIZE 128

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 3, 3, T_PIPE_W,
                                  T_PIPE_W, T_PIPE_W);
    bcopy2 ((PIPE *) argv[1], (PIPE *) argv[2],
            (PIPE *) argv[3]);
}
```

```
void bcopy2 (PIPE *p1, PIPE *p2, PIPE *p3)
{
    PBUF *inbuf, *outbuf1, *outbuf2;
    char *databufin, *databufout;
    int  bufcount;

    pipe_open (p1, P_READ);
    pipe_open (p2, P_WRITE);
    pipe_open (p3, P_WRITE);

    inbuf   = pbuf_open (p1, BUF_SIZE);
    outbuf1 = pbuf_open (p2, BUF_SIZE);
    outbuf2 = pbuf_open (p3, 0);

    databufin  = pbuf_get_data_ptr(inbuf);
    databufout = pbuf_get_data_ptr(outbuf1);
    pbuf_set_data_ptr(outbuf2,databufout);
    pbuf_set_max_cnt(outbuf2,BUF_SIZE);

    while (1)
    {
        pbuf_get (inbuf);
        bufcount = pbuf_get_cnt(inbuf);

        memcpy (databufout, databufin,
            bufcount*sizeof(int));
        pbuf_set_cnt(outbuf1,bufcount);
        pbuf_set_cnt(outbuf2,bufcount);
        pbuf_put (outbuf1);
        pbuf_put (outbuf2);
    }
}
```

Note that the minimum data count for `inbuf` is set to one by default when `inbuf` is opened. This ensures that any available data values are continuously processed, even if the input pipe contains less than `BUF_SIZE` entries. The **`memcpy`** function requires a number of bytes, rather than a number of items, so the multiplier `sizeof(int)` is used. Note that the number of output items is always updated before performing the write operation. The two output pipes share the same data buffer, so only one copy operation is performed.

Another example of blocked pipe operations is a more efficient version of the `ZTRUNC` command. This command reads and processes data in blocks rather than a single value at a time, with 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 negative numbers
*         to zero, and output data to pipe 'p2'
*/
#include <cdapcc.h>
#define  BUF_SIZE 128
void  bztrunc (PIPE *in_pipe, PIPE *out_pipe);

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

void bztrunc (PIPE *in_pipe, PIPE *out_pipe)
{
    PBUF *inbuf, *outbuf;
    int  datacount, *datain, *dataout;
    int  i, tmp;

    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  = pbuf_get_data_ptr(inbuf);
    dataout = pbuf_get_data_ptr(outbuf);
```

```
    while (1)
    {
        pbuf_get(inbuf);
        datacount = pbuf_get_cnt(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);
    }
}
```

Other Pipe Routines

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 width of the pipe, in bytes. A word pipe has a width of two bytes and a long pipe or a float pipe has a width of four bytes.

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

Task Control Routines

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 intertask 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 no longer uses any CPU time.

DAC Access

The Data Acquisition Processor has two on-board digital-to-analog converters (DACs). 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. Because the DAPL operating system is multitasking, **dac_out** updates the DAC in an asynchronous manner. 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 Output Lines

The Data Acquisition Processor provides sixteen digital outputs. To control these lines, call **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 output port. Because the operating system is multitasking, **digital_out** updates the digital port in an asynchronous manner. 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);
    }
}
```

Note: The real-time clock on the DAP 800, DAP 1200e, and DAP 2400e is derived from the CPU clock and provides good long-term accuracy. The DAP 2400 real-time clock is derived from the DAPL time-slicing clock. Depending on system load, this clock can vary by as much as ten percent. DAP 2400 applications requiring long-term timing accuracy should perform timing by counting acquisition samples generated by the input procedure sampling clock.

5. 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 intertask 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 theLIMIT command in the DAPL operating system. LIMIT2 is a signaling task.

```
/*  LIMIT2 (p1, region, t1)
*       - asserts trigger 't1' when data from pipe
*         'p1' enters 'region'
*/
#include <cdapcc.h>
void limit2 (PIPE *, int, int, int, TRIGGER *);

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 5, 5, T_PIPE_W,
             T_RFLAG, T_CONST_W, T_CONST_W, T_TRIGGER);
    limit2 ((PIPE *) argv[1], *(const int *) argv[2],
            *(const int *) argv[3], *(const int *) argv[4],
            (TRIGGER *) argv[5]);
}

void limit2 (PIPE *p, int rflag, int low, int high, TRIGGER
*t)
{
    long int   d;
    pipe_open (p, P_READ);
    trigger_open (t, P_WRITE);
```

```
        while (1)
        {
            d = pipe_get (p);
            if (rflag == R_INSIDE)
            {   /* INSIDE region */
                if   ((d >= low) && (d <= high))
                    trigger_updt_put(t,1);
                else
                    trigger_updt_status(t,1);
            }
            else
            {   /* OUTSIDE region */
                if  ((d < low) || (d > high))
                    trigger_updt_put (t,1);
                else
                    trigger_updt_status(t,1);
            }
        }
    }
```

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)
*      - transfer n1+n2 data values from pipe 'p1'
*         to pipe 'p2' when a trigger assertion
*         occurs in trigger 't1'
*/
#include <cdapcc.h>
void wait2 (PIPE *, TRIGGER *, int, int, PIPE *);

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 5, 5, T_PIPE_W,
             T_TRIGGER, T_CONST_W, T_CONST_W, T_PIPE_W);
    wait2 ((PIPE *) argv[1], (TRIGGER *) argv[2],
          *(const int *) argv[3], *(const int *) argv[4],
          (PIPE *) argv[5]);
}

void wait2 (PIPE *in_pipe, TRIGGER *t, int pretrigger,
    int posttrigger, PIPE *out_pipe)
{
    long int d;
    int I;

    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);
    trigger_open (t, P_READ);
    while (1)
    {
        trigger_wait (t, in_pipe, pretrigger,1);
        for (i=0; i < (pretrigger+posttrigger); i++)
        {
            d = pipe_get (in_pipe);
            pipe_put (out_pipe, d);
        }
        trigger_updt_status (t, (pretrigger+posttrigger));
    }
}
```

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.

```
/*  TSTAMP2 (t)
*      - prints the assertion count of all assertions
*        that appear in trigger 't'
*/
#include <cdapcc.h>
void main (PIB **plib)
{
    void **argv;
    int argc;
    TRIGGER *  trig;
    unsigned long int  assertion;

    argv = param_process (plib, &argc, 1, 1, T_TRIGGER);
    trig = (TRIGGER *) argv[1];
    trigger_open (trig,P_READ);

    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 an assertion occurs every N samples
*     otherwise, signals trigger tout and terminates
*/
#include <cdapcc.h>
void  main (PIB **plib)
{
    void **argv;
    int argc;
    TRIGGER   *tin, *tout;
    unsigned long int   baseline, status;
    unsigned long int   N;
    int  flag;

    argv = param_process (plib, &argc, 3, 3,
      T_TRIGGER, T_CONST_W, T_TRIGGER);
    N = *(int *) argv[2];
    tin  = (TRIGGER *) argv[1];
    tout = (TRIGGER *) argv[3];
    trigger_open(tin,P_READ);
    trigger_open(tout,P_WRITE);
    baseline = 0xFFFFFFFF;

    while (1)
   {
        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();
    }
}
```

6. 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 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. Only the 32-bit `float` type is compatible with DAPL pipes.

The Toolkit Libraries

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 the 80x486 processors have an on-chip floating point unit (FPU). When a hardware FPU is available, the FPU executes the floating point operations specified by a custom command. 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.

The Developer's Toolkit for DAPL library has four versions. Two versions support operation in the 16-bit DAPL version 4 environment, and two versions support the 32-bit DAPL2000 environment. For each of the two operating systems, both versions of the library are provided:

`SMALL`

- This library version does not support any floating point operations or floating point formatting conversions. This version of the library produces the smallest and most efficient code. It uses the least amount of resources on the Data Acquisition Processor and runs the fastest. It is compatible with all Data Acquisition Processor models and DAPL versions.

`FP`

- This library version provides full IEEE floating point support, including formatting conversions. When floating point coprocessor hardware is available on the Data Acquisition Processor, this library uses it. Otherwise, this library automatically uses the floating point emulation software built into the operating system. The code size is a little larger, and there is slightly more system overhead. This library is available for all Data Acquisition Processor models except the ones with software in ROM. It requires DAPL version 4.3 or later, or any version of DAPL2000.

The appropriate library version is selected at compile time by the `SMALL` or `FP` switch (in all capitals) on the DOS command line. Use the MCC4 or BCC4 batch file for 16-bit custom commands compatible with DAPL version 4, or use the MCC16 or BCC16 batch files for 16-bit custom commands compatible with DAPL 2000.

Floating Point Library Functions

The floating point library functions provided with C compilers will not work in the Data Acquisition Processor environment. The Developer's Toolkit for DAPL replaces the compiler's standard floating point library functions with equivalent functions. These functions, in addition to working in the DAPL environment, are smaller, faster, and make better use of a hardware FPU when available.

The library functions are fully compatible with the corresponding functions provided in your compiler libraries. No changes to a custom command's C code or to the libraries provided with the compilers are required to use the Developer's Toolkit for DAPL floating point libraries. The function names, parameters, and return values are exactly the same. You may include the `MATH.H` file into your source. The modified library functions are automatically included by specifying the `FP` library at compile time.

The following table lists math library floating point functions that are supported by the Developer's Toolkit for DAPL.

acos	atan	fabs	log10
asin	atan2	floor	modf
atan	atof	fmod	pow
atan2	ceil	frexp	sin
acos	cos	hypot	sqrt
asin	exp	log	tan

The Bessel functions are not supported. The hyperbolic functions and their inverses are not supported by the `FP` library.

Compiler Limitations

The Microsoft and Borland compilers are alike in most ways, but they differ in their handling of floating point. Most of the differences are resolved by the `msftfp4.obj`, `msftfp16.obj`, `borlfp4.obj`, or `borlfp16.obj` files. These are linked automatically with the custom command when it is compiled using the `FP` command line switch. One difference, however, cannot be corrected in this manner. When using a Microsoft compiler, the return value from a floating point function is stored in memory, and a pointer to the value is passed back to the calling function. Currently, this is not compatible with the `EXEPROC` relocation utility. Until a solution is found, it will not be possible for custom commands compiled with Microsoft compilers to return a floating point value from a function directly. Instead, an extra pointer parameter must be passed to the function, and the return value must be stored in that location. This problem does not affect functions in the Developer's Toolkit for DAPL floating point function library.

For example, the following is fine for the Borland compilers, but will fail for Microsoft compilers:

```
float my_function(float);
float ff, result;
.
.

ff = my_function(ff);
```

The solution is to change the local function `my_function` to store the resulting value using a passed pointer, returning no result.

```
void my_function(float, float *);
float ff, result;
.
.
my_function(ff,&result);
```

The `FP` library gives total control of real or emulated FPU hardware. See any 80x386 or 80x486 assembly language programming manual for details of the FPU instruction set. Most of the FPU capabilities are supported by the inline assembly features of the C compilers, but some are not. Separate assembly instructions, coded as a subprogram for C language, will make these additional instructions available to advanced floating point programmers.

Using Pipes

There are a few special considerations when using PIPE inputs and outputs for floating point values. These features are available only when using the FP version of the Developer's Toolkit for DAPL.

There are three special facilities of DAPL available for passing floating point data to and from a custom command.

For a pipe that passes floating point data from one custom command to another, declare the PIPE to have FLOAT type in the DAPL command file. For example:

```
PIPE PF1 FLOAT
TRIGGER T1
 .
 .
PDEF A
FLTCMD1(IPIPE0,PF1)
FLTCMD2(PF1,T1)
END
```

In this example, the custom command FLTCMD1 performs a floating point analysis on data from input channel pipe 0, and writes floating point data into pipe PF1. Custom command FLTCMD2 checks floating point data in pipe PF1 and generates trigger T1 if certain special conditions are detected.

For obtaining floating point data from the PC, a communications pipe may be defined using the CPIPE command, as follows.

```
CPIPE FLIP INPUT FLOAT
```

For transferring floating point values to the PC, use a similar definition to set up an output communications pipe:

```
CPIPE FLOP OUTPUT FLOAT
```

It is possible to transfer floating point data to and from the PC on the normal binary data channel using the MERGE command:

```
PIPE FL1 FLOAT, FL2 FLOAT

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

When using the function **param_process** to check parameter types, use the T_PIPE_FL code for floating point pipes.

A custom command can define, compute, and store data values in double precision or single precision floating point. However, transfers through DAPL pipes are in single precision, 32-bit representation only. This constraint should seldom be a problem, since a single-precision floating point value has approximately seven significant decimal digits. If it is necessary to retain all 64 double-precision bits, place the 64-bit double precision values into the buffer storage array of a float pipe, and use the **pbuf_set_cnt** field to double the number of double-precision values present. DAPL will transfer the data via the 32-bit pipe, but the receiving task must extract the values pairwise, and reassemble the pairs of 32-bit codes to restore the original 64-bit double precision value.

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 standard output stream is the DAPL text pipe to the PC, and the **fprintf** function writes to a pipe rather than to an output stream. The format conversions for the FP library are compatible with Standard C.

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.

Example Application

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)
*    - computes a scientific function of pipes 'p1'
*      and 'p2' and sends the results to pipe 'fp3'
*/
#include <math.h>
#include <cdapcc.h>
void main (PIB **plib)
{
void **argv;
int argc;
PIPE *p1,*p2,*fp3;
```

```
/* Check and extract parameters */
argv = param_process (plib, &argc, 3, 3, T_PIPE_W,
                      T_PIPE_W, T_PIPE_FL);
p1 = (PIPE *) argv[1];
p2 = (PIPE *) argv[2];
p3 = (PIPE *) argv[3];

/* Perform initializations. Open all pipes. */
pipe_open (p1,P_READ);
pipe_open (p2,P_READ);
pipe_open (fp3,P_WRITE);

/* Perform the real time processing */
while (1)
{
float x1, x2, y;

/* read inputs and scale */
x1 = (float) (pipe_get(p1) * 3.0518e-5);
x2 = (float) (pipe_get(p2) * 3.0518e-5);

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

/* send the floating point result */
pipe_put_float (fp3, y);
}
}
```

In the above 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 FP versions of the library support the errno feature defined in the Standard C library file ERRNO.H. The following error codes are used:

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

To use the error checking feature, include the ERROR.H at the top of the custom command C code. This will define an errno variable. (In the jargon of the system programmer, errno is actually a thunk. See the file ERRNO.H for your compiler for more information on how it is implemented. Treat it as a simple integer variable.) Before the floating point operation or operations under test, set the errno variable to zero. Then, after the operation or operations, test it again for a nonzero value.

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

```
#include <errno.h>
double a,b,c,tanh
  .
  .

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

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

The FP library does not support the non-standard matherr function provided by the Microsoft and Borland runtime libraries. The matherr function is useful only for errors that occur inside math library functions. The errno mechanism is effective for any sequence of floating point computations, whether library functions are used or not.

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 desired 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. This word can be examined using inline assembly to determine the exact nature of the error. The value of the code stored in the `errno` variable is not directly related to the code in the status word. 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. This could be a useful diagnostic technique in some situations, but changing the FPU exception masks is not recommended.

7. 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.

These functions are supported only by DAPL 2000. See Appendix A for more information about compatibility with Data Acquisition Processor models that have a DSP 56000 coprocessor and use DAPL 4.

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 = 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 C-language 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:

```
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 */
int  * tone_buffer,  errcode;
tone_buffer = ralloc( 9000*sizeof(int) );

/* Construct the three parts of the waveform */
errcode  = isinewave( 4000, 100, sizeof(int),
   1, tone_buffer);
errcode |= isinewave( 1000, 100/4, sizeof(int),
   1, (tone_buffer+4000) );
errcode |= isinewave( 4000, 100, sizeof(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 computation services are requested, and results accessed, by means of the following sequence of functions:

- **`fft_init`**
- **`fft_request`**
- **`fft_status`**
- **`fft_receive`**

The **`fft_init`** function defines the properties of an FFT, and is further described in the next section of this chapter. It is executed once during command startup.

The **`fft_request`** function initiates FFT computation, using the transform characteristics previously defined by the **`fft_init`** function.

The remaining two functions guarantee synchronization between the DAPL computations and the custom command. The **`fft_status`** function verifies that the FFT computation is complete. The **`fft_receive`** function guarantees that the FFT results are stored and ready for further processing. If portability between different Data Acquisition Processor models is not a consideration, and the custom command is intended for operation on a specific Data Acquisition Processor model which does not use a separate processor for DSP computations, these two steps may be omitted. In general, skipping these steps is not recommended practice.

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
real_buf = ralloc(FFTSIZE);                 /* wrong! */
imag_buf = ralloc(FFTSIZE);
real_buf = ralloc(FFTSIZE * sizeof(int));   /* right! */
imag_buf = ralloc(FFTSIZE * sizeof(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
real_buf = ralloc(FFTSIZE * sizeof(int));       /* wrong! */
real_buf = ralloc(FFTSIZE * 2 * sizeof(int));   /* right! */
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
int   * input_real;
long  * output_long;

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

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 clean input signal that does not significantly contribute to the noise level, frequency peaks as small as 1/8 of the least-significant bit of the sampling resolution should 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. Note that this behavior is different from previous versions of the Developer's Toolkit for DAPL, which split most-significant and least-significant bits of the power results into the realbuf and the imagbuf respectively.

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.
- In previous versions of the Developer's Toolkit for DAPL, it was necessary to fill the imaginary input terms with zeroes when the input values were strictly real. This is no longer the case. As described previously in this chapter, 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.

- Application notes for previous Developer's Toolkit for DAPL versions have suggested a computational “trick” in which two real-valued FFTs are computed in a single operation, by treating the two real-data streams as `real` and `imaginary` terms of a complex FFT. This “trick” is no longer recommended. In effect, it is already built into the real-data FFT computations and is applied automatically. Use two real-data FFT operations instead.
- 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 built-in `FFT` command in DAPL. 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.

```
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 is n
*      - output placed into pipe p2
*      - output is magnitude values
*/
#include <cdapcc.h>
#define   NULL    0
#define   FOREVER 1

void main (PIB **plib)
{
    void **argv;
    int  argc;
    PIPE *in_pipe;      /* Input pipe, real data */
    PIPE *out_pipe;     /* Output pipe, magnitude data */
    int  n;             /* Size of FFT input and output
block */

    PBUF *inbuf, *outbuf;
    FFTB *fft;
    int  *databuf;

    /* PARAMETER PROCESSING SECTION */
    argv = param_process (plib, &argc, 3, 3,
        T_PIPE_W,T_CONST_W, T_PIPE_W);
    in_pipe = (PIPE *) argv[1];
    n = *(const int *) argv[2];
    out_pipe = (PIPE *) argv[3];
```

```
    /* INITIALIZATION SECTION */
    /* 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 = 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);

    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 reals out
*/
    if (fft == NULL )
      param_error();

    /* RUN-TIME PROCESSING LOOP */
    while ( FOREVER )
    {
        pbuf_get(inbuf);
        fft_request(fft);
        while (!fft_status(fft))  task_switch();
        fft_receive (fft);
        pbuf_set_cnt(outbuf,n);
        pbuf_put(outbuf);
    }
}
```

The Fast Fourier transform is the basis for many powerful signal processing algorithms. The following example illustrates a cepstrum computation using a fast Fourier transform. Cepstrum is useful for some types of mechanical vibration analysis. The cepstrum of an input signal is computed by:

- performing a forward Fourier transform of an input signal,
- computing the logarithm of the power of each input frequency component,
- performing an inverse Fourier transform on the logarithm data.

Some floating point operations are required, so this command must be compiled using the FP library option as described in Chapter 12.

The computation is applied to non-periodic data, so either a built-in window operator or a user-supplied vector must be specified. This example shows how to access window vector information specified by a VECTOR command.

The following listing of the cepstrum custom command has a few interesting features. FFTPOST_POWER post-transform processing is applied, but the FFT_FULLOUT option is used to preserve term N/2 and to avoid combining power terms from the low and high end of the transform spectrum. Data storage is provided for N long output values (rather than N word values), but only half of this storage is used for data input. The even symmetry of the power data is still present after the logarithm operation is applied, and this fact is used to advantage to avoid unnecessary log function evaluations. When an FFT is applied to data with even symmetry (forward or reverse), the resulting imaginary terms are zero, hence the inverse transform generates a full set of N real output values.

```
/*
** CEPSTRUM (p1,
n, window, p2)
**      - compute cepstrum from input data
**      - real data taken from pipe p1
**      - n is size of FFT input and output block
**      - window specifies built-in option or VECTOR
**      - output sent to pipe p2
*/
#include  <math.h>
#include  <cdapcc.h>
#define   NULL   0L
#define   FOREVER  1
```

```
void main (PIB **plib)
{
    void **argv;
    int  argc;

    PIPE *in_pipe;
    PIPE *out_pipe;
    int  n;
    unsigned long window;

    VECTOR *windvect;
    PBUF *inbuf, *outbuf;
    FFTB *fft1, *fft2;
    int  *int_buffer;
    long *long_buffer;

    double logpower;
    double scale = 16384.5 / log(32767.0);
    int  I;

    /* PARAMETER PROCESSING SECTION */
    argv = param_process (plib, &argc, 4, 4,
        T_PIPE_W, T_CONST_W, T_CONST_W|T_VECTOR_L,
T_PIPE_W);

    in_pipe = (PIPE *) argv[1];
    n = *(const int *) argv[2];
    out_pipe = (PIPE *) argv[4];

    if (param_type(plib,3)==T_CONST_W)
    {   /* Predefined window type */
        window = *(const int *) argv[3];
    }
    else
    {   /* User-defined window data */
        windvect = (VECTOR *) argv[3];
        if ( vector_length(windvect) != n )
          param_error();
        window = (unsigned long) windvect;
    }
```

```
/* INITIALIZATION SECTION */
pipe_open (in_pipe, P_READ);
pipe_open (out_pipe, P_WRITE);

/* Open pipes sharing a buffer for n longs */
inbuf  = pbuf_open(in_pipe, 2*n );  /* Extra storage */
outbuf = pbuf_open(out_pipe,0);
pbuf_set_min_cnt(inbuf,n);
pbuf_set_max_cnt(inbuf,n);
pbuf_set_min_cnt(outbuf,n);
pbuf_set_max_cnt(outbuf,n);
int_buffer = pbuf_get_data_ptr(inbuf);
pbuf_set_data_ptr(outbuf,int_buffer);
long_buffer = (long *) int_buffer;

/* initialize forward transform options */
fft1 = fft_init( n, int_buffer, NULL,
    window,           /* Passed window parameter */
    FFTDIR_FORWARD,   /* Use forward transform */
    FFTSOLN_ACCURATE, /* Accuracy most important */
    FFTPOST_POWER,    /* Convert to 32-bit POWER */
    FFT_REALIN|FFT_FULLOUT );
if (fft1 == NULL )
    param_error();

/* initialize reverse transform options */
fft2 = fft_init( n, int_buffer, NULL,
    NULL,             /* No window operation */
    FFTDIR_REVERSE,   /* Use reverse transform */
    FFTSOLN_ACCURATE, /* Accuracy most important */
    FFTPOST_REAL,     /* Select real components */
    FFT_REALIN|FFT_FULLOUT);
if (fft2 == NULL )
    param_error();
```

```
    /* RUN-TIME PROCESSING LOOP  */
    while ( FOREVER )
    {
        /* transform real valued input data to power */
        pbuf_get(inbuf);
        fft_request(fft1);
        while (!fft_status(fft1))  task_switch();
        fft_receive (fft1);

        /* compute the logarithms and then apply symmetry */
        for (i=0; i<=n/2; I++)
        {
            logpower = (double) long_buffer[i];
            if (logpower > 0.0)  logpower = log(logpower);
            int_buffer[i] =  (int) (logpower * scale);
        }
        for (i=1; i<n/2; ++I)
            int_buffer[n/2+i] = int_buffer[n/2-i];

        /* perform the inverse transform and send results */
        fft_request(fft2);
        while (!fft_status(fft2))  task_switch();
        fft_receive (fft2);
        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`**
- **`fir_request`**
- **`fir_status`**
- **`fir_receive`**

The **`fir_init`** function defines the characteristics of a FIR filter. This function is executed once during command startup.

The **`fir_request`** function supplies the FIR filter with a block of data in a transfer buffer, and initiates FIR filter computation, using the filter characteristics previously defined by the **`fir_init`** function.

The remaining two functions guarantee synchronization between the DAPL computations and the custom command. The **`fir_status`** function verifies that the FIR computation is complete, and reports the number of computed results which were generated. The **`fir_receive`** function guarantees that the FIR filtering results are stored in the transfer buffer and ready for further processing by the custom command.

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. 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."

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:

```
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.

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.

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.

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.

FIR Filter Status

After a call to the **fir_request** function, the **fir_status** function will report the number of computed results which are available.

If the filter has not completed the computations for the data block, the **fir_status** function returns a negative code. If the data provided to the filter shift register did not fill the shift register, hence no outputs, the **fir_status** function returns zero. If the filter computations for the data block are complete, a positive count of computed results is returned.

Accessing FIR Results

The **fir_receive** function guarantees that all computed results reported by the **fir_status** function are stored and available to the custom command, replacing data originally passed to the FIR filter by the call to the **fir_request** function. The `data` array must not be used for other purposes between the **fir_request** and **fir_receive** function calls.

If portability between different Data Acquisition Processor models is not a consideration, and the custom command is intended for operation on a specific Data Acquisition Processor model which does not use a separate processor for DSP computations, the **fir_status** and **fir_receive** steps may be omitted. In general, skipping these steps is not recommended practice. When there is no separate processor for DSP computations, computed results are available immediately upon return from the **fir_request** function. The return code from the **fir_request** function reports the number of samples available, exactly as described above for the **fir_status** function. When a separate processor for DSP computations is used, the **fir_request** function returns a negative code.

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 for purposes of data display.

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 custom command implements the filter.

```
/*
**  LSFILTER (p1, p2)
**   - reads data from pipe p1
**   - applies 7-point least-squares smoothing
**   - places results into pipe p2
*/
#include  <cdapcc.h>
#define   NULL   0L
#define   FOREVER  1

static int  ls7vect[7] =
{  -6241, 9362, 18724, 21845, 18724, 9362, -6241 } ;

void main (PIB **plib)
{
    void **argv;
    int  argc;

    PIPE *in_pipe;        /* Passed parameters */
    PIPE *out_pipe;
    FIRB *fir;            /* To be initialized */
    int  avail;           /* Utility variables */
    int  I;

    /* Using an array avoids compiler aliasing */
    int  sample[1];

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

    /* INITIALIZATION SECTION */
    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);

    fir = fir_init (ls7vect,
        7, /* length */
        2, /* scale factor */
        0  /* no decimation */ );
    if (fir == NULL)
        param_error();
```

```
    /*
    ** Compensate for the 3-sample delay of a 7-term
    ** symmetric filter.
    */
    for ( i=0; i<3; ++i) pipe_put(out_pipe,0L);

    /* RUN-TIME PROCESSING LOOP  */
    while ( FOREVER )
    {
        /* Send each input sample to the filter */
        sample[0] = (int) pipe_get(in_pipe);
        fir_request(fir,sample,1);
        while ((avail=fir_status(fir))<0)
            task_switch();
        fir_receive(fir);

        /* Output any returned value */
        if (avail>0)
            pipe_put(out_pipe,sample[0]);
    } /* end of run-time loop */
}
```

An EEG Filtering Example

The purpose of this second example is to monitor alpha brain waves. This might be used, for example, to detect REM sleep, or as biofeedback to assist relaxation. As in most medical applications, safety is the most important consideration, and special high-impedance instrumentation amplifiers provide electrical isolation to protect the patient. These amplifiers are presumed to have a natural high-frequency rolloff which attenuates high frequencies, avoiding aliasing problems when sampling at 60 Hz.

The digital filter selects frequencies in the range 8 to 11 Hz. There is a slight rolloff at 12 Hz, with very high attenuation beyond 12 Hz. The transition to the stopband at lower frequencies is more gradual.

In this example, the data vector is specified in the DAPL configuration file, so that it can be shared by a number of signal channels.

```
VECTOR  EEG = ( -1, 37, 50, -43, -180, -163, 89, 366, 339,
  -50, -449, -445, -43, 305, 231, -74, -69, 412, 762, 196,
  -1175, -2067, -1151, 1370, 3450, 2882, -454, -4015, -4620,
  -1384, 3216, 5355, 3216, -1384, -4620, -4015, -454, 2882,
  3450, 1370, -1151, -2067, -1175, 196, 762, 412, -69, -74,
  231, 305, -43, -445, -449, -50, 339, 366, 89, -163, -180,
  -43, 50, 37, -1 )
```

The signed sum of the coefficients is greater than 32768, so this filter has the effect of slightly amplifying alpha frequencies as it attenuates other frequencies. This leaves the filter output subject to possible saturation, but this is considered an unimportant transient condition. The absolute sum of the coefficients equals 65531, so the filter output will saturate correctly if overflow does occur. Precision of the frequency response is not critical, so scaling is not applied. This application is not sensitive to the delay induced by the filter, so no phase correction is applied.

The following is a listing for the alpha wave filtering command:

```
/*
**  BWAVE (p1, vect, p2)
**     - selects 8-12 Hz frequencies from 60 Hz sampling
**     - filter vector provided by DAPL command file
**     - filters data from p1
**     - places results into pipe p2
*/
#include  <cdapcc.h>
#define   FOREVER         1
#define   NULL            0L
#define   BUFFER_LENGTH   128

void main (PIB **plib)
{
  void **argv;
  int  argc;

  PIPE *in_pipe;
  PIPE *out_pipe;
  VECTOR *vect;
  FIRB *fir;
  PBUF *inbuf;
  PBUF *outbuf;
  int  v_length;
  int  *v_start;
  int  avail;
  int  *dataptr;
  int  datacount;

  /* PARAMETER PROCESSING SECTION */
  argv = param_process (plib, &argc, 3, 3,
      T_PIPE_W,T_VECTOR_W, T_PIPE_W);
  in_pipe = (PIPE *) argv[1];
  vect = (VECTOR *) argv[2];
  out_pipe = (PIPE *) argv[3];

  /* INITIALIZATION SECTION */
  /* Determine the filter length from the DAPL vector */
  v_length = vector_length(vect);
  v_start = vector_start(vect);
```

```
  /* Prepare pipes with shared input/output buffer */
  pipe_open (in_pipe, P_READ);
  pipe_open (out_pipe, P_WRITE);
  inbuf = pbuf_open(in_pipe, BUFFER_LENGTH);
  dataptr = pbuf_get_data_ptr(inbuf);
  outbuf = pbuf_open(out_pipe, 0);
  pbuf_set_data_ptr(outbuf,dataptr);
  pbuf_set_min_cnt(outbuf, 1);
  pbuf_set_max_cnt(outbuf, BUFFER_LENGTH);

  /* Set up filter to use shared vector */
  fir = fir_init (v_start, v_length,
    0, /* no scaling */
    0  /* no decimation */ );
  if (fir == NULL)
    param_error();

  /* RUN-TIME PROCESSING LOOP  */
  while ( FOREVER )
  {
    /* Apply input samples to the filter */
    pbuf_get(inbuf);
    datacount = pbuf_get_cnt(inbuf);
    fir_request(fir,dataptr,datacount);
    while ((avail=fir_status(fir)) <0)
      task_switch();
    fir_receive(fir);
    if ( avail>0 )
    {
      pbuf_set_cnt(outbuf,avail);
      pbuf_put(outbuf);
    }
  }  /* end of run-time loop */
}
```

8. 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.

A real-time system is often required to monitor a continuous input quantity by sampling it at regular intervals. It may be required to respond to input events on many input channels, with different input rates on each channel. To meet these requirements, most real-time systems employ "interrupt-driven processing," in which a computing resource is applied upon demand, and then released for other processing. This capability provides efficient utilization of a shared computing resource, but 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."

Interrupt-driven control is possible in the native processor of an 80x86-based PC, but the latency depends on the hardware and software configuration. For example, an application using DOS extender software on a PC with a 80x286 processor can take about 250 microseconds to switch between real and protected mode, and during this time interrupts cannot be serviced. The 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 the control task. The kernel services of the DAPL and DAPL 2000 operating 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 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. In Chapter 9, the `BPID2` processing

command illustrates one application that compromises response latency for individual inputs to optimize the response latency for a large block of input channels.

Another characteristic of many 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 and DAPL 2000 operating systems provide means of adjusting the trade-offs between processing speed and latency. DAPL 2000 provides `SCHEDULING`, `QUANTUM` and `BUFFERING` options. DAPL version 4 provides two pre-defined combinations of these.

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 very 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, leaving less computing resources for other processing.

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.

A fourth strategy is turning off DAPL multitasking. This reduces the multitasking overhead to zero and provides the fastest possible speed, but it also turns off all of the services that multitasking provides. This advanced topic is detailed in Chapter 10.

Using Floating Point

On Data Acquisition Processor models which have a Floating Point Unit (FPU), DAPL will use it to perform floating point operations. The FPU can perform floating point operations almost as fast as the main processor performs ordinary instructions. This capability can be very attractive in some control applications.

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.

9. Customizing PID Control

This chapter will show how Developer's Toolkit for DAPL services can be used to configure customized Proportional-Integral-Derivative (PID) control applications. In the process, this chapter illustrates the design and construction of a practical real-time controller for several simultaneous PID loops. The structure of this example can serve as a model for other real-time applications as well.

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 set 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 `PID` command provided in DAPL. The `PID` command is optimized for controlling a single PID loop. Though very general, the standard `PID` command has some limitations. It provides great flexibility for adjustment of parameter values, but there is a small execution speed penalty. 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.

Designing Control Commands

This section will examine the general structure of a control task, and cover the Developer's Toolkit for DAPL services which are useful in building customized PID control functions.

The following is a template for custom real-time PID control commands.

```
Decode command parameters
Set up system structures: pipes, buffers, etc.
Set up PID control parameter structures
Establish initial PID parameter values
Real-time update loop
```

This is not much different from the structure of any custom command built with the Developer's Toolkit for DAPL. The third and fourth steps set up and initialize two special data structures which are required for PID control. The last step is a process loop, which will read digitized samples, perform the control computations, and produce the control output.

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 poor 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.

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 output to DAC
  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 **pid_update** function 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. Optionally, this value can be modified by an additional control algorithm. 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.

Example Applications

This section applies the Developer's Toolkit for DAPL PID control functions to two very different PID control applications. The first controls a single PID loop, using a minimum configuration for lowest response latency. The second controls a large number of PID loops, with block updates for maximum efficiency and predictable performance.

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 follows the general form for real-time commands as described earlier in this chapter, except that it does not make any control parameter adjustments and 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 (p1, vdac)
**  Fixed, single-loop PID control command with minimum
latency
**    - parameters fixed at compile-time
**    - reads system output feedback from 'p1'
**    - sends control outputs to DAC specified by 'vdac'
*/
```

```
#include        "cdapcc.h"

#define  FOREVER        1
#define  OKAY           0
#define  INITIAL_STATE  0
void     main  ( PIB ** );
void     real_time_updates( PIPE * inpipe, int dac_id );
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;  */
   } ;

/*
** SPID2 command main routine.  Initialize structures for
**   input pipes and PID control.
*/

void  main ( PIB ** plib )
{
    void    ** parameters;          /* parameters from plib
*/
    PIPE    * in_pipe;
    int     dac_id;
    int     n_params;
```

```
    /* Obtain parameters and open feedback data pipe  */
    parameters =  param_process(plib, &n_params, 2, 2,
          T_PIPE_W, T_CONST_W );
    in_pipe =    (PIPE *) parameters[1];
    dac_id =    * (int *) parameters[2];
    pipe_open(in_pipe,P_READ);

    /* Set up PID control parameters */
    PID_block = pid_open( INITIAL_STATE );
    pid_tune(PID_block, &coeffs);

    /* Perform real-time updates.  Does not return.  */
    real_time_updates( in_pipe, dac_id );

}  /*  End of SPID2 main function  */

/*
** Real-time update loop for PID control.  The output sign
**  is not inverted, consistent with the DAPL PID command.
*/
void   real_time_updates ( PIPE * in_pipe, int  dac_id )
{
    int  sample;

    while (FOREVER)
    {
        sample = (int) pipe_get(in_pipe);
        dac_out ( dac_id , pid_update(PID_block, sample) );
    }  /* End real-time loop */
} /* End of real-time update function */

/* End of SPID2 custom command */
```

In contrast to the `SPID2` command, which controls a single PID loop, the `BPID2` command updates a large number 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 plus the time to compute all of the updates. This 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 both much better than would be achieved by an equivalent number of independent DAPL `PID` 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 follows the general form for real-time commands, except that it does not perform any supplementary control functions, it uses fixed setpoint and parameter values, it uses a pipe as a source of data for initializing PID parameters, and it uses 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 completed C code for the `BPID2` command.

```
/*
**  BPID2 (p1, p2, n, vdac)
**  Blocked-PID custom 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'
*/

#include        "cdapcc.h"

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

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

static PID   * PID_blocks [iMaxPIDLoops] ;

struct tagged_PIDCOEF  {
    int   tag;
    PIDCOEF  coeffs;
    };
```

```
/*
** BPID2 command main routine.  Initialize structures for
**   input pipes and PID control.  Start the real-time loop.
*/
void  main ( PIB ** plib )
{
    void    ** parameters;        /* parameters from plib */
    PIPE    * in_pipe, * param_pipe;
    VECTOR  * DACs;
    int     blocksize;
    PBUF    * coef_buf, * in_buf;   /* other system data */
    int     * samples;
    int     channel;                /* PID control */
    struct  tagged_PIDCOEF  * coeff_set;
    PID     * current_PID;

    /* Strip parameters from the parameter structure  */
    parameters =  param_process(plib, &channel, 4, 4,
          T_PIPE_W, T_PIPE_W, T_CONST_W, T_VECTOR );
    in_pipe =    (PIPE *) parameters[1];
    param_pipe = (PIPE *) parameters[2];
    blocksize = * (int *) parameters[3];
    DACs = (VECTOR *) parameters[4];

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

```
    /* Prepare pipe for fetching feedback data input in
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 PID parameter training pipe.  Access
parameter
    **  data directly from buffer storage.  Make buffer
allocation,
    **  cover exactly one tagged_PIDCOEFF structure.
    */
    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 block of input samples for initialization.
*/
    pbuf_get(in_buf);
    samples = (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 all 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();
    }
```

```
    /* Perform real-time updates.  Does not return.  */
    real_time_updates( in_buf, DACs, blocksize );
}  /*  End of BPID2 main function  */

/*
** Real-time update loop for PID control.  The output sign
**  is not inverted, to keep it consistent with the DAPL
**  PID command.
*/
void   real_time_updates ( PBUF * in_buf, VECTOR * DACs,
           int blocksize)
{
    int  * samples;
    int const * outputs;
    int  channel;

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

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

10. Multitasking Support

This chapter is an advanced topic which applies to control applications requiring the fastest possible real-time processing and the smallest possible response latency. A special Developer's Toolkit for DAPL function can halt DAPL multitasking and dedicate all CPU resources to a single control task. When this is done, only essential interrupt-driven DAPL services remain active. Very few control systems require this extra margin of capability, but it is available for the systems that need it.

Changing multitasking operation is an option that should be used with the greatest care. There are numerous hazards that occur when multitasking stops. DAPL services that you would normally take for granted cease to function. Error checking and data formatting services do not respond. Buffering of input samples is severely limited. User pipes are of little use, because other tasks have no opportunity to read from them. All functionality for the application must be custom-programmed and built into a downloaded custom command.

Suspending and Resuming Multitasking

The **sys_set_multitasking** function gives the custom command direct control of DAPL multitasking operation. The parameter specifies the multitasking mode.

Calling the **sys_set_multitasking** function with parameter `eMultiOff` turns task switching off. Multitasking remains off until one of the following occurs:

- the task calls the **sys_set_multitasking** function with parameter `eMultiOn`
- input channels overflow
- output channels underflow
- the task terminates

Some of the events that terminate a task include a call to **exit**, a parameter error in **param_process**, or a call to **param_error**.

Calling the **sys_set_multitasking** function with parameter `eMultiOffSYSIN` turns task switching off, and additionally allows multitasking to resume when the Data Acquisition Processor receives any character from the PC interface through the `$SYSIN` communication pipe.

The `eMultiOff` parameter is used for time-critical control processes that must not be interrupted by DAPL commands before completion. After calling the **sys_set_multitasking** function with parameter `eMultiOff` to turn off multitasking, commands sent to the DAPL command interpreter are received and buffered, but not interpreted. Commands remain unprocessed in the communication buffers, either on the Data Acquisition Processor or in the PC's ACCEL driver buffers, until multitasking resumes. This means that sending a `STOP` or `RESET` command is not sufficient to terminate operation of the real-time command while multitasking is off. If multitasking is turned back on, the buffered commands are processed, and any error messages generated during task termination are sent to the PC in the normal manner. Control messages also can be sent directly to the custom command while multitasking is off, avoiding the DAPL interpreter. One way to do this is to use the `$BININ` pipe as an auxiliary input to the custom command. The custom command must occasionally call **pipe_num** o see if the PC has sent a message, and if so, read the message codes and take the appropriate action. If `$BININ` is already used for other purposes, or if text mode is required, a separate communications pipe can be defined in DAPL using the `CPIPE` command.

The `eMultiOffSYSIN` parameter is used for control applications which need to be easily started and stopped without resetting the **sys_set_multitasking** function.

After calling the **sys_set_multitasking** function with parameter `eMultiOffSYSIN` to turn off multitasking, sending a `STOP` or `RESET` command enables multitasking and terminates operation of the real-time command.

Note: If the DAPL interpreter receives other commands, or portions of a command, multitasking will be turned on. The real-time control task continues to run, but with multitasking enabled. While multitasking is on, data backlog, response delays, or input channel overflow can occur. Be particularly careful with PC applications that send commands to the Data Acquisition Processor one character at a time.

Available Services with Multitasking Off

The following DAPL services remain available when multitasking is off:

- Reading from input channel pipes or communication pipes
- Writing to output channel pipes or communication pipes
- Writing to asynchronous output devices
- Interrupt-driven data sampling
- Interrupt-driven output updates
- Interrupt-driven PC data transfers
- Real-time clock services
- All Developer's Toolkit for DAPL system functions except the **sys_exec_command** function

Be careful when using the input burst mode. When multitasking is off, the system services which normally re-arm the burst mode hardware are not scheduled. The next burst will not be initiated unless all data from a previous burst is removed from all input channel pipes. This could introduce timing problems for burst events which occur close together. To minimize this problem, when using input burst mode, use an input channel list containing all of the input channel pipes. In the custom command, read and store all data from the multiplexed input data first, to re-arm the burst mode, and then analyze the data.

Input Procedure Buffering

Dynamic input channel buffering stops when multitasking is turned off. While multitasking is active, DAPL continuously allocates as much onboard memory as it needs to buffer input data samples. This is not the case when multitasking is off. It is necessary to use the following special command in the input procedure:

```
BUFFERS STATIC
```

This command goes into effect when the input procedure containing it is started, and it remains in effect until another input procedure is started. `BUFFERS STATIC` forces the input procedure to use a small, fixed region of memory for data buffering. The active task must read data from the input channel data buffers continuously, so that the input channel pipe buffer storage is not exhausted.

The input channel buffer area is large enough to store the data captured in 100 milliseconds of continuous input procedure operation. The real-time custom command must read all available data from the input channel pipe at least once every 100 milliseconds. The data can be read and processed an item at a time, or it can be read as a block. The routine **`pbuf_get`** can safely capture long data blocks, even blocks which require sampling for more than 100 milliseconds, because input channel pipe data is copied from the input channel into a pipe buffer. Processing of a long data block must be completed within 100 milliseconds, however, so that **`pbuf_get`** can be called again to begin copying the next block.

Application Examples

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. Multitasking is turned off to minimize the response latency.

```
/*
** RTALARM (p1)
**   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.
*/

#include <cdapcc.h>
void main (PIB **plib);
void rtalarm (PIPE *input_channel_pipe);

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

```
/*  Real-time processing routine.  */
void rtalarm (PIPE *in_pipe)
{
    int val;
    int  consec = 0;

    /* Multitasking is off for max speed! */
    sys_set_multitasking(eMultiOff);

    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;
    }
}
```

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

```
#reset
#idef a 1
  >set ipipe0 s0
  >time 50.0
  >buffers static
  >end
#pdef b
  >rtalarm(ipipe0);
  >end
#start a,b
```

After the DAPL interpreter performs the `START` command, the `RTALARM` command initializes, turns off multitasking, and begins continuous operation. The `eMultiOff` parameter rather than the `eMultiOffSYSIN` is used because this command is intended for continuous duty. The DAPL interpreter appears to "hang" at this point, not responding to new commands. The `DAPLINIT` program used with the `/RESET` option will reinitialize DAPL operation.

The second example is a PID controller application. The requirements for this command are the same as those for the `BPID2` application described in Chapter 9,

except that this command does not run concurrently with other tasks. Multitasking is turned off to reduce the CPU overhead, maximizing the number of high-speed channels that can be controlled simultaneously, and improving response latency.

This application satisfies all of the constraints for operating with multitasking off. Data is obtained from an input channel using buffered pipe operations, the outputs are asynchronous, the amount of data to be processed is small, and operation is continuous.

The main routine is identical to the main `BPID2` program listing shown in Chapter 9 and is not repeated here. The following partial listing shows the modified real-time update routine.

```
/*
**  RTBPID (p1, p2, n, vdac)
**    - reads control parameter data from 'p2'
**    - reads system output feedback from 'p1'
**    - controls 'n' PID loops
**    - sends control outputs to DACs specified by 'vdac'
**    - runs with multitasking OFF
*/
void   real_time_updates ( PBUF * in_buf, VECTOR * DACs,
           int blocksize)
{
    int  * samples;
    int const * outputs;
    int  channel;

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

    /* Multitasking is OFF for max speed! */
    sys_set_multitasking(eMultiOff);

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

Interrupts and Latency

When multitasking is turned off, the worst-case response latency depends only on the custom command processing time and interrupt processing time.

The following interrupts can occur when multitasking is off:

- Host Input Interrupt — occurs for each data block transferred from the ACCEL driver on the PC to a communication pipe on the Data Acquisition Processor.
- Host Output Interrupt — occurs for each data block transferred from a communications pipe on the Data Acquisition Processor to the ACCEL driver on the PC.
- Synchronous Input Interrupt — occurs periodically when DAPL updates internal data buffers after capturing a number of input samples.
- Synchronous Output Interrupt — occurs periodically when DAPL updates internal data buffers after completing output updates for a block of synchronous output data.
- RTC Interrupt — occurs periodically when DAPL updates the status of the internal real-time clock.
- DSP Interrupt — occurs each time the DSP coprocessor completes a DSP operation.
- SIO Interrupt — occurs during data transfers on the serial port, each time a received character becomes available and each time a transmitted character is sent.

Current information about interrupt response times is provided in the file `INTTIME.TXT`, which is provided on the Developer's Toolkit for DAPL diskette.

To calculate response latency for a specific application, determine the maximum number of interrupts of all kinds that could occur between the time a data sample arrives and the time the corresponding output is generated by the custom command. Combine the time required for interrupt processing and the time required for execution of the custom command code to obtain the estimate for worst-case response latency.

11. Programming Suggestions

This chapter provides some coding guidelines for writing C custom commands.

Task Parameters

A custom command usually requires one or more DAPL parameters. DAPL parameters serve the following functions:

- DAPL parameters provide information unique to each task. Since several instances of a command can be executing at once and since each task normally performs a unique computation, each task must operate on different data. A DAPL parameter list provides a means of specifying different data for different tasks.
- DAPL parameters provide a means to assign a distinct identity to each task.
- DAPL parameters provide tasks with access to DAPL data structures. When a DAPL symbol appears in a custom task's parameter list, the task receives a handle to the data structure named by the DAPL symbol. This allows the task to reference the DAPL data structure. DAPL symbols specified in a task's parameter list are the only DAPL symbols that can be referenced by a task.

The following example illustrates these uses of task parameters. The `CTEST` custom command is coded to accept two parameters, a constant and a pipe. Note that the actual value of each parameter is not known at the time the custom command is compiled:

```
void main (PIB **plib)
{
void **argv;
int argc;
int c1;
PIPE *p;
argv = param_process (plib, &argc, 2, 2, T_CONST_W,
T_PIPE_W);
c1 = *(const int *) argv[1];
p = (PIPE *) argv[2];
  .
  .
```

The custom command is compiled and downloaded to the Data Acquisition Processor. The values of the parameters are not fixed until a task definition command is entered:

```
PDEF A
  CTEST (1, P1)
  CTEST (3, P25)
END
```

This processing procedure defines two tasks. When execution of the tasks begins, the first task is executed with the value of C variable `c1` equal to `1` and the C pointer `p` pointing to the DAPL pipe `P1`. The second task is executed with the value of C variable `c1` equal to `3` and the C pointer `p` pointing to the DAPL pipe structure `P25`. The two `CTEST` task definitions provide different data to the two `CTEST` tasks.

DAPL Names and C Names

There is an important difference between names defined in DAPL and names defined in C. For example:

```
int *VAR1;
```

defines a pointer, named `VAR1`, which points to a word of storage in the C command environment. The DAPL command:

```
VARIABLE VAR1=4
```

defines a DAPL variable named `VAR1` , which references a word of storage in the DAPL system environment. Although the C code uses the same symbol name as the DAPL command, the C pointer does not automatically point to the DAPL variable `VAR1`. In order to reference DAPL variable `VAR1`, the DAPL variable must be passed as a parameter to the C custom command, and the C custom command must use the parameter handle to initialize the `VAR1` pointer:

```
VAR1 = (VAR *) argv[1];
```

After this statement is executed, dereferencing `VAR1` in the task's C environment accesses the contents of the `VAR1` variable defined in the DAPL system environment. The following C statements change the value of the DAPL variable `VAR1` from `4` to `7`:

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

Naming Task Parameters

The function **param_process** returns an array of parameter pointers. Custom commands are more understandable if each element of the parameter array is assigned a symbolic name. This is usually done in one of the following ways:

- Each element in the parameter array is assigned to an explicitly-named automatic variable of the appropriate type. If the value is not subject to change, the pointer can be dereferenced immediately and only the value retained. If the value is subject to change, the pointer can be saved and dereferenced at a later time.
- An auxiliary function can be called. Each element in the parameter array is passed to the auxiliary function. The formal parameters of the auxiliary function are given mnemonic names.
- If many functions in the C code must reference a task's parameters, it may be more efficient to assign parameter pointers or values to static variables to avoid the overhead of function calls with many parameters.

The following example uses an auxiliary function and a static variable for managing the two parameters of the `CTEST` command:

```
static void  ctest_aux (PIPE *p);
static int  taskid;
void main (PIB **plib)
{
    void **argv;
    int argc;
    PIPE *p;
    argv = param_process (plib, &argc, 2, 2, T_CONST_W,
    T_PIPE_W);
    taskid = *(CONSTANT *) argv[1];
    ctest_aux ((PIPE *) argv[2]);
}
static void ctest_aux (PIPE *p)
{
    .
    .
    .
}
```

Note the use of the `CONSTANT` type name rather than `int` while accessing the value stored in the DAPL environment. The `CONSTANT` data type includes a `const` keyword to protect the first argument of the task from accidental modification. Accidentally changing the value of a task parameter that is a constant can cause a system failure.

Debugging Custom Commands

The environment in which custom commands are executed is considerably more demanding than the environment of a typical PC application. The DAPL operating system is multitasking, with many tasks executing simultaneously. In many cases several copies of the same custom command are executed simultaneously. Furthermore, execution of a custom command often is subject to timing constraints; a custom command must process data efficiently and respond rapidly to random external events.

Traditional symbolic debugging tools such as Microsoft CodeView are not well suited to debugging custom commands. DAPL multitasking code executes hundreds of different code fragments every second; this activity is very difficult to trace with a debugger. A debugger also tends to slow down execution; and this has undesirable effects in a real time operating system and can introduce spurious timing errors.

Custom commands should be written with care. When coding errors do occur, these are best found by carefully examining custom command program logic.

System routines are not able to check their input parameter pointers for validity, so incorrect use of pointers is a common cause of custom command errors. When an incorrectly initialized pointer is passed to a DAPL system routine, the system routine may corrupt memory by modifying data referenced by the illegal pointer. For example, if an illegal pipe pointer is passed to the function **`pipe_put`**, the DAPL firmware may overwrite an internal system data structure. Later this could cause a failure that seems unrelated to the custom command.

During custom command debugging, it often is useful to send intermediate results directly to the PC using the function **`printf`**. When running a custom command from DAPview, the intermediate data values are displayed immediately on the PC's screen. Some custom commands need to respond too rapidly to allow execution of a **`printf`** function. In these cases, useful information about the status of a custom command can be sent to the 16-bit digital output port of the Data Acquisition Processor, using the **`digital_out`** routine. Alternatively, a custom command can send debugging information to an auxiliary output pipe; this information could be formatted by a separate task such as a DAPL `FORMAT` task.

Optimizing Custom Commands

The Data Acquisition Processor constantly gathers statistics about how much CPU time various tasks use. This information is available to assist the C programmer in determining whether custom tasks are using CPU resources efficiently.

The DAPL command TASKSTAT has two forms, TASKSTAT CLEAR and TASKSTAT STATUS. TASKSTAT CLEAR resets all the CPU time use statistics to zero. To get useful information about an application, this command should be issued after an application is running. TASKSTAT STATUS prints statistics about the amount of CPU time used by each task, from the time of the last TASKSTAT CLEAR command.

Typical output from the TASKSTAT command is:

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

The first four tasks are system tasks that always are defined. All remaining tasks are defined in active processing procedures.

Note that **pipe_get**, **pbuf_get**, **task_pause**, and **trigger_wait** are the common system routines that automatically release the CPU. If a C custom command seems to be using a disproportionately large amount of CPU time, check the C code to verify that the task releases the CPU when necessary. A custom task that waits for the value of a variable to change or which polls the status of a pipe or trigger using **pipe_num** or **trigger_get_immediate** should include explicit calls to **task_switch**.

Using Assembly Language in Custom Commands

If a task defined by a custom command must be executed very rapidly, portions of the custom command can be coded in assembly language. There are two methods of including assembly language into a custom command. Version 6 and above of the Microsoft C compiler and version 3.5 and above of the Borland C compiler allow assembly code to be embedded directly into C code, using the `_asm` directive. These compilers also allow the main body of a C custom command to call an assembly language routine coded for the Compact Memory Model. See your compiler manuals for information about mixed language programming.

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

- The segment registers `SS` and `DS` point to system data areas and must never 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.
- Assembly code must be relocatable. It can access the custom command data segment via `DGROUP`, but it is safer to make the function reentrant, passing all parameters, and using no references to fixed storage locations.
- Assembly code must not generate software interrupts or access BIOS data areas, since a custom command does not run under DOS.
- 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.

12. Compiling Custom Commands

This chapter provides information about how to compile custom commands. Most of the detail about the organization of the Developer's Toolkit for DAPL library files and the MS-DOS batch files is of interest only to advanced programmers who have special need for customizing their development process.

An Overview: Compiling and Running Custom Commands

After a custom command is written in C according to the instructions provided in this document, the C code must be compiled and downloaded to the Data Acquisition Processor. Let the base name for command's C source code file be the name that will be used in the DAPL file to identify the command. For example, the file `COPY2.C` provided on the Developer's Toolkit for DAPL diskette will generate a custom command with name `COPY2`.

Select the appropriate batch file from the Developer's Toolkit for DAPL directory. The choice will depend on the compiler to be used, and the operating system for which the custom command is intended. Use `BCC4.BAT` for Borland compilers or `MCC4.BAT` for Microsoft compilers when the custom command is intended to run under DAPL version 4. Use `BCC16.BAT` for Borland compilers or `MCC16.BAT` for Microsoft compilers when the custom command is intended to run under DAPL 2000. Make sure that the computer configuration information at the top of the batch file is correct for the computer system on which the Developer's Toolkit for DAPL is installed.

Change the current directory to the directory containing the source file. Then, run the batch file from the DOS command line. Possible error messages are listed in Chapter 15. Depending on the configuration information in the batch file, the COMLOAD utility will automatically download the custom command binary code to the Data Acquisition Processor. The COMLOAD utility, DAPview, or a customized PC application may also be used to download the custom command binary code as a separate operation; see Chapter 13.

The following dialog is typical of the compilation and downloading process:

```
C>MCC4 COPY2  FP
C>DV
Control / Download / One / Command
Custom command file: COPY2
Stack size: 1000
Transfer completed
```

Notice that the stack size requirement of the custom command must be specified. If the stack specified for a custom task is too small, the task may terminate with an error message. The default stack size of one thousand bytes is sufficient for most custom commands. If a custom command uses many automatic variables or nested function calls, the stack size must be increased. See the compiler manuals for an explanation of the number of bytes required by various C data types. The compiler manuals do not have information about the stack memory required in the DAPL environment, so some experimentation may be necessary. In most cases, allowing 600 bytes for DAPL system and 100 bytes for **`printf`** formatting.

Batch Files

The Developer's Toolkit for DAPL provides batch files for compiling and linking custom commands. The files `BCC4.BAT` and `BCC16.BAT` are compatible with the Borland 3.1, 4.0 and 4.5 C/C++ compilers. The files `MCC4.BAT` and `MCC16.BAT` are compatible with the Microsoft C version 5.1 and 6.0 compilers and the C/C++ version 7.0 and 8.0 (Visual C 1.0 or 1.5 Professional Edition) compilers.

The batch files require that the PC development system is configured for running the compiler and linker. In particular, the `PATH` must be established so that the compiler and linker can be executed. The compiler may also require that some DOS environment variables be established using the `SET` command. The compiler installation will usually alter the development system's `AUTOEXEC.BAT` file to take care of the environment and path. If this was not done, the compiler environment must be explicitly set up before compiling a custom command. One option is to edit a copy of the batch file to include the extra system configuration information.

The `BCC4.BAT`, `BCC16.BAT`, `MCC4.BAT`, and `MCC16.BAT` files also require information about where the Developer's Toolkit for DAPL files and utilities are located. Find the "configuration section" near the top of these files. Edit the lines that indicate the execution path for Developer's Toolkit for DAPL if the installation directory is not `C:\DTDC`. It is also helpful to have the Developer's Toolkit for DAPL directory on the execution path.

The batch file default configuration will automatically download the compiled custom command to a Data Acquisition Processor. This will not be useful if the development system used to compile the custom command is not the same as the one where the Data Acquisition Processor is installed, or if more than one Data Acquisition Processor is present. For these situations, set the option `AUTO=FALSE` in the configuration section of the batch file.

There are additional parameters that you must specify depending on your compiler type. For a Microsoft compiler, specify the optimization options, which are different for each compiler version. More details about compiler options are provided in the next section of this chapter. For a Borland compiler, specify the `BLIB` parameter, which tells the `TLINK` program where to find the Borland run-time libraries.

After the compiler and the batch file are configured, a custom command file can be compiled from any directory. First, make the directory containing the C code for the command the current directory. Then, execute the appropriate batch file, `BCC4.BAT`, `BCC16.BAT`, `MCC4.BAT` or `MCC16.BAT`. If the batch file is not on the execution path, the fully qualified path to the batch file must be specified on the command line.

Three additional parameters may be specified on the command line after the batch file name. The first is the name of the custom command file, less the `".C"` file extension. The next parameter specifies the library type to use. Use all capitals: `SMALL` or `FP`. To use the default stack size and the default `SMALL` library, omit this parameter and leave the rest of the command line blank. For custom commands with special stack usage requirements, the library type must be specified, and after that, the stack requirement in bytes is specified. A minimum stack size of 700 is recommended, or 1,000 for any custom command that uses **`printf`** and its variants for text formatting.

The Developer's Toolkit for DAPL installation establishes the `SMALL` and `FP` subdirectories to the Developer's Toolkit for DAPL directory. The `BCC4.BAT`, `BCC16.BAT`, `MCC4.BAT` and `MCC16.BAT` batch files expect this structure. It is strongly recommended that you do not alter this arrangement.

The batch file executes four steps for compiling and converting a custom command file:

- Compile the custom command source code
- Link the startup code, interface code, and custom command code
- Convert the relocatable DOS application file to a binary image
- Download the binary image

The compile step is identical to a compile-only operation for a PC application written in C.

The link step is similar to the link step for an ordinary DOS application, except that special code must be included for the custom command to run in the Data Acquisition Processor environment. The object file `CSTART4.OBJ` or `CSTART16.OBJ` contains startup code which prepares a task to begin execution and establishes the C environment expected by the compiler's code. For the `FP` library, an additional object module is included after the startup code, providing interfaces to the floating point function library and other floating point services. Following the object module is the compiled custom command object file. The custom command may make references to any number of DAPL services through Developer's Toolkit for DAPL functions, and these references are satisfied by the Developer's Toolkit for DAPL library file `DTDC4.LIB` or `DTDC16.LIB`. Finally, any function references not yet resolved are satisfied by accessing the function library provided for the compiler.

Custom commands that do not use floating point features may be compiled using any library type. For the smallest and most efficient code, the `SMALL` library should be used. If floating point is required, or to stay with one compile option that always works, use the `FP` library. Custom commands use an inline 8087 code option at compile time.

The Microsoft compilers require user configuration of the compiler runtime libraries. Use the compiler's configuration program to establish the combined library `CLIBC7.LIB` which supports:

- the Compact memory model
- native 8087 instruction set (no emulation).

The following compiler options are used for Microsoft compilers:

`/AC /Aw`

- specify the compact memory model with separate stack and data segments. Do not change these. The code pointers are small, data pointers are far, the value of the `DS` register is fixed, and the `SS` register is not equal to the `DS` register.

`/c`

- Separate compilation, required so that startup and interface code can be introduced during the link process.

/I

- Enables inclusion of header files at compile time.

`/G1`
- Specifies 16-bit code generation for 80x186 series processors; the resulting code is more efficient and more compact than 8086/8088 code, and compatible with 16-bit code running under 80x486 protected mode.

`/FPi87`
- Specifies the type of floating library support. As discussed above, the `/FPi87` option is used.

`/Ze`
- Enables language extensions.

`/W3`
- Specifies maximum level of diagnostics. This option sometimes produces numerous warning messages, even for code which is perfectly compliant with the C standard. In such cases, `W2` or `W1` might be preferred.

`/O`
- The compiler optimizations are different depending on the compiler version. In general, less aggressive optimizations are acceptable, but optimizations more aggressive than the ones recommended below may produce code which is not compatible with the Data Acquisition Processor environment, and could lead to unpredictable software failures. The `/Oi` (generate intrinsics) option is not supported. The following optimizations are recommended.

```
/Olt        C version 5
/Owelt      C version 6
/Owgelt     C/C++ version 7 and Visual C 8.0
```

The following compiler options are used for Borland compilers:

`-c`
- Separate compilation, required so that startup and interface code can be introduced during the link process.

`-mc`
- This specifies the compact memory model with separate stack and data segments. Do not change these. The code pointers are small, data pointers are far, the value of the `DS` register is fixed, and the `SS` register is not equal to the `DS` register.

`-I`
- Enables inclusion of the file with command source at compile time.

`-f87 or -f-`

- Specifies the type of floating library support. 8087 native instructions are used for the `FP` library, no floating point support for the `SMALL` library.

Code Conversion

The conversion command is:

```
EXEPROC %1 /c
```

The `EXEPROC` program is provided in the Developer's Toolkit for DAPL. This program relocates the executable file generated by the linker, producing a binary code image.

`EXEPROC` creates a file with a `.BIN` suffix which can be downloaded to a Data Acquisition Processor. `EXEPROC` is similar in function to the DOS `EXE2BIN` command, but intended for the DAPL environment rather than the DOS environment. `EXEPROC` will work both with DAPL version 4 and DAPL 2000.

Be sure to download the custom command binary code to the system for which it was compiled. A custom command compiled for DAPL version 4 will not run on DAPL 2000, and a custom command compiled for DAPL 2000 will not run under DAPL version 4.

C Restrictions

Not all compiler runtime library functions are compatible with the Data Acquisition Processor environment. Incompatible functions generally require DOS services, such as file or screen input/output. If a custom command attempts to use incompatible functions, error messages will be displayed during linking or converting. See Chapter 14 for information about compatible library functions.

In some instances, the compiler does not generate code that can be relocated to a binary image by the `EXEPROC` program.

C variables defined globally within the compile module must be preceded by the `static` keyword.

Custom command must not define a static variable that is initialized to the address of another static data structure, for example:

```
static *s = "abcd";
```

The above example generates a conversion error. In most cases, the static definition can be modified to avoid this restriction. The following code is converted without error:

```
static s[] = "abcd";
```

Different compilers have different ways of doing things. Compilers often generate very different code under different optimization options. Different maintenance releases of the same compiler may produce different code under any option. If custom command code conforms to the restrictions given in this chapter, and the code compiles cleanly but still has problems with linking or binary conversion, please call immediately for technical support.

13. PC Support

Custom commands can be sent from the PC to the Data Acquisition Processor using DAPview or COMLOAD. These programs are described in the Data Acquisition Processor manuals. Programs running in the PC also may download custom commands by reading the binary code of the commands and sending the commands to the Data Acquisition Processor. Microstar Laboratories provides Pascal and C library routines that perform this operation.

All filenames in this chapter refer to files on the Microstar Laboratories Data Acquisition Processor diskettes rather than the Developer's Toolkit for DAPL diskettes.

Downloading from C

The following routines are used for downloading custom commands to the Data Acquisition Processor. These functions are declared in the file `CDLOAD.H`.

```
int DownloadCmd ( int DapTextIO, int DapBinIO, char
    *CCFilename, unsigned int
    CCStackSize, unsigned int
    *CCLength, int BinaryFlag)
```

`DownloadCmd` downloads a custom command to the Data Acquisition Processor using device input/output. The parameter `DapTextIO` contains a DOS file handle for a text ACCEL device, and `DapBinIO` contains a DOS file handle for a binary ACCEL device. The parameter `DapBinIO` is not used if text downloading is selected. Both ACCEL devices should be numbered ACCEL devices. The parameter `CCFilename` contains the name of the DOS custom command file to be downloaded. The DAPL command is given the same name, minus any directory path or filename suffix. The stack size of the command must be specified as the `CCStackSize` parameter. The binary downloading mode should be used, except for downloading to a Data Acquisition Processor in stand-alone mode using the serial port. To specify the binary mode, set the `BinaryFlag` parameter to a nonzero value, otherwise, text mode is used.

After `DownloadCmd` returns, `CCLength` contains the number of bytes of code that were transferred. The function also returns an integer error flag:

`0`	no errors
`1`	file could not be opened
`2`	file read error
`3`	code transfer failed
`-1`	ACCEL driver not installed
`-2`	Data Acquisition Processor hardware not responding

```
int fDownloadCmd ( FILE *DapTextIn, FILE *DapTextOut, FILE
    *DapBinOut, char *CCFilename,
    unsigned int CCStackSize, unsigned
    int *CCLength, int BinaryFlag)
```

`fDownloadCmd` downloads a custom command to the Data Acquisition Processor using stream input/output. The parameter `DapTextIn` contains a C input stream pointer to a text ACCEL device, `DapTextOut` contains a C output stream pointer to a text ACCEL device, and `DapBinOut` contains a C output stream pointer to a binary

ACCEL device. The parameter DapBinOut is not used if text downloading is selected. All other parameters and return values are the same as those for DownloadCmd.

```
int DownloadList ( int DapTextIO, int DapBinIO, char
   *CCListFilename, int PrintFlag, int
   BinaryFlag)
```

DownloadList downloads a list of custom commands to the Data Acquisition Processor using device input/output. The parameter CCListFilename contains the name of the DOS file that contains the list of custom command information. See the DAPview documentation in the Data Acquisition Processor manuals for a complete description of this file's format. If the PrintFlag parameter is nonzero, DownloadList prints a message after successfully downloading each file. All other parameters and return values are the same as those for DownloadCmd.

```
int fDownloadList ( FILE *DapTextIn, FILE *DapTextOut, FILE
   *DapBinOut, char *CCListFilename,
   int PrintFlag, int BinaryFlag)
```

fDownloadList downloads a list of custom commands to the Data Acquisition Processor using stream input/output. The parameter DapTextIn contains a C input stream pointer to a text ACCEL device, DapTextOut contains a C output stream pointer to a text ACCEL device, and DapBinOut contains a C output stream pointer to a binary ACCEL device. The parameter DapBinOut is not used if text downloading is selected. All other parameters and return values are the same as those for DownloadList.

A C program that uses routines in CDLOAD must include the Microstar Laboratories C library when linking. See the Data Acquisition Processor manuals for more information about using Microstar Laboratories library routines from C.

The example program DOWNLOAD.C is a PC application which demonstrates binary downloading of the custom command file RAVE.BIN. The source file RAVE.C must be compiled into custom command format RAVE.BIN before the DOWNLOAD application can be run. The DOWNLOAD application first downloads the RAVE custom command, and then executes it using the DAPL file RAVE.DAP. The RAVE custom command calculates a running average. The DOWNLOAD program then displays the data values returned from the custom command.

Downloading from Borland Pascal

The following routines are used for downloading custom commands to the Data Acquisition Processor. These functions are in the file `CDLOAD.PAS`.

```
DownloadCmd ( DapTextIO: DosHandle; DapBinIO: DosHandle;
   CCFilename: Filepath; CCStackSize:
   Integer; var CCLength: Integer;
   BinaryFlag: boolean):Integer
```

`DownloadCmd` downloads a custom command to the Data Acquisition Processor. The parameter `DapTextIO` contains a DOS file handle for a text ACCEL device, and `DapBinIO` contains a DOS file handle for a binary ACCEL device. The parameter `DapBinIO` is not used if text downloading is selected. The parameter `CCFilename` contains the name of the DOS custom command file to be downloaded. The DAPL command is given the same name, minus any directory path or filename suffix. The stack size of the command must be specified as the `CCStackSize` parameter. The binary downloading mode should be used, except for downloading to a Data Acquisition Processor in stand-alone mode using the serial port. To specify the binary mode, set the `BinaryFlag` parameter to a nonzero value, otherwise, text mode is used.

After `DownloadCmd` returns, `CCLength` contains the number of bytes of code that were transferred. The function also returns an integer error indicator:

`0`	no errors
`1`	file could not be opened
`2`	file read error
`3`	code transfer failed
`-1`	ACCEL driver not installed
`-2`	Data Acquisition Processor hardware not responding

```
DownloadList ( DapTextIO: DosHandle; DapBinIO: DosHandle;
   CCListFilename: Filepath;
   PrintFlag: boolean; BinaryFlag:
   boolean):Integer
```

`DownloadList` downloads a list of custom commands to the Data Acquisition Processor. The parameter `CCListFilename` contains the name of the DOS file that contains the list of custom command information. See the DAPview documentation in

the Data Acquisition Processor manuals for a complete description of this file's format. If the `PrintFlag` parameter is true, `DownloadList` prints a message after successfully downloading each file. All other parameters and return values are the same as those for `DownloadCmd`.

A Pascal program that uses these routines must include the `CDLOAD` unit. See the Data Acquisition Processor manuals for more information about compiling Data Acquisition Processor applications using Borland Turbo Pascal.

The example program `DOWNLOAD.PAS` is a PC application that demonstrates binary downloading of the custom command file `RAVE.BIN`. The source file `RAVE.C` must be compiled into custom command format `RAVE.BIN` before the `DOWNLOAD` application can be run. The `DOWNLOAD` application first downloads the `RAVE` custom command, and then executes it using the DAPL file `RAVE.DAP`. The `RAVE` custom command calculates a running average. The `DOWNLOAD` program then displays the data values returned from the custom command.

14. Data Acquisition Runtime Library

This chapter describes the system routines provided by the Developer's Toolkit for DAPL. Many additional library routines are available in the Microsoft and Borland C compiler runtime libraries.

The following list summarizes all Developer's Toolkit for DAPL system routines. Some routines are available only in specific versions of DAPL. When this is true, the description of the routine will contain a section titled “Restrictions” that describes any DAPL version restrictions.

Pipe Operations

pipe_get	get a value from a pipe
pipe_get_float	get a floating point 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 value into a pipe
pipe_put_float	put a floating point 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_get	get a block of data from a pipe
pbuf_get_cnt	determine the current pipe buffer count
pbuf_get_data_ptr	get a pointer to the data array of a pipe buffer
pbuf_get_max_cnt	determine the maximum pipe buffer count
pbuf_get_min_cnt	determine the minimum pipe buffer count
pbuf_open	open a pipe buffer
pbuf_put	put a block of data into a pipe
pbuf_set_cnt	set the current pipe buffer count
pbuf_set_data_ptr	alter a pipe buffer storage pointer
pbuf_set_max_cnt	set the maximum pipe buffer count
pbuf_set_min_cnt	set the minimum pipe buffer count

Data Access

`memcpy`	copy one memory region to another memory region
`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
`var32_get`	obtain the value of a long DAPL variable
`var32_set`	assign a value to a long DAPL variable
`ralloc`	dynamically allocate bulk 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
`sys_set_multitasking`	turn multitasking on or off
`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
`fsend`	print a string
`send`	print 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_receive`	synchronize access to FFT result
`fft_request`	initiate FFT processing
`fft_status`	test for FFT completion

Digital Filters

`fir_advance`	bypass selected FIR filter computations
`fir_change`	modify FIR characteristics
`fir_init`	define a FIR filter
`fir_receive`	synchronize access to FIR result
`fir_request`	initiate FIR filter processing
`fir_status`	test for FIR completion

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_update`	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

C Compiler Runtime Routines

Many routines from the C compiler runtime library are available for use in custom commands. The compiler runtime library also contains routines that are not compatible with the Data Acquisition Processor environment. Incompatible routines generally require operating system services, such as file system, memory management, and screen displays. If a custom command attempts to use incompatible routines, error messages will be displayed during linking or compressing. A custom command should use only functions from the following categories:

- floating point math (include file `MATH.H`), except 8087 specific routines, hyperbolic and Bessel functions
- data conversion (include file `STDLIB.H`), except `ecvt`, `fcvt`, `strtod`
- string manipulation (include file `STRING.H`), except `strdup`, `strerror`
- buffer manipulation (include files `MEMORY.H` and `STRING.H`)
- variable-length argument lists (include file `VARARGS.H`)

To use floating point math functions, include the Standard C `MATH.H` file.

Most function prototypes defined in the file `CDAPCC.INC` are consistent with Standard C and with the compiler run-time libraries, but there are some exceptions. To make sure that the correct prototypes are used, the `#include` directive for the file `CDAPCC.INC` should appear after `#include` directive for all compiler header files.

The following table lists compiler runtime library routines that are compatible with both the `SMALL` and `FP` versions of the Developer's Toolkit for DAPL library.

abs	atoi	atol	bsearch	div
exit	isalnum	isalpha	isascii	iscntrl
isdigit	isgraph	islower	isprint	ispunct
isspace	isupper	isxdigit	itoa	labs
ldiv	lfind	lsearch	ltoa	memccpy
memchr	memcmp	memcpy	memicmp	memmove
memset	rand	srand	strcat	strchr
strcmp	strcpy	strcspn	stricmp	strlen
strlwr	strncat	strncmp	strncpy	strnicmp
strnset	strpbrk	strrchr	strrev	strset
strspn	strstr	strtok	strtol	strtoul
strupr	swab	toascii	tolower	toupper
ultoa	_exit	_lrotl	_lrotr	_rotl
_rotr	_tolower	_toupper		

The mathematical functions that operate on floating point data are supported by the FP version of the Developer's Toolkit for DAPL library, not by the compiler runtime library. The following table lists floating-point library routines provided with the FP version of the Developer's Toolkit for DAPL library. These functions are not available when using the SMALL version of the Developer's Toolkit for DAPL library.

acos	asin	atan	atan2	atof
ceil	cos	exp	fabs	floor
fmod	frexp	hypot	ldexp	log
log10	modf	pow	sin	sqrt
tan				

The strtod and gcvt functions work only with Microsoft compilers. The gcvt function works only with the FP versions of the Developer's Toolkit for DAPL library.

Future releases of the compiler runtime libraries may add new compatible functions, or may change some functions to make them incompatible.

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

This parameter is not used and is present only for compatibility with the Standard C library and the function prototypes defined in the compiler run-time libraries.

Return Values

There is no return value.

Description

The function **exit** causes a task to terminate. After a task calls **exit**, DAPL does not give the task any CPU time, 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 de-allocation 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
   int * real,                          // Pointer to storage
   int * imag                           // Pointer to storage
   );
```

Restrictions

This function requires DAPL 2000.

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,
   int *realbuf,                    // Pointer to storage
   int *imagbuf,                    // Pointer to storage
   unsigned long window,            // Enumeration pointer
   int direction,                   // Enumeration
   int solver,                      // Enumeration
   int post,                        // Enumeration
   int options                      // Bit mask
   );
```

Restrictions

This function requires DAPL 2000.

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 7 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
   int *realbuf,                 // Pointer to storage
   int *imagbuf,                 // Pointer to storage
   int post,
   int options
   );
```

Restrictions

This function requires DAPL 2000.

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 is not replaced by output data.

When the `FFTPOST_DEFER` option is selected by the **`fft_init`** function, the call to **`fft_postop`** serves in place of the **`fft_receive`** function to assure that output results are available to the custom 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 7 for more information about the various configuration options.

See Also

`fft_init`, **`fft_request`**, **`fft_status`**, **`fft_receive`**

fft_receive

Synchronize access to an FFT result.

```
void  fft_receive (
   FFTB * fft                          // FFT control block handle
   );
```

Restrictions

This function requires DAPL 2000.

Parameters

fft

Pointer variable containing a handle for the FFT control block to be used.

Return Values

There is no return value.

Description

Return from function **fft_receive** guarantees that results of an FFT operation, initiated by an **fft_request** function, are available in the storage areas specified by the **fft_init** function. If the operation is not complete, this function suspends task execution until the operation is complete. Use the **fft_status** function to check for a completed operation without blocking the execution of the task.

When the FFTPOST_DEFER option is selected by the **fft_init** function, the call to **fft_receive** may be omitted.

See Also

fft_init, **fft_status**, **fft_receive**, **fft_request**

fft_request

Initiate FFT processing.

```
void  fft_request (
   FFTB * fft                          // FFT control block handle
   );
```

Restrictions

This function requires DAPL 2000.

Parameters

fft

Pointer variable containing a handle for the FFT control block to be used.

Return Values

There is no return value.

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

fft_status

Test for FFT completion.

```
int  fft_status (
   FFTB * fft                          // FFT control block handle
   );
```

Restrictions

This function requires DAPL 2000.

Parameters

fft

Pointer variable containing a handle for the FFT control block to be used.

Return Values

The function returns a nonzero value when the computations are completed, and a zero value when computations are not yet completed.

Description

The function **fft_status** reports whether an FFT computation initiated by the **fft_request** function has completed. It is used to avoid blocking task execution. If blocking task execution is useful, the **fft_receive** function can be called directly, and it will wait for the transform results to become available.

See Also

fft_request, **fft_receive**

fir_advance

Bypass selected FIR filter computations.

```
int fir_advance (
   FIRB *fir,                          // FIR filter control block handle
   int count
   );
```

Restrictions

The function requires DAPL 2000.

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
   int * coeffs,                       // Pointer to coefficient array
   int length,
   int scale,
   int decimate
   );
```

Restrictions

The function requires DAPL 2000.

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 (
   int * coeffs,                          // Pointer to coefficient array
   int length,
   int scale,
   int decimate
   );
```

Restrictions

The function requires DAPL 2000.

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 7 for more information about the meaning and application of the various configuration options.

See Also

fir_change, **fir_request**

fir_receive

Synchronize access to FIR result.

```
void  fir_receive (
   FIRB * fir                          // FIR filter control block handle
   );
```

Restrictions

The function requires DAPL 2000.

Parameters

fir

Pointer variable containing a handle for the FIR filter control block to be accessed.

Return Values

There is no return value.

Description

Completion and return from function **fir_receive** guarantees that results of a digital filtering operation are available in the data array specified to the **fir_request** function. The return value of the **fir_request** or **fir_status** function indicates the number of items placed into the array. The data array must not be used for any other purpose between the calls to the **fir_request** and **fir_receive** function.

See Also

fir_init, **fir_request**, **fir_status**, **fir_receive**

fir_request

Initiate FIR filter processing.

```
int  fir_request (
   FIRB * fir,                    // FIR filter control block handle
   int * data,                    // Data to be filtered
   int count
   );
```

Restrictions

The function requires DAPL 2000.

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. If the particular model of Data Acquisition Processor running this command does not have a separate DSP processor, 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. Using the **fft_status** function to determine the number of results, rather than depending on the return value from the **fir_request** function, works with any Data Acquisition Processor model.

See Also

fft_init, **fft_status**, **fft_receive**

fir_status

Test for completion of FIR filter processing.

```
int  fir_status (
   FIRB * fir                          // FIR filter control block handle
   );
```

Restrictions

The function requires DAPL 2000.

Parameters

fir

Pointer variable containing a handle for the FIR filter control block to be examined.

Return Values

If filter computations are in progress but not completed, the returned code is -1. If the amount of data provided to the filter was not sufficient to fill the internal filter shift register, and computations could not proceed, a 0 is returned. If a positive value is returned, the filter operation is complete and this number of results was generated. Result values replace the original data in the *data* array provided to the **fir_request** function.

Description

The function **fir_status** reports whether a digital filtering operation, initiated by a call to the **fir_request** function for the specified filter *fir*, is completed.

This function works with any Data Acquisition Processor model.

See Also

fir_init, **fir_request**, **fir_receive**

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 double conversions and data types. Floating point types and conversions are available when using the FP version of the Developer's Toolkit for DAPL library.

Note: To keep task stack requirements to a minimum, there is a limit on the length of the final formatted string. For DAPL 2000 the limit is 132 characters, and for DAPL version 4 it is 100. Be particularly careful not to format a very large floating point number using the %f format conversion code.

fsend

Print a string.

```
void fsend (
   PIPE *output,                    // Pipe handle
   char *str                        // Pointer to a string of characters
   );
```

Parameters

output

Pointer variable containing a handle for the pipe to be examined.

str

The string of characters to send to the byte output pipe.

Return Values

There is no return value. The first character of *str* is set to `'\0'` before **fsend** returns; this sets *str* to the empty string.

Description

The function **fsend** sends *str* to the byte output pipe *output* without formatting.

See Also

send

icosine

Return the integer cosine of an integer value.

```
int icosine (
    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
   );
```

Restrictions

This function requires DAPL 2000.

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 `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 integer values can be requested at task initialization time using the **`ralloc`** function:

```
int *waveptr;
waveptr = ralloc((500+1)*sizeof(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
   );
```

Restrictions

This function requires DAPL 2000.

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 (
    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
   );
```

Restrictions

This function requires DAPL 2000.

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`.

memcpy

Copy one memory region to another memory region.

```
char *memcpy (
   char *dest,                          // Pointer to data storage array
   char *src,                           // Pointer to data storage array
   unsigned int count
   );
```

Parameters

dest

A memory region where copied bytes are written.

src

A memory region where the original data are found.

count

Number of bytes to copy.

Return Values

The function returns the value of pointer *dest*.

Description

The function **memcpy** copies *count* bytes from *src* to *dest*. The two memory regions must not overlap.

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 the following error message and then calls **exit**:

```
<name>: 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 ParamError 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_VaueInconsistent	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 format and then calls **exit** to terminate the task:

```
Error 1236: <cmdname> - parameter <ip> - <descriptive text>
```

Values for *pecode* are defined in the file PARAMS.H. Specify *pecode* by name. The DAPL operating system supplies the cmdname, and also provides the descriptive text based on the value of the parameter *pecode*.

If the DAPL ERRORQ option is on, the error message is suppressed and ERRORQ is set to a nonzero value.

The function **param_error_msg** should be used rather than function **param_error** when more diagnostic information is necessary to identify the error.

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 file `CDAPCC.H` defines a number of type names that are allowed in the **`param_process`** parameter list:

`T_PIPE_B`	byte pipe
`T_PIPE_W`	word pipe
`T_PIPE_L`	long pipe
`T_PIPE_FL`	float pipe
`T_TRIGGER`	trigger
`T_VAR_W`	word variable
`T_VAR_L`	long variable
`T_CONST_W`	word constant
`T_CONST_L`	long constant
`T_RFLAG`	region flag
`T_STR`	string
`T_VECTOR_W`	word vector
`T_VECTOR_L`	long vector (DAPL 2000 only)

If a task allows several types for a parameter, the C bitwise “or” operation can be used to combine the type names.

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.

```
int 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 then distinguish between these two types.

See Also

param_process

pbuf_get

Get a block of data from a pipe.

```
void 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

There is no return value.

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 contains a field that points to the pipe from which data will be read. This field is initialized 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 using function **pbuf_get_cnt**.

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 using 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 maximum size of the data array must not exceed 65520 bytes.

The function **pbuf_open** also initializes three internal fields:

```
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 of the internal fields 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 initialize the internal fields.

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. On exit, **pbuf_put** sets the pbuf_cnt field to zero.

If **pbuf_put** cannot add the required number of values to the pipe because the maximum size of the pipe has been reached, the calling 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 DAPL WAIT /NOWAIT parameter when the pipe is defined using the DAPL command PIPES.

See Also

pbuf_open

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 area to a pipe buffer handle.

```
int *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 area 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 a common buffer 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. Usually, 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.

```
unsigned int pbuf_set_max_cnt (
   PBUF  *pb,                          // Pipe buffer handle
   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.

```
unsigned int pbuf_set_min_cnt (
   PBUF  *pb,                          // Pipe buffer handle
   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_open

Open and initialize a PID control block.

```
PID *pid_open (
   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_update** functions are called.

See Also

pid_update, **pid_set_setpoint**, **pid_tune**

pid_preset

Establish a pre-determined PID operating state.

```
int pid_preset (
   PID  *pid,                          // PID control block handle
   int sysval,
   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_update** 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_update**

pid_set_setpoint

Assign a PID setpoint.

```
void pid_set_setpoint (
   PID *pid,                          // PID control block handle
   int val
   );
```

Parameters

pid

Pointer variable containing a handle for the PID control block to be examined.

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 = input - <setpoint>
 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 error term `e` and a positive coefficient produce a positive computed output. For most

controlled systems, this will tend to increase, not decrease, the error `e`. These systems should use the negative of the value returned by **`pid_update`** as the control output.

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_update`**

pid_update

Compute new PID state and output.

```
int pid_update (pid, val)
   PID *pid,                          // PID control block handle
   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_update** 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_update** function returns the value of the PID control output.

Note: For many controlled systems, the negative of the value returned by **pid_update** 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_update** function is called.

See Also

pid_tune

pipe_get

Get a 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.

See Also

pipe_num, **pipe_num_complete**

pipe_get_float

Get a floating point value from a pipe.

```
float pipe_get_float (
   PIPE  *input                          // Pipe handle
   );
```

Parameters

input

Pointer variable containing a handle for the pipe to be examined.

Return Values

The function returns one floating point value from a pipe.

Description

The function **pipe_get_float** reads one floating point value from a pipe. If the pipe is empty when **pipe_get_float** 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 to determine whether the pipe contains data.

See Also

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 performance 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
   );
```

Restrictions

The function **pipe_num_complete** is available only in DAPL versions 4.21 and higher.

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 parameter when the pipe is defined in the DAPL configuration.

pipe_put_float

Put a floating point value into a pipe.

```
void pipe_put (
   PIPE  *pipe,                          // Pipe handle
   float  pval
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to receive the value.

fval

Floating pointer data value to be added to a pipe.

Return Values

There is no return value.

Description

The function **pipe_put_float** places floating point data value *fval* into 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 parameter when the pipe is defined in the DAPL 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_width

Return the width of a pipe in bytes.

```
int pipe_width (
   PIPE *pipe                          // Pipe handle
   );
```

Parameters

pipe

Pointer variable containing a handle for the pipe to be examined.

Return Values

The function returns the width of a pipe in bytes.

Description

The function **pipe_width** returns the width of a pipe in bytes. The width is one for a byte pipe, two for a word pipe, and four for a long pipe or a float 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 varying number of optional parameters appearing after the mandatory 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 is identical to 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 long double conversions and types. Floating point types and conversions require the FP version of the Developer's Toolkit for DAPL library.

To keep task stack requirements to a minimum, there is a limit on the length of the final formatted string. For DAPL 2000 the limit is 132 characters, and for DAPL version 4 it is 100. Be particularly careful not to format a very large floating point number using the %f format conversion code.

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. 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.

send

Send a message text to the default text output pipe.

```
void send (
   char *str                                  // Pointer to a character string
   );
```

Parameters

str

Pointer to a character string

Return Values

There is no return value. The first character of *str* is set to `'\0'` before **send** returns; this sets *str* to the empty string.

Description

The function **send** sends *str* to the output pipe `$SYSOUT`.

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 double conversions and types. Floating point types and conversions regulate the FP version of the Developer's Toolkit for DAPL library.

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 double conversion codes and long double pointer types are not supported. Floating point types and conversions require the FP version of the Developer's Toolkit for DAPL library.

Note: This function is dangerous, as are all implementations conforming to the C Language standard. Be very careful that the data types correspond exactly to the types implied by the conversion codes in the format string. Also verify that each conversion code has a corresponding pointer in the varying portion of the parameter list.

sys_exec_command

Send a DAPL command to the DAPL system command interpreter.

```
void sys_exec_command (
   char * command                          // Pointer to a character string
   );
```

Restrictions

For DAPL version 4, this function is only available with release 4.10 and higher.

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 far pointer. The return value should be cast to the appropriate data type. 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_OPTIMIZE	int
GI_FLOAT_ERROR	int
GI_ROUNDING	int
GI_AINEXPAND	int
GI_IN_ACTIVE	int
GI_OUT_ACTIVE	int
GI_IN_CNT	unsigned long int
GI_OUT_CNT	unsigned long 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 long int
GI_TMEMSIZE	unsigned long 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 GI_OPTIMIZE request code is used only in DAPL version 4. It is replaced by GI_BUFFERING in DAPL 2000. The GI_FLOAT_ERROR, GI_ROUNDING and GI_INEXPAND codes are available in DAPL 2000 and in DAPL version 4.3 or higher. Task scheduling request codes GI_SCHEDULE_MODE and GI_QUANTUM are available only in DAPL 2000. The GI_FFTSIZE, GI_IBURST_ACTIVE, GI_OBURST_ACTIVE codes are available only in DAPL 2000.

The following request codes return the value of the corresponding DAPL options: 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. These return a nonzero value if ON, a zero value if OFF.

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 currently 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 built-in DAPL processing commands, and 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. GI_HMEMAVL, GI_HMEMSIZE, GI_TMEMAVL and GI_TMEMSIZE are available only on DAPL version 4.2 or higher.

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. GI_OEM_ID is available only on DAPL version 4.2 or higher.

The GI_FFTSIZE code is available only with DAPL 2000. It 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.

DAPL 2000 provides the GI_SCHEDULE_MODE and GI_QUANTUM options. A GI_QUANTUM request returns a number indicating the length of the task scheduling quantum in microseconds. The GI_SCHEDULE_MODE request returns one of the codes eSchedFixed or eSchedAdaptive. The special codes are defined in the file CDAPCC.H.

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.

Note: DAP 2400 applications should count acquisition samples generated by input procedure sampling rather than using this function.

sys_get_version

Return the software and hardware version numbers of the Data Acquisition Processor.

```
void sys_get_version (
   int *software,                    // Pointer to integer
   int *hardware,                    // Pointer to integer
   int *rev                          // Pointer to integer
   );
```

Parameters

software

Pointer to an integer for storing the software version of DAPL.

hardware

Pointer to an integer for storing a code indicating the type of Data Acquisition Processor.

rev

Pointer to an integer for storing a code indicating the hardware revision level for the Data Acquisition Processor

Return Values

Return values are placed into integer variables specified by the three pointer parameters, *software*, *hardware*, and *rev*. The number returned in the software variable represents the software version of DAPL. A value of 400 represents DAPL 4.00, etc. Symbols for the Data Acquisition Processor hardware types are provided in the CDAPCC.H header file. Hardware revisions are numbered sequentially, 1, 2, etc.

Description

The function returns the software and hardware version numbers of the Data Acquisition Processor.

sys_set_multitasking

Turn multitasking on or off.

```
void sys_set_multitasking(
   int  mode
   );
```

Parameters

mode

A value specifying the multitasking mode.

Return Values

There is no return value.

Description

The function **sys_set_multitasking** gives the custom command direct control of DAPL multitasking operation.

eMultiOn	enables multitasking operation
eMultiOff	disables multitasking operation
eMultiOffSYSIN	disables multitasking operation until DAPL receives another direct command

Termination of the custom command task automatically restarts normal DAPL multitasking operation.

This command should only be used by specialized custom command applications requiring a minimum real-time response latency.

Note: Special input procedures configurations are necessary when using this function. See Chapter 10 for complete information on the advantages and hazards of using this function.

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 the multitasking in the Data Acquisition Processor, there can be several milliseconds of additional delay before a task continues after a call to **task_pause**.

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 is typically used to improve real-time response. 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.

The use of this function is demonstrated in the TSTAMP2.C custom command example.

See Also

trigger_get_immediate, **trigger_wait**, **trigger_num**

trigger_get_immediate

Return the next available assertion or task 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

A pointer to 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 contents of the integer variable indicated by pointer *flag* are set:

- *flag* is zero (logical false) if no assertion is present in the trigger structure and a status report is returned,
- *flag* is nonzero (logical true) if an assertion is returned.

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 indicated by pointer *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.

Use of this function is demonstrated in the TSTAMP2.C and HOLDOFF.C custom command examples.

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 one of the following integer codes indicating the operating mode of trigger *trig*.

```
TRIG_NATIVE_MODE
TRIG_MANUAL_MODE
TRIG_AUTO_MODE
TRIG_NORMAL_MODE
TRIG_DEFERRED_MODE
```

These codes are defined by the CDAPCC.H file.

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.

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 current value of a specified trigger property *prop* for trigger *trig*.

Description

The function **trigger_get_property** returns a trigger's property value. The property `prop` is an integer number from the following list, defined in the `CDAPCC.H` file.

```
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 custom 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.

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

A value representing the sample number.

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.

The use of this function is demonstrated in the WATCHDOG.C custom command example.

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 value representing the sample number.

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.C 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,
   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_get** or **pbuf_get** to fetch the data associated with the signaled event.

See Also

pipe_get, **pbuf_get**

var32_get

Return the value of a long DAPL variable.

```
long int var32_get (
   LVAR  *var                          // Pointer to long variable
   );
```

Parameters

var

Pointer variable containing a handle for the variable to be examined.

Return Values

The function returns the 32-bit value of a DAPL long variable.

Description

The function **var32_get** is required only for advanced applications that use DAPL long variables for communication between several tasks.

This function is equivalent to the C operator '*v.' The high order and low order 16-bit words of *var* are fetched in an atomic manner that avoids task preemption during the fetch operation.

See Also

var32_set

var32_set

Assign a value to a long DAPL variable.

```
long int var32_set (
   LVAR *var,                          // Pointer to long variable
   long int value
   );
```

Parameters

var

Pointer variable containing a handle for the variable to be examined.

value

32-bit value to be assigned to a DAPL long variable

Return Values

The function returns the new contents of the variable *var*.

Description

The function **var32_set** is required only for advanced applications that use DAPL long variables for communication between several tasks.

The function **var32_set** assigns a 32-bit value to a DAPL long variable. This function is equivalent to the C statement:

```
*v = value;
```

The high order and low order 16-bit words of *var* are assigned in an atomic manner that avoids task preemption during the assignment operation.

See Also

var32_get

vector_length

Determine the length of a DAPL vector.

```
unsigned 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 DAPL vector.

```
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 DAPL vector. 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.

```
unsigned 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 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**

15. Error Messages

This chapter contains a list of error messages that may be generated during compilation, linking, conversion, downloading, or execution of custom commands. The compilation, linking, and conversion steps are controlled by the batch files `MCC4.BAT`, `BCC4.BAT`, `MCC16.BAT`, or `BCC16.BAT`. Downloading is performed by DAPview, COMLOAD, or a user-supplied program. The Data Acquisition Processor begins execution of a task when a `START` command is issued.

This is not a comprehensive listing of all possible error message that you might see when developing a new custom command. These are error conditions that indicate problems special to the DAPL or DAPL 2000 operating environment.

Compilation Messages

```
Warning: Address of frame variable taken
    DS != SS
```

This message is issued by the C compiler if the address of a local variable is passed to a function which requires a near pointer. This will result in incorrect execution since the compiler makes incorrect segment register assumptions. The solution is to use a parameter list pointer which is "far." Since the compilation batch files specify the compact memory model (with far data pointers), conversion to a valid far pointer is automatic, provided that function prototypes are used.

Linking Messages

```
Unresolved
externals: xxxxxx in file yyyyyy
```

The linker issues this message if it cannot find an external routine. If the external routine is a Microstar Laboratories system routine or compiler runtime library routine, verify that the spelling of the routine name is correct, including underscores and capitalization. If the function is not listed in Chapter 14, it is not available to a custom command.

```
Symbol defined more that once: xxxxxx
```

Not all compiler runtime library routines are compatible with the Data Acquisition Processor environment. Incompatible routines generally require DOS services, such as

file or screen input/output. If a custom command attempts to use incompatible routines, the linker tries to include DOS-only versions of several Developer's Toolkit for DAPL system routines. This results in duplicate symbol errors. See Chapter 14 for information about compatible library routines.

This message can also occur when floating point operations are used without specifying the `FP` library on the batch file command lines when compiling.

```
Undefined symbol
N_FTOL@
```

This error message can occur if floating point operations are used without specifying the `FP` library on the batch file command line when compiling.

```
Error: '_fpsp': unresolved external
```

A custom was command compiled with the alternate math library provided with the compiler. The 8087 inline math library is required. This error message will normally not be seen unless the batch file for compiling custom commands has been altered in such a way that the wrong library type is used.

```
Error: '__CIxxxxx': unresolved external
```

The use of the `Oi` compiler option to enable code generation of floating point intrinsics is not supported.

Conversion Messages

```
Could not convert - relocation required
```

C variables which are defined globally must be preceded by the `static` keyword. In addition, custom commands require special relocation which places some constraints on the use of initialized static variables. See Chapter 12.

Downloading Messages

```
Error: command transfer failed
Error: could not open command list file
Error: syntax error in command list file
```

These messages are issued by DAPview or COMLOAD if the custom command(s) could not be downloaded to the Data Acquisition Processor. See the Error Message chapter of the Systems Manual for more information about downloading errors.

```
<task>: out of
heap memory
```

Both DAPL version 4 and DAPL 2000 must allocate space for custom command tasks from a memory segment limited to 64 Kbytes If there are too many tasks which are defined by custom commands, or if each instance uses too much stack memory, this error can occur. For DAPL version 4, static data memory is also allocated from the same 64 Kbyte segment, which further limits the number of custom command tasks which can run at one time.

```
<task>: too many
parameters
<task>: too few parameters
<task>: 'xxxx' should not be a 'yyyy'
```

These messages are issued by the function **param_process** if a parameter list incompatibility is detected.

```
<task>: stack overflow
```

This message is issued if a task's stack usage exceeds its stack allocation at any time while the custom command is running.

```
Division by zero error
```

A task attempted to perform integer division by zero.

```
Error: Illegal library function invocation
```

A task invoked an unsupported Microstar Laboratories library function. This error can occur when using a very old custom command binary file with DAPL version 4. The old custom command code uses hardware-specific system functions which are no longer supported.

16. Appendix A. Compatibility with Previous Versions

The Developer's Toolkit for DAPL was previously named the Advanced Development Toolkit. This appendix explains compatibility issues between the Advanced Development Toolkit versions 1-3 and the Developer's Toolkit for DAPL version 4.0.

In this chapter and throughout this document, the Advanced Development Toolkit is referenced as a “previous version of the Developer's Toolkit for DAPL.”

Binary Code Compatibility

The Developer's Toolkit for DAPL versions 1 through 3 and the DAPL operating system versions 3 and 4 maintained a very high degree of backward source code and binary code compatibility. With very few exceptions, binary code modules generated with any Developer's Toolkit for DAPL release ran correctly on any Data Acquisition Processor model using any DAPL operating system release. Compatibility among hardware and software versions depended to a great degree on binary code compatibility in the Intel 80x86 processor family.

This compatibility has a price. The price becomes larger as processors become more powerful. New 32-bit processor and system features are not always compatible with old binary code intended for 8-bit and 16-bit systems. Newer processors are optimized for a large 32-bit address space, but for full backward compatibility, it is necessary to use the older and less efficient “real mode” scheme with multiple, overlapping 64K, 16-bit address spaces. For example, some functions provide access to elements of the DAPL system data through pointers. This works well if the pointer uses an offset that is representable in 16 bits. If the offset is larger, such a pointer is not representable and therefore not usable by a 16-bit custom command.

Another subtle problem is alignment. There is a surprising loss of speed when 32-bit processors operate upon data that is not aligned to 32-bit data boundaries. Some of the older Developer's Toolkit for DAPL implementations force other data alignments, causing significant impacts on all parts of DAPL system operation.

Taken individually, the problems are small, but taken all together, the constraints are very significant. For this reason, starting with release 4.0 of Developer's Toolkit for DAPL, backward binary code compatibility is not maintained. Commands compiled for DAPL version 4 will not run under DAPL 2000, and commands compiled for

DAPL 2000 will not run under DAPL 4. For the most part, however, custom command source code is compatible. An existing custom command can be recompiled using the Developer's Toolkit for DAPL, with few or no source code changes, to run on either system.

Source Code Compatibility

Many functions supported in previous version of the Developer's Toolkit for DAPL have new names. Most of renaming is strictly for purposes of consistent notation. The `CDAPCC.H` file automatically includes a backward compatibility file `CDAPBACK.H`, which maps older names used in previous Developer's Toolkit for DAPL versions into the new names. If only the current notations are used, as described in the rest of this manual, there is no need for the backward compatibility notations, and the line `#include "CDAPBACK.H"` can be removed from the `CDAPCC.H` file.

Custom command source code using certain special features must be modified to be compatible with the Developer's Toolkit for DAPL.

Direct addressing of `struct_pbuf` is no longer supported by DAPL 2000. Access to `PBUF` data and control fields must be through the pipe buffer access functions **`pbuf_get_cnt`**, **`puf_set_cnt`**, **`puf_get_max_cnt`**, **`pbuf_set_max_cnt`**, **`pbuf_get_min_cnt`**, **`puf_set_min_cnt`**, **`pbuf_get_data_ptr`** and **`pbuf_set_data_ptr`**. These functions should be used as described in Chapter 4. The `pbuf_cnt`, `pbuf_min_cnt`, `pbuf_max_cnt`, and `pbuf_ptr` macros used in earlier releases of the Developer's Toolkit for DAPL now expand as references to the new functions, and will work under DAPL 2000, with one exception. Assignment may not be made to the data pointer field by direct assignment to the `pbuf_ptr` macro, and this syntax must be replaced by **`pbuf_set_data_ptr`**. The macro expansions are not as efficient as using the new functions directly, but the application will work correctly if the old macros compile successfully.

For example, with previous versions of the Developer's Toolkit for DAPL, it was common practice to assign a data buffer storage address using the following syntax:

```
pbuf_ptr( inbuf )  = user_pointer;        /* ERROR */
pbuf_ptr( outbuf ) = pbuf_ptr( inbuf );   /* ERROR */
```

The expressions on the left hand side are no longer valid for DAPL 2000, and will generate a compile-time error. The **`pbuf_set_data_ptr`** function must be used instead, as in the following syntax:

```
pbuf_set_data_ptr( inbuf, user_pointer );
pbuf_set_data_ptr ( outbuf, pbuf_get_data_ptr ( inbuf ));
```

The following minor differences can also affect backward source code compatibility:

1. The new Toolkit provides an explicit prototype for the `main` function. The compiler may diagnose a conflicting, explicitly-declared prototype for `main`, or a main function that does not have conforming type or parameters.

2. The functions **`dsp_done`**, **`dsp_alloc`**, **`dsp_request_init`**, **`dsp_receive_result`**, **`dsp_send_request,`** and various supporting macros used with these functions are now obsolete for all products except the DAP 2400a and DAP 2416a. These should not be used for any new development. The old functions deliver only a fraction of the FFT and filtering capabilities provided by supported functions described in Chapter 7. The obsolete functions are automatically included with the file `CDAPBACK.H`.

3. The `typedef struct _vector` is no longer supported. Access to vector data must use the `vector` data access functions **`vector_start`** and **`vector_length`**.

4. The **`param_type`** function returns a type `unsigned long` rather than type `int`. This may cause the compiler to warn of a fixed-point type conversion if the value is stored in an intermediate variable. The diagnostic can be eliminated by storing the return value in an `unsigned long` variable rather than an `int`.

5. Obsolete functions `wait` and `trigger_count` are no longer supported. These functions became obsolete in earlier releases of the Developer's Toolkit for DAPL.

6. The `EXTENDED` library version is not supported in this version of the Toolkit. Commands needing floating point services should use the `FP` library instead. The function `gcvt,` which was supported by the `EXTENDED` library, is no longer supported directly by the Developer's Toolkit for DAPL. Full formatting capabilities for `float` and `double` types are provided by the **`printf`** function.

17. Appendix B: DAP 2400a/DAP 2416a DSP Support

The Developer's Toolkit for DAPL allows a DAP 2400a or DAP 2416e custom command to send blocks of data directly to the onboard Motorola DSP 56001 digital signal processor. The DSP 56001 performs fast Fourier transform processing and returns processed data. Since many digital signal processing algorithms are based on fast Fourier transforms, a DAP 2400a or DAP 2416a custom command gains powerful signal processing capabilities. Furthermore, a custom command can schedule DSP processing so that the 80186 processor and the DSP 56001 operate simultaneously.

A task communicates with the DSP 56001 by creating a DSP request structure. This structure contains information about the type of DSP operation and the locations of the DSP parameters. Many tasks can submit simultaneous DSP requests to the DAPL operating system. A single task may even submit several DSP requests simultaneously. DAPL schedules DSP requests on a first-come first-served basis, and notifies each task when one of its DSP operations has been completed.

The system interface file `CDAPCC.H` defines the data structures and routines required to communicate with the DSP 56001.

A DSP request structure is created by including the file `CDAPCC.H` in a custom command, and then defining and initializing the following pointer:

```
DSP_REQ *dsp_ptr;
dsp_ptr = dsp_alloc ();
```

The `dsp_ptr` variable is used to access fields of the DSP request structure. The function **`dsp_request_init`** initializes most fields of the DSP request structure.

```
rc = dsp_request_init (dsp_ptr, dsp_op, n, op_params,
                       data1, data2);
```

A nonzero return code from **`dsp_request_init`** indicates successful parameter initialization.

The parameter dsp_op selects the DSP processing operation; this always is set to the predefined constant FFT_OP . The parameter n is the size of the Fourier transform. This parameter must be a power of two. The allowed sizes are shown in the following table:

	Transform Size	
DAP Model:	**Minimum**	**Maximum**
DAP 2400a/4 and DAP 2416a/4	4	512
DAP 2400a/5	4	2048
DAP 2400a/6 and DAP 2416a/6	4	8192

The parameter op_params selects the type of fast Fourier transform and input/output options. This parameter should be the bitwise “or” of one predefined constant from each of the following columns:

FFT_FORWARD	FFT_RECTANGULAR	FFT_COMPONENTS
FFT_REVERSE	FFT_HANNING	FFT_AMPLITUDE
	FFT_HAMMING	FFT_POWER

Constants in the first column select a forward or reverse transform. Constants in the second column select the type of input window (rectangular, Hanning, or Hamming). Constants in the third column select the type of transform output (complex numbers, amplitude, or power). The last two output options are defined for forward transforms only. A typical value for the parameter op_params is

```
FFT_FORWARD | FFT_HANNING | FFT_COMPONENTS.
```

The final two parameters to **dsp_request_init** specify two C arrays which store the input/output data for the FFT operation. Each array must contain at least n 16-bit locations. Before performing an FFT operation, the real components of the input data are placed in data1 and the imaginary components of the input data are placed in data2 . A transform of real-only data should fill data2 with zero data. The contents of data1 and data2 are over-written during the transform operation.

After the transform operation is completed, both data1 and data2 arrays contain output data. If the FFT output parameter is FFT_COMPONENTS, data1 and data2 contain the real and imaginary output components, respectively. An FFT of size n generates n real and n imaginary output values. If the FFT output parameter is FFT_AMPLITUDE , the first n/2 values of data1 contain 16-bit amplitude data. If the FFT output parameter is FFT_POWER , 32-bit power values are returned. The first n/2 components of data1 contain the least significant 16-bits of each power value and the first n/2 components of data2 contain the most significant 16-bits of each power value.

After initializing a DSP request structure, a task places input data into its two data arrays. Then, a request is sent to the DSP 56001 by calling the function **dsp_send_request**. After sending a request, a task waits until the request is completed. The function **dsp_done** returns a nonzero value when the request is finished. A task may choose to pause while a request is processed, or may perform other activities while the DSP 56001 is busy. Once a request has been completed, **dsp_receive_result** returns the result of the FFT processing in the data arrays. Typical C code for processing a block of data is:

```
dsp_send_request (dsp_ptr);
while (!dsp_done (dsp_ptr))
    task_switch();
dsp_receive_result (dsp_ptr);
```

Once a request has been finished, a new request can be initiated. New data values are placed in the data arrays and **dsp_send_request** is called again. The function **dsp_request_init** should be called only once, unless different FFT parameters are desired.

FFT Programming Examples

The following code illustrates a complete example using a fast Fourier transform from a custom command. This custom command accepts three DAPL parameters: an input pipe, the size of the fast Fourier transform, and an output pipe.

```
/* FFT2 (p1, n, p2)
* - compute 'n' point FFT transforms on the
* data in pipe 'p1' and transfer amplitude
* output data to pipe 'p2'
*/

#include <cdapcc.h>
void main (PIB **plib)
{
    void **argv;
    int argc;
    PIPE *in_pipe, *out_pipe;
    int n, I;
    PBUF *inbuf, *outbuf;
    int *y_array;
    DSP_REQ *dsp_ptr;
    argv = param_process (plib, &argc, 3, 3, T_PIPE_W,
                                    T_CONST_W, T_PIPE_W);

    /* open pipes and initialize parameters */
    in_pipe = (PIPE *) argv[1]; open_pipe (in_pipe, P_READ);
    n = *(const int *) argv[2];
    out_pipe = (PIPE *) argv[3]; open_pipe (out_pipe,
P_WRITE);

    /* allocate input and output pipe buffers */
    inbuf = open_pbuf (in_pipe, n);
    outbuf = open_pbuf (out_pipe, n/2);
    pbuf_min_cnt(inbuf) = n;
    pbuf_max_cnt(inbuf) = n;
```

```
    /* allocate a second DSP parameter array */
    y_array = (int *) ralloc (2*n);

    /* allocate and initialize the DSP request structure */
    dsp_ptr = dsp_alloc ();
    if (!dsp_request_init (dsp_ptr, FFT_OP, n,
            FFT_FORWARD | FFT_AMPLITUDE | FFT_HANNING,
            pbuf_ptr(inbuf), y_array))
        param_error();

    while (1) {
            /* read the next block of input data */
            get_bpipe (inbuf);

            /* zero the imaginary components of the input */
            for (i=0; i<n; I++)
            y_array[i] = 0;

            /* send data to the DSP and wait */
            dsp_send_request (dsp_ptr);
            while (!dsp_done (dsp_ptr))
            task_switch();
            dsp_receive_result (dsp_ptr);

            /* copy n/2 words of amplitude data to the */
            /* output buffer */
            memcpy ( (char) pbuf_ptr(outbuf), (char)
pbuf_ptr(inbuf), n);

            /* send the amplitude data to the output pipe */
            pbuf_cnt(outbuf) = n/2;
            put_bpipe (outbuf);
    }
}
```

Fast Fourier transform is the basis for many powerful signal processing algorithms. The cepstrum of an input signal is determined by:

1. performing a forward Fourier transform of an input signal,
2. computing the logarithm of the power of each input frequency component,
3. performing an inverse Fourier transform on the logarithm data.

Cepstrum is useful for some types of mechanical vibration analysis.

The following custom command listing illustrates a cepstrum calculation using the on-board digital signal processor of the DAP 2400a:

```
/*  CEPSTRUM (p1, n, window, p2)
*      - compute 'n' point cepstrum from input data
*         in pipe 'p1' and send results to output
*         pipe 'p2'
*/
#include <cdapcc.h>
#include <math.h>

void main (PIB **plib)
{
    void **argv;
    int argc;
    PIPE *in_pipe, *out_pipe;
    int i, n, window;
    PBUF *inbuf, *outbuf;
    int *y_array;
    DSP_REQ *dsp_ptr1,*dsp_ptr2;

    argv = param_process (plib, &argc, 4, 4, T_PIPE_W,
                   T_CONST_W, T_CONST_W, T_PIPE_W);

    in_pipe = (PIPE *) argv[1];
    n = *(const int *) argv[2];
    window = *(const int *) argv[3];
    out_pipe = (PIPE *) argv[4];

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

```
        inbuf = open_pbuf (in_pipe, n);
        pbuf_min_cnt(inbuf) = n;
        outbuf = open_pbuf (out_pipe, 0);
        pbuf_ptr(outbuf) = pbuf_ptr(inbuf);
        pbuf_max_cnt(outbuf) = n;

        /* allocate an array for imaginary components */
        y_array = (int *) ralloc (2*n);

        /* initialize the window vector */
        switch (window) {
                case 0: window = FFT_RECTANGULAR;
                        break;
                case 1: window = FFT_HANNING;
                        break;
                case 2: window = FFT_HAMMING;
                        break;
                default: param_error();
        }

        /* allocate forward transform request header */
        dsp_ptr1 = dsp_alloc ();
        if (!dsp_request_init (dsp_ptr1, FFT_OP, n,
                        FFT_FORWARD | FFT_COMPONENTS | window,
                        pbuf_ptr(inbuf), y_array))
                                param_error();

        /* allocate reverse transform request header */
        dsp_ptr2 = dsp_alloc ();
        if (!dsp_request_init (dsp_ptr2, FFT_OP, n,
                        FFT_REVERSE | FFT_COMPONENTS | window,
                        pbuf_ptr(inbuf), y_array))
                                param_error();

        while (1) {
                int *x,*y;
                double scale = log(32767.0*32767.0)/32767.0;

                /* get real valued data and zero imaginary data
    */
                get_bpipe (inbuf);
                for (i=0; i<n; I++)
                        y_array[i] = 0;
```

```
            /* perform forward transform */
            dsp_send_request (dsp_ptr1);
            while (!dsp_done (dsp_ptr1))
                    task_switch();
            dsp_receive_result (dsp_ptr1);

            /* compute the logarithm of the transform data
*/
            x = pbuf_ptr(inbuf);
            y = y_array;
            for (i=0; i<n; i++) {
                    long int power;
                    double logpower;
                    power = (long int)(*x)*(long int)(*x) +
                             (long int)(*y)*(long int)(*y);
                    if (power > 0)
                            logpower = log((double) power);
                    else
                            logpower = 0.0;
                    *x = (int) (logpower / scale);
                    *y = 0;
                    x++;
                    y++;
            }

            /* perform the inverse transform */
            dsp_send_request (dsp_ptr2);
            while (!dsp_done (dsp_ptr2))
                    task_switch();
            dsp_receive_result (dsp_ptr2);

            /* send the results to the output pipe */
            pbuf_cnt(outbuf) = n;
            put_bpipe (outbuf);
    }
}
```

DSP Routines for the DAP 2400a and DAP 2416a

The following Developer's Toolkit for DAPL routines are compatible with the DAP 2400 and DAP 2400a. The same functionality is available for DAPL 2000-compatible products using FFT routines. See Chapter 14 for more information on FFT routines.

dsp_alloc

DSP_REQ *dsp_alloc ( )

The function **dsp_alloc** allocates a DSP request structure and returns a pointer to the structure. Individual fields of the DSP request structure are initialized by the function **dsp_request_init**.

dsp_done

```
int dsp_done (
   DSP_REQ *dsp_req
   )
```

The function **dsp_done** returns a nonzero value when an active DSP request has completed. This function should be called only after a call to the function **dsp_send_request** and before a call to the function **dsp_receive_result**.

dsp_receive_result

```
void dsp_receive_result (
   DSP_REQ *dsp_req
   )
```

The function **dsp_receive_result** returns the result of a completed DSP operation into the data arrays defined in the DSP request structure. This function should be called only after a call to the function **dsp_send_request** and only after the function **dsp_done** returns a nonzero value.

dsp_request_init

```
int dsp_request_init (
   DSP_REQ *dsp_req,
   int dsp_op,
   int n,
   int op_params,
   int *data1,
   int *data2
   )
```

The function **dsp_request_init** initializes a DSP request structure.

The parameter *dsp_op* selects the DSP processing operation; this always is set to the predefined constant FFT_OP. The parameter n is the size of the Fourier transform. This parameter must be a power of two. The allowed sizes are shown in the following table:

	Transform Size	
DAP Model:	**Minimum**	**Maximum**
DAP 2400a/4 and DAP 2416a/4	4	512
DAP 2400a/5	4	2048
DAP 2400a/6 and DAP 2416a/6	4	8192

The parameter *op_params* selects the type of fast Fourier transform and input/output options. This parameter should be the bitwise "or" of one predefined constant from each of the following columns:

```
FFT_FORWARD     FFT_RECTANGULAR     FFT_COMPONENTS
FFT_REVERSE     FFT_HANNING         FFT_AMPLITUDE
FFT_HAMMING     FFT_POWER
```

Constants in the first column select a forward or reverse transform. Constants in the second column select the type of input window (rectangular, Hanning, or Hamming). Constants in the third column select the type of transform output (complex numbers, amplitude, or power). The last two output options are defined for forward transforms only.

The final two parameters to **dsp_request_init** specify two C arrays which store the input/output data for the FFT operation. Each array must contain at least 'n' 16-bit locations.

A nonzero return code from **dsp_request_init** indicates successful parameter initialization.

dsp_send_request

```
void dsp_send_request (
   DSP_REQ *dsp_req
   )
```

The function **dsp_send_request** schedules a DSP operation. The function **dsp_done** will return a nonzero value when the DSP operation has completed.

18. Appendix C: Software Triggering Compatibility

The software triggering functions described in the main body of this document are available with DAPL 2000 versions 1.20 or later. The software triggering functions described in this section are backward compatible with all DAP models and operating system releases supported by the Developer's Toolkit for DAPL version 4.00.

Commands which required backward source code compatibility and operating system compatibility with DAPL 2000 prior to release 4.01 must use the triggering functions described in this appendix.

Using the Old Triggering Functions

The Developer's Toolkit for DAPL provides access to triggers through a set of system routines. 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.

The signaling task and the receiving tasks reference a pointer, called a trigger pointer, to a trigger data structure. Each task obtains this pointer from one of the task's parameters. The trigger's data structure includes a 32-bit number called a trigger count. The trigger count is the most recent sample number that the signaling task has processed.

If an input value corresponding to a trigger event is detected, the signaling task calls the function **`trig_assert`**.

The responsibilities of the signaling task are:

- Call the function **`trig_open_writer`** to open the trigger.
- Read data values from an input pipe, scanning for a trigger event.
- Increment the trigger count by one after each sample value is scanned. If a trigger event is asserted, increment the trigger count after asserting the event. Updating the trigger count is done using the following call, where variable t points to the trigger:

```
trig_update_writer (t, 1);
```

The trigger count can be incremented by a number larger than one if the task performs blocked pipe reads. The process is modified slightly for this case:

- Scan through the block of data samples until a trigger assertion is required.
- Update the trigger count by the number of scanned samples before the one requiring the trigger event.
- Assert the trigger event.
- Update the trigger count by one to cover the sample associated with the assertion.

Each receiving task must obtain a pointer called a trigger handle. A trigger handle is a pointer to the trigger's receiving data structure. The trigger handle data structure includes a 32-bit number called the receive count. The receive count is the most recent sample number that the receiving task has processed.

Receiving tasks normally wait for trigger assertions and then process input pipe data values relative to the location of the trigger event. In some cases, pipe data preceding the location of the trigger event are needed. The number of values preceding the trigger event is called the pretrigger count.

The responsibilities of a receiving task are:

- Call **`trig_open_reader`** to obtain a trigger handle.
- Call **`trig_wait_for_assert`**, specifying a pointer to an input data pipe, a trigger pointer, a trigger handle pointer, and a pretrigger count.
- When **`trig_wait_for_assert`** returns, remove and process data values from the input pipe. The minimum number of data values that must be processed is the value of the pretrigger count. After each data value is processed, the receive count must be incremented by one. If the trigger handle pointer is in `th`, the receive count is incremented using the following call:

```
trig_update_reader (th, 1);
```

The receive count can be incremented by a number larger than one if the task performs blocked pipe operations to fetch data. Update by one count for each sample taken from the data pipe.

Note that receiving tasks call **`trig_open_reader`** to initialize the trigger and return a trigger handle, while signaling tasks call **`trig_open_writer`** to initialize the trigger.

As examples of the trigger routines, simplified versions of the DAPL `LIMIT` and `WAIT` commands are listed below. `LIMIT` is a signaling task and `WAIT` is a receiving task.

The C code for `LIMIT` is:

```
/*  LIMIT2 (p1, region, t1)
*      - asserts trigger 't1' when data from pipe
*         'p1' enters 'region'
*/
#include <cdapcc.h>
void limit2 (PIPE *, int, int, int, TRIGGER *);

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 5, 5, T_PIPE_W,
             T_RFLAG, T_CONST_W, T_CONST_W, T_TRIGGER);
    limit2 ((PIPE *) argv[1], *(const int *) argv[2],
            *(const int *) argv[3], *(const int *) argv[4],
            (TRIGGER *) argv[5]);
}

void limit2 (PIPE *p, int rflag, int low, int high, TRIGGER
*t)
{
    long int d;
    trig_open_writer (t);
    pipe_open (p, P_READ);
    while (1)
    {
        d = pipe_get (p);
        if (rflag == R_INSIDE)
        {
            if   ((d >= low) && (d <= high))
                trig_assert (t);
        }
        else
        {
            if  ((d < low) || (d > high))
                trig_assert (t);
        }
        trig_update_writer(t, 1);
    }
}
```

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 when **`trig_assert`** is called.

The C code for `WAIT` is:

```
/*  WAIT1 (p1, t1, n1, n2, p2)
*      - transfer n1+n2 data values from pipe 'p1'
*         to pipe 'p2' when a trigger assertion
*         occurs on trigger 't1'
*/
#include <cdapcc.h>
void wait1 (PIPE *, TRIGGER *, int, int, PIPE *);

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 5, 5, T_PIPE_W,
             T_TRIGGER, T_CONST_W, T_CONST_W, T_PIPE_W);
    wait1 ((PIPE *) argv[1], (TRIGGER *) argv[2],
          *(const int *) argv[3], *(const int *) argv[4],
          (PIPE *) argv[5]);
}
```

```
void wait1 (PIPE *in_pipe, TRIGGER *t, int pretrigger,
    int posttrigger, PIPE *out_pipe)
{
    THANDLE *th;
    long int d;
    int I;

    pipe_open (in_pipe, P_READ);
    pipe_open (out_pipe, P_WRITE);
    th = trig_open_reader (t);
    while (1)
    {
        trig_wait_for_assert (t, th, in_pipe,
          pretrigger,1);
        for (i=0; i < (pretrigger+posttrigger); I++)
        {
            d = pipe_get (in_pipe);
            pipe_put (out_pipe, d);
            trig_update_reader (th, 1);
        }
    }
}
```

If a receiving task needs to test for triggering events, but must perform other duties while waiting for a trigger event to arrive, the task may call the function **`trig_get_assertion`** instead of **`trig_wait_for_assert`**. The function **`trig_get_assertion`** returns immediately, reporting whether it succeeded or failed to receive a new trigger assertion. A zero return value indicates that no trigger events are awaiting. In this case, the function also reports the current trigger count of the signaling task, so that the processing task can track progress of the signaling task when necessary. A nonzero return value indicates that one or more assertion events are posted and available for processing. In this case, the function also removes the first posted assertion from the trigger and reports the sample count for that event.

If a receiving task calls **trig_get_assertion**, its responsibilities are:

- Call **trig_open_reader** to obtain a trigger handle.
- Call **trig_get_assertion**, specifying a trigger pointer, a trigger handle pointer, and a count variable.
- Examine the return value of **trig_get_assertion** to determine whether an assertion has occurred. Use the value returned in the count variable for processing.
- Increment the trigger receive count based on the value returned in count:

```
trig_update_reader (th,
    count - trig_get_reader_cnt(th));
```

The following custom command waits for trigger assertions and prints the sample count of each assertion. This custom command is a combination of the DAPL TSTAMP and FORMAT commands.

```
/*  TSTAMP2 (t)
*      - prints the assertion count of all assertions
*         that occur on trigger 't'
*/
#include <cdapcc.h>
void tstamp_print (TRIGGER *);

void main (PIB **plib)
{
    void **argv;
    int argc;
    argv = param_process (plib, &argc, 1, 1, T_TRIGGER);
    tstamp_print ((TRIGGER *) argv[1]);
}

void tstamp_print (TRIGGER *t)
{
    THANDLE *th;
    unsigned long int count;

    th = trig_open_reader (t);
    while (1)
```

```
        {
            while ( !trig_get_assertion(t,th,&count) )
            {
                trig_update_reader (th,
                    (int) (count-trig_get_reader_cnt(th)));
                task_switch();
            }
            trig_update_reader (th,
                (int) (count-trig_get_reader_cnt(th)));
            printf ("Count=%ld \n",count);
        }
    }
```

The following pages provide a function reference for the old triggering functions.

trig_assert

```
void trig_assert (
   TRIGGER *t
   )
```

The function **trig_assert** generates a trigger assertion. The sample count of the trigger assertion is set to the current trigger count stored in the TRIGGER, so the trigger count must be current. See the **trig_update_writer** function for information about how to keep the trigger count current.

trig_get_assertion

```
int trig_get_assertion (
    TRIGGER *t,
    THANDLE *th,
    unsigned long int *count
    )
```

The function **trig_get_assertion** provides trigger information to a receiving task. Unlike the **trig_wait_for_assert** function, which will cause the task to wait until a trigger assertion occurs, the **trig_get_assertion** function attempts to obtain the value of the next asserted trigger without waiting. If it returns a nonzero value, one or more trigger assertions are waiting in the trigger. In this case, the next trigger assertion is removed from the trigger and `count` is set to the sample count of this assertion. If **trig_get_assertion** returns zero, no trigger assertions are waiting and `count` is set to the current trigger count. The current trigger count is the sample number that the asserting task currently is processing.

trig_get_reader_cnt

```
unsigned long int trig_get_reader_cnt (
    THANDLE *th
    )
```

The routine **trig_get_reader_cnt** returns the current sample count of a trigger handle. This function is most useful after calling the **trig_wait_for_assert** function, which automatically updates the sample count stored in the specified trigger handle as it waits for trigger events. When the processing task reawakens, it can determine the current value of the sample count by calling **trig_get_reader_cnt** before processing the event.

trig_get_writer_cnt

```
unsigned long int trig_get_writer_cnt (
   TRIGGER *t
   )
```

The routine **trig_get_writer_cnt** returns the current count of a trigger.

Note: **trig_get_writer_cnt** should be used only by a signaling task. Receiving tasks should check for active assertions before reading the value of a trigger count -- this is done using the routine **trig_get_assertion**. In a typical application, the **trig_get_writer_cnt** function is called after a sequence of updates to check the number of samples which have been processed.

trig_open_reader

```
THANDLE *trig_open_reader (
   TRIGGER *t
   )
```

The function **trig_open_reader** initializes a trigger and returns a trigger handle pointer. This function must be called by a task which receives and responds to trigger assertion events, prior to calling other triggering functions.

Note that **trig_open_reader** is used by receiving tasks while **trig_open_writer** is used by signaling tasks.

trig_open_writer

```
void trig_open_writer (
   TRIGGER *t
   )
```

The function **trig_open_writer** initializes a trigger. This function must be called by a task which asserts trigger events, prior to calling other triggering functions.

Note that **trig_open_writer** is used by signaling tasks while **trig_open_reader** is used by receiving tasks.

trig_set_writer_cnt

```
void trig_set_writer_cnt (
    TRIGGER *t,
    unsigned long int count
    )
```

The function **trig_set_writer_cnt** is used by a signaling task to set the value of a trigger's count. This function is useful for triggering commands which generate events at predetermined times, for example, automatic sweep generation.

The function **trig_set_writer_cnt** always must set the trigger count to a value which is greater than or equal to the current trigger count.

trig_update_reader

```
void trig_update_reader (
   THANDLE *handle,
   int val
   )
```

The function **trig_update_reader** increments the count of a trigger handle by *val* samples. This informs the trigger that any events with a lesser or equal sample count do not need to be maintained in the trigger pipe, and informs the DAPL system that for all sample counts less than or equal to the updated handle count, any buffered data reserved for this task are no longer needed, so corresponding memory can be released.

This function must be used after skipping or using all data samples. When data is obtained in blocks, use the following sequence of operations:

1. call **trig_update_reader** for those samples which are not used,
2. process the samples which are used,
3. call **trig_update_reader** for those samples which were just used.

Note that **trig_update_reader** is used by receiving tasks while **trig_update_writer** is used by signaling tasks.

trig_update_writer

```
void trig_update_writer (
   TRIGGER *t,
   int val
   )
```

The function **trig_update_writer** increments the count of a trigger by *val* samples. This informs the trigger processing commands of progress of the signaling task. It also makes the trigger count current prior to signaling an event.

When the command processes data in blocks, it is important to update the writer using the following sequence of steps:

1. call function **trig_update_writer** for the number of samples processed without asserting any events,
2. assert the event,
3. call function **trig_update_writer** after the event is asserted.

Note that **trig_update_writer** is used by signaling tasks while **trig_update_reader** is used by receiving tasks.

trig_wait_for_assert

```
unsigned long int trig_wait_for_assert (
   TRIGGER *t,
   THANDLE *th,
   PIPE *p,
   unsigned long pre_count,
   int mult
   )
```

The function **trig_wait_for_assert** removes data from a pipe while waiting for a trigger assertion. It automatically updates the number of processed samples in the trigger handle count. As the trigger count increases, **trig_wait_for_assert** removes data from pipe *p*, always leaving *pre_count* pretrigger samples in the pipe. When **trig_wait_for_assert** returns, pipe *p* contains data beginning *pre_count* samples before the next trigger assertion.

The *mult* parameter specifies a trigger rate time correction and always should be the number one.

The value returned by **trig_wait_for_assert** is the sample count of the trigger assertion.

Index