DAPL 2000 Manual

Data Acquisition Processor
Analog Accelerator Series

Version 6.00

Microstar Laboratories, Inc.

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

Part Number MSDAPL2000M600

Contents

Section I. Overview

1. Introduction

The Data Acquisition Processor from Microstar Laboratories is a complete data acquisition system that occupies one expansion slot in a PC. Data Acquisition Processors are suitable for a wide range of applications in laboratory and industrial data acquisition and control.

Two other manuals complement the DAPL Manual:

- The Applications Manual introduces the Data Acquisition Processor by showing how to set up a wide variety of data acquisition applications.
- A Data Acquisition Hardware manual contains installation instructions and a configuration reference.

New and Changed Information

This edition of the DAPL Manual contains information for version 2.05 and above of DAPL 2000. While many user functions of DAPL 2000 are consistent with other versions of the DAPL operating system, there are some fundamental differences:

- DAPL 2000 is a 32-bit operating system and is compatible only with Data Acquisition Processors that have a 32-bit processor.
- Appendix A discusses command forms that are superceded in current versions of the DAPL 2000 system.

2. Introduction to DAPL

A Data Acquisition Processor is capable of simultaneously running sophisticated real-time data capture, data processing, and signal generation tasks, under control of the DAPL operating system. The DAPL system supports a broad range of hardware, timing, and processing configurations, and provides a library of built-in processing functions for data selection, conversion and on-line analysis tasks.

This versatility is made accessible through a very powerful, high-level scripting language. The configuration commands are typically organized as text files and downloaded to the DAPL system, as needed, by a software application. A single command line in the DAPL configuration script is the equivalent of hundreds of lines in most programming or scripting languages.

Architectural Basics

The DAPL operating system is downloaded into Random Access Memory (RAM) of the Data Acquisition Processor during the boot sequence of the host processor. A configuration describing the input sampling, the output signal generation, and the data processing is then downloaded by application software. That configuration is translated automatically into a set of tasks. When the application tells the DAPL system to run the configuration, a multitasking scheduler coordinates acquisition, signal generation, and communication events. Streams of data are routed to processing tasks. The results of the processing are usually transferred back to the host software — but not always. Sometimes the Data Acquisition Processor is configured to directly control various output signals independent of the host software system.

DAPL tasks communicate through buffering structures called pipes. Tasks place data in pipes and remove data from pipes. A pipe holds data temporarily until the next task in the processing sequence is ready for that data. DAPL keeps the data in correct sequence by enforcing a first-in, first-out discipline. Data in pipes are considered available for all tasks to read, but only those tasks that specify the pipe as a data source are allowed access. Each task that asks to read data from a pipe will see all of the data from that pipe as if it were reading its own private copy.

During processing, data samples are recorded, output signal data are consumed, and intermediate computed values are used and discarded. All of these activities require intermediate memory buffers. The DAPL system takes care of all task synchronization and all memory buffer management. For example, if a task is temporarily unable to continue because it has read all of its input data or because no space is available in its output pipe, the DAPL system will suspend that task temporarily, scheduling other tasks to consume or provide data. Depending on the demands of the processing, the amount of buffering memory in each pipe expands and shrinks dynamically.

The Data Acquisition Processor organizes the physical channels into configurable groups, known as channel lists. The characteristics of each channel in the channel list are also configurable: signal source, gain, etc. Data captured from the various channels are moved as efficiently as possible into memory and grouped according to the channel list. This mixed organization is rather awkward for processing the data, so the DAPL system provides a mechanism called “input channel pipes” for accessing the data in a more orderly fashion. The “input channel pipes” are treated like any other data pipes for accessing data channels individually or in groups. In a similar fashion, “output channel pipes” provide an orderly mechanism for collecting data for clocked output signal updates.

The DAPL system also enforces rules of “scheduling fairness.” That is, a task that demands a lot of CPU computing resource is not allowed to exclude other tasks. If a task exceeds its time allocation or cannot proceed for any other reason, the DAPL system goes on to schedule the next task.

A unique mechanism for inter-task coordination called “software triggers” is also provided. Software triggers are really a kind of pipe, but instead of passing ordinary data, they pass information about where to locate data of special interest. Sometimes this information is called “trigger events.” These are not really time-events in the ordinary sense, rather, they are indications of where to find data. For example, suppose that a block of 100 data samples is to be retained every time the block contains an odd number of positive values greater than 10000 but no negative values. A task responding to these “trigger events” would know exactly which data to retain. The data to be collected sometimes corresponds to a moment in real-time prior to the triggering event, which seems implausible. How can the system respond to an event before it happens? This “pre-trigger sampling,” as it is often called, is really an illusion. It doesn’t have anything to do with sampling. It is really intelligent data management.

Data Processing Configuration

The configuration script for a DAPL application is very different from a programming language in the usual sense. Most scripting languages are essentially procedural. That is, they specify "do this, then if this condition is satisfied, do this…" and so forth. Even object-oriented languages operate this way, though their procedures are bound to data objects: "If this object receives this message, do this, and if this condition is satisfied issue this message to that object…" In contrast, a DAPL command specifies configuration rather than immediate action. It is sometimes useful to say that only one command, the **START** command, actually executes. The rest of the commands just specify what to start. (This is not strictly true. Still, it is a useful concept.)

In fact, all commands execute, but in most cases the effect is to configure some aspect of the system, not to immediately perform an application process.

The commands can be assigned to the following categories.

System Commands

These commands configure the system environment, provide operating status information, and start or stop application configurations established by the other commands.

A task called the "command interpreter" is always available and active. It is responsible for receiving the text of all commands and providing the appropriate response. The command text is usually received through the built-in `$SYSIN` text channel, but commands can also arrive from other sources. When it receives a system command, the command interpreter executes this command immediately. Some commands, such as **START** or **STOP**, control the execution of an application configuration. Commands such as **DISPLAY** report current status information. Commands such as **PAUSE** affect the operation of the interpreter task, for example, to allow time for processes to complete. Commands such as **OPTIONS** affect the system operating environment. Commands such as **RESET** clear the operating environment for beginning a new application configuration.

For example, the following sequence of commands will collect data using sampling procedure `A` for three seconds, but allow three additional seconds to complete the data analysis using processing procedure `ANALYZE`.

```
START  A, ANALYZE
PAUSE  3000
STOP   A
PAUSE  3000
RESET
```

Defining Commands

These commands define shared data elements used by processing tasks.

The defining commands can be considered a special set of system commands, whose effect is to define a named data element. Because these named elements are known to the system, processing tasks can use them to share data access. Pipes establish connections between tasks and are always shared elements, so it is always necessary to declare pipes before defining the processing tasks that use them. A shared data element persists in memory, and will remain defined until removed by a system command such as **ERASE** or **RESET**.

For example, the following commands define a variable called `LAST_EVENT` and a software trigger called `T_EVENT`.

```
VARIABLE   LAST_EVENT
TRIGGER    T_EVENT   MODE=NORMAL  HOLDOFF=128
```

`CONSTANT` elements in the DAPL configuration maintain a fixed value while the configuration is running. `VARIABLE` elements, however, are "active" and any tasks that access variables can change the values at any time. Furthermore, the value of a variable can carry over from one run to the next, unless the variable is erased or explicitly assigned a new initial value.

Input Configuration Commands

These commands configure input channel lists, physical input channels, timing, and sample collection sequences.

Input configuration commands occur in a group that begins with an **IDEFINE** command and ends with an **END** command. Special commands appear inside of the input configuration, such as the **SET** command for assigning input channel pipes to input pins, and the **TIME** command for establishing sampling intervals.

For example, the following input procedure configures input sampling to capture a sample every 10 microseconds, taking samples alternately from the digital port and single-ended analog channel 1 with gain of 10.

```
IDEFINE   ALT
  CHANNELS 2
  SET  IP0  B
  SET  IP1  S1  10
  TIME 10
END
```

Output Configuration Commands

These commands configure output channel lists, output converters, timing, and clocked signal update sequences.

The isochronous (clocked) output configuration commands occur in a group that begins with an **ODEFINE** command and ends with an **END** command. Special commands appear inside of the output configuration, such as the **SET** command for assigning a data channel to an output device, and the **TIME** command for establishing update timer intervals.

For example, the following configuration generates a periodic signal of length 250 samples at analog output converter 0, clocking a new update every 400 microseconds.

```
ODEFINE   SIGNAL   1
  SET  OP0  A0
  TIME   400
  CYCLE  250
END
```

Processing Configuration Commands

These commands configure a network of processing tasks that consume, process, modify and transfer data in various ways. The configuration can be considered a simple list without an enforced order of execution. Pipes serve as the "wiring" — or "plumbing" if you like — for routing the outputs of a task to the inputs of subsequent tasks. If data are available to process, a task can execute. Otherwise, some other task will run. This kind of configuration is known as a "dataflow model."

A task definition uses a processing command, but isn't one. This distinction is important, and explains why a command like **SKIP** or **LIMIT** can be used any number of times. If you like, you can think of each task as a separate thread of execution

through a body of shared command code, each thread of execution using its own set of data sources and destinations.

Most task definition commands in the processing procedure consist of:

1. A command name.
2. Keyword options. Only a few commands use these.
3. A parameter list. The parameter list specifies various explicit or shared data elements, such as the pipes serving as data sources or destinations.

For example, the `AVE` processing configuration below alternately processes and discards data blocks of length 100 elements, and computes the average value for each retained block. Pipe `PSKIP` is used to transfer the intermediate data between tasks. Assume that pipes `PDATA`, `PSKIP` and `PAVE` have been defined previously.

```
PDEFINE   AVE
  SKIP(  PDATA,  0,100,100,  PSKIP)
  AVERAGE(PSKIP,100,PAVE)
END
```

There is a second kind of task definition called a DAPL expression that looks very different. The DAPL expression in the following example takes each value from pipe `P1` and rounds it to the next smaller multiple of 10, placing the results into pipe `P2`.

```
P2 = (P1/10) * 10
```

This looks very much like an assignment statement in an ordinary programming language, but there is a big difference. An assignment statement operates on one value. A DAPL expression converts one or more streams of data to another stream. For more information about DAPL expressions, see Chapter 7.

Task definitions must refer to command names that are known to the system, hence, these must either be processing commands built into the system or custom-programmed commands that previously have been downloaded into the Data Acquisition Processor memory.

The tasks defined in a processing procedure remain available, though inactive, until the system command **START** activates the procedure.

General Rules for Command Syntax

Case

The DAPL system does not distinguish between upper case and lower case letters. They may be used interchangeably. For example, the names "`AaA`" and "`aAa`" are considered to be one and the same within the DAPL system.

Names

Names assigned to shared elements must begin with an alphabetic character and may contain up to 22 additional alphabetic, numeric, or underscore characters. Some pre-defined system names can begin with a "`$`" character, for example, the `$BINOUT` pipe.

Uniqueness

Names assigned to shared data elements must be distinctive from all built-in command names, pre-defined symbols, reserved command keywords, and other user-assigned names.

Abbreviations

Some system command names can be abbreviated. This is not generally recommended in configuration files, but is sometimes useful for direct interaction with the command interpreter. See the listings for the individual commands for information about the accepted abbreviations.

Blanks

Blank characters serve as separators, but have no meaning except within string constants. With rare exceptions, blanks can appear anyplace where other separator or termination characters appear, for example, the expressions "`A=B`" and "`A = B`" have equivalent meanings. However, the commands "`AB = C`" and "`A  B = C`" are not equivalent, because the intervening blank separates characters `A` and `B` into two distinct names.

Comments

DAPL supports "trailing comments" that begin with a "`//`" character pair outside of a string constant. Comments continue to the end of the current line. Continued lines

cannot be commented. Comments have the same interpretation as blanks, that is, they act as a separator but otherwise have no meaning.

Note: Trailing comments beginning with ";" are accepted but discouraged. Avoid comment lines that begin with the character sequence ";$" or ";%". These are valid and acceptable in the DAPL environment, but these notations are interpreted specially by some Microstar Laboratories software utilities.

Continuation

A very long command line can be temporarily terminated with a backslash "\" character at the end of the line, with the command continued on the next line. The effect is the same as if the command were on a single very long line. String constants cannot be continued this way, however. Continued lines cannot have trailing comments.

NOTE: Some commands provide alternative notations for line continuations. See for example the **VECTOR** command.

Line Termination

A combination of a carriage-return character followed immediately by a linefeed character, or a linefeed character followed immediately by a carriage-return character, is treated as a single line termination character. Otherwise, each carriage-return or linefeed character terminates a separate line.

Numbers

Number constants can be entered in a decimal or hexadecimal notation. Hexadecimal constants are prefixed by a "$" character without any intervening blanks. For example:

```
$ABCD
```

The hexadecimal notation is treated as a pattern of bits, and the value depends on the interpretation of the sign bit. If the value above is placed into a 16-bit data pipe, the high-order bit is a 1 so the value is -21555. But if placed into a 32-bit long pipe, the high-order bit is not in the sign bit position, so the value is 43981.

For certain command notations, numbers must be specified in a decimal notation. Two examples:

```
TIME  $1F     : invalid!
TIME  $8      : invalid!
```

Channel Pipe Notations

Input and output channel pipes are accessed using input channel pipe and output channel pipe notations, respectively.

An input channel pipe notation has two parts, an `IPIPE` notation and a channel specifier, with an optional separating blank. The `IPIPE` keyword can be abbreviated to `IP`. An output channel pipe notation is similar, except for the `OPIPE` or `OP` keyword.

The channel specifier part has two forms, a single channel specifier or a channel pipe list.

The following example shows an input channel pipe notation with a single channel specifier. This form is used in an input procedure or output procedure for assigning a signal pin to a channel pipe.

```
SET   IPIPE0  S9  // input procedure
SET   OP 0    A0  // output procedure
```

A similar notation can be used in a task definition parameter list to access the channel pipes. For example:

```
COPY( IP  0, $BINOUT )
SINEWAVE(5000,200,OP0)
```

The channel pipe list notation specifies sets of samples to be collected. Channels can be listed individually, or ranges of numbers can be specified using a “dot-dot” notation, but the channels must always be in a strictly ascending order. The specifications are separated by commas, and the list enclosed in parentheses. For example, the following command will transfer the data from the first four input channel pipes directly to the first four output channel pipes.

```
COPY(  IP (0,1,2,3),  OP (0..3) )
```

Usually, a shared constant or vector value is also accepted as a channel or channel list specification. For example, if constant `CCHAN` is defined

```
CONSTANT  CCHAN = 1
```

then the following would be an acceptable input channel specification:

```
IPIPE  CCHAN
```

The blank separator is required in this example; otherwise the characters "IPIPECCHAN" would be interpreted as a shared element name rather than an input channel pipe.

Task Parameter Notations

Most task definitions require specification of a number of configuration parameters. Tasks that require no parameters can omit the parameter list. The list consists of a number of parameter items, separated by commas, and enclosed in parentheses. The elements in the list can be:

1. Explicit constant values
2. Named constant values
3. Named variable values
4. Explicit vectors, enclosed in parentheses, with the vector terms separated by commas
5. Named vectors
6. Explicit strings
7. Named string values
8. Pipe names
9. Input or output channel pipe notations
10. Software trigger pipe names
11. Range notations (see the discussion below)

The **FORMAT** command allows some additional formatting notations. See the **FORMAT** command reference pages for full information.

Range Notations

A region notation is a special combination of task definition parameters. It consists of three parameters: a special reserved name INSIDE or OUTSIDE, a 16-bit numeric parameter specifying a lower range limit, and a 16-bit numeric parameter specifying an upper range limit. The terms are separated by commas. If the range limits are variable names, these limits can be adjusted dynamically while the task is running.

The range values usually represent data values, but sometimes they constrain an index. The INSIDE condition is satisfied if the value under test, data or index, is greater than or equal to the lower limit, and also less than or equal to the upper limit. The

OUTSIDE condition is satisfied if the value under test is strictly less than the lower limit or strictly greater than the upper limit.

In the following example, a **LIMIT** command uses a range specification to test for data falling outside of the range -32768 to 0 (that is, any strictly positive number).

```
LIMIT( P1,    OUTSIDE, -32768, 0,     T1)
```

In the following example, the range specification instructs the **FINDMAX** command to examine only the first 128 values from blocks of data containing 256 samples.

```
FINDMAX( P1, 256,    INSIDE,0,127,    P2 )
```

About Efficiency

A processing configuration might use perhaps two dozen commands to invoke very powerful processing options. This processing is specified using a very high-level DAPL configuration script. But very high level languages tend to achieve their power at a cost to speed and efficiency. When the application runs, what is the speed penalty, compared to (say) a hand coded application in Assembly language?

The answer is: *No penalty*. How is this possible?

Recall, the high level commands used to configure an application do not execute directly, rather, they configure system elements. This configuration is typically expressed in terms of interrelated data structures, not executable code. The code that actually runs, in all of the data sampling, updating and processing tasks, is highly optimized at the machine code level. Execution of the configuration script is slow (by comparison), but this does not apply to the run-time processing.

Direct Interaction with the Interpreter

Most of the time, DAPL configuration files are downloaded through the $SYSIN command pipe under software control, using software facilities such as the DAPIO32.DLL, or utility programs like DAPview for Windows. But it is also possible to transfer configuration information manually and directly to the DAPL interpreter. For this kind of manual operation, typically the interpreter is put into the SYSINECHO mode using the **OPTIONS** command. Any input text is echoed back to the sender through the $SYSOUT text pipe in the manner of a data terminal operating in half-duplex mode. The interpreter will also send prompt characters to indicate when it is expecting the next command. Normally the prompt character is the "#" character. After sending an **IDEFINE**, **ODEFINE** or **PDEFINE** command, the prompt character is

changed to ">" indicating that the interpreter is awaiting the commands that make up the body of the input, output or processing definition respectively. After the terminating **END** command, the "#" prompt returns.

About Custom Processing Commands

While the capabilities of the built-in processing commands are extensive, it is not possible for any given set of commands to meet every possible processing requirement. The DAPL system allows developing and downloading command modules specifically suited to special data acquisition applications. Once downloaded into the Data Acquisition Processor memory, commands in a module have the same status as a built-in task definition command. The most common reasons for employing custom command modules are:

1. Specialization and Extension.
   A custom command can provide processing features that are not available using pre-defined commands. For example, there are hundreds of specialized digital signal processing algorithms besides the basic filtering and transform operations provided by the operating system. Specialized operations can be downloaded to act in combination with, or as replacements for, the built-in commands.

2. Combination for Efficiency.
   Some applications require sequences of processing operations. When applied to very long input channel lists, this can mean hundreds of processing tasks, transferring data through hundreds of data pipes. The operating overhead of any one pipe or task is small, but multiply this by several hundred and this can lead to inefficiencies that compromise processing capacity. A custom command can perform the equivalent of several pre-defined processing operations, improving processing efficiency.

3. Combination for Speed.
   A custom command task can substitute for very complicated DAPL expressions. Some complex computations can be computed more quickly if compiled to native machine code and optimized by a compiler.

4. Real-time Response.
   When it is important to respond to time-critical events, having a large number of tasks can be a hazard. Any tasks that execute between the time that an event is detected but before the response can be sent introduce a delay, also known as "latency." If the delay is unacceptable, processing can often be packaged into a limited number of tasks, sometimes as few as one, so that the latency is bounded to an acceptable level.

Custom command modules are written in the C++ programming language, compiled into a binary code image, and downloaded to a Data Acquisition Processor using one of the utilities such as CDLOAD32 or DAPview for Windows. Tools for preparing the command code, all of the required utilities, and lots of working examples are provided by the Microstar Laboratories Developer's Toolkit for DAPL.

3. System Commands

DAPL system commands start and stop sampling, set system options, request status information and set initial conditions. They are executed immediately when received by the DAPL command interpreter.

The following system commands are built into the DAPL operating system:

DIAGNOSTIC	test Data Acquisition Processor hardware
DISPLAY	display symbol and system status information
EDIT	modify input and output procedures and com pipes
EMPTY	empty all data from a pipe
ERASE	remove a symbol
FILL	add data values to a pipe
HELLO	return a line including the DAPL version number
LET	change the value of a variable or constant
OPTIONS	change a system option
OUTPORT	define output expansion board types
PAUSE	pause DAPL interpreter
RESET	reset DAPL interpreter
RESTART	perform a power-up restart of DAPL
SAMPLEHOLD	wait for input processing to stop
SDISPLAY	display information about symbols
START	start input, output, and processing procedures
STATISTICS	display task statistics
STATUS	display system status
STOP	stop input, output, and processing procedures

4. System Element Definition Commands

A defining command is a special kind of system command. It executes immediately, but its effect is to construct a persistent, shared system element with an assigned name. Execution of the command will:

- allocate memory for the element
- assign and record its name
- initialize the memory with appropriate values

The element is inactive until a configuration that uses it is started. A defined element will remain defined until removed by an **ERASE** command or a system initialization command such as **RESET**.

Some elements reserve data storage memory when the element is defined. Other elements will allocate data memory dynamically when the processing configuration runs. For example, the data storage areas for pipes will grow and shrink as data are accumulated or extracted.

The following system element definition commands are built into the DAPL operating system:

CONSTANTS	define constants
PIPES	define pipes
STRING	define a string
TRIGGERS	define triggers
VARIABLES	define variables
VECTOR	define a vector.

Once an element is defined, it can be used in DAPL processing configurations. For example, a variable `VALUE` and a pipe `PSTREAM` can be combined by a DAPL expression `PSTREAM + VALUE`.

5. Input and Output Configuration Commands

Input configuration commands establish an operating configuration for input sampling hardware. Output configurations establish an operating configuration for output updating hardware. Other than this major difference, input and output configurations are similar in many ways. Both are closely related to the hardware features of the Data Acquisition Processor, so the supported options may vary with the Data Acquisition Processor model. Both are optional. More than one input and output procedure can be defined, but at most one of each can run at any one time. Both associate physical pins to logical data channels called channel pipes. Both use sets of configuration commands to select buffering, clocking, hardware triggering, and time interval options.

The commands that make up an input or output procedure definition are the equivalent of a single system element defining command, in the sense that together they define a data structure, though it is a complex one.

Some Data Acquisition Processor models are specialized for high speed data capture only, and do not support output updating configurations.

Input Configuration Commands

An input configuration begins with an **IDEFINE** statement, ends with an **END** statement, and contains a number of commands that configure input sampling options.

IDEFINE	begin an input configuration definition
END	complete an input configuration definition

CHANNELS	configure the number of channels to receive data
CLCLOCKING	select the channel list clocking mode
CLOCK	select internal or external clocking
COUNT	specify the number of samples to acquire
GROUPS	configure the number of channel groups to receive data
GROUPSIZE	define the number of channels in a programmable input channel group
HTRIGGER	select the hardware triggering mode
SET	associate a channel pipe with a physical pin or pin group
TIME	select the sampling interval
VRANGE	set configurable input voltage range limits
UPDATE	select continuous input operation or burst input operation

Output Configuration Commands

An output configuration begins with an **ODEFINE** statement, ends with an **END** statement, and contains a number of commands that configure output updating options. Output configurations are not supported on DAP 3400a boards, DAP 4400a boards or DAP 5400a boards, which are specialized for data sampling only.

ODEFINE	begin an output configuration definition
END	complete an output configuration definition
CLCLOCKING	select the channel list clocking mode
CLOCK	select internal or external clocking
COUNT	select the number of output updates
CYCLE	specify the cycle length for periodic output data
HTRIGGER	select hardware triggering mode
OUTPUTWAIT	select the number of samples to buffer before updating begins
SET	associate a channel pipe with a physical pin
TIME	select the output update interval
UPDATE	select continuous output operation or burst output operation

6. Task Definition Commands

Processing configurations define groups of tasks known collectively as a "processing procedure." A processing procedure begins with a **PDEFINE** statement, ends with an **END** statement, and contains commands that define processing tasks. Task definitions can apply any of the built-in processing commands or a command in a downloaded command module. A DAPL expression can also define a processing task. A processing command can be used any number of times, assigning different data sources and destinations for each task.

The tasks created in this manner become available for execution, but they do not run until their processing procedure is activated by the **START** system command. When the **START** command selects the processing procedure containing a task, the **START** command will

- allocate stack space for the task
- set up a data storage area for the task
- assign a scheduling entry for the task

Once running, a task locates the system elements specified by its parameter list and executes the body of command code. This code is separate from the DAPL interpreter, and is optimized for efficient execution. A running task can access data sources and data destinations, including: read data from data pipes, perform computations, write result data to data pipes, access shared variable or vector data, or report software trigger events.

The following task definition commands are built into the DAPL operating system:

PDEFINE	begin a processing configuration
END	complete a processing configuration
ABS	compute absolute values
ALARM	generate digital alarm signals
AVERAGE	average pipe data
BAVERAGE	perform block averaging
BINTEGRATE	integrate blocks of data
BMERGE	merge blocked data
BMERGEF	merge blocked data with identifying flags
BPRINT	transfer binary input channel pipe data to the PC
CABS	compute the sum of squares of the values in two pipes

CHANGE	scan for changes in data
COMPRESS	compress data flow for inputs that change infrequently
COPY	copy the data in a pipe into several other pipes
COPYVEC	copy data from a vector to a pipe
CORRELATE	compute cross correlations
COSINEWAVE	generate cosine waveforms
CROSSPOWER	compute cross power spectrum
CTCOUNT	accumulate long word counts from counter timer board data
CTRATE	compute frequencies from counter timer board data
DACOUT	send data to an analog output port
DECIBEL	convert positive values to decibels
DELTA	compute the differences of the values in a pipe
DEXPAND	calculate data required for digital output port expansion
DIGITALOUT	send data to a digital output port
DLIMIT	scan data for slopes that are out of range
EXTRACT	extract single bits from word data
FFT	compute fast Fourier transforms, emphasizing speed
FINDMAX	find the locations of maxima in blocks of data
FIRFILTER	apply digital filtering and reduce data volume
FIRLOWPASS	apply predefined lowpass digital filtering
FORMAT	format data as text and transfer to the PC
FREQUENCY	determine frequencies of trigger assertions
HIGH	compute maxima of blocks of data
INTEGRATE	compute the running integral of pipe data
INTERP	interpolate with a lookup table
LCOPY	copy the data in a pipe into several other pipes with minimal latency
LIMIT	scan data for values that are out of range
LOGIC	scan binary data for bit transitions
LOW	compute minima of blocks of data
MERGE	merge data from several pipes into one pipe
MERGEF	merge binary data from several pipes, adding identifying flags
NMERGE	merge different quantity of data from several pipes into one pipe
NTH	remove excess trigger events
OFFSET	perform an offset adjustment
PCASSERT	generate triggers based on PC control
PCOUNT	count the number of values placed in a pipe
PEAK	search for maxima and minima of pipe data
PID1	compute data for closed-loop process control, reducing the control error to 0
POLAR	convert from rectangular to polar coordinates

PRINT	print all input channel pipe data to the PC
PULSECOUNT	count the number of digital input pulses
PVALUE	determine the most recent value in a pipe
PWM	perform pulse width modulation
RANDOM	generate pseudorandom numbers
RANGE	remove data values that are out of range
RAVERAGE	compute running averages of pipe data
REPLICATE	copy data, repeating each value a specified number of times
RMS	compute root mean square values
SAWTOOTH	generate sawtooth waveforms
SCALE	scale pipe data and add an offset
SCAN	wait for a group of input pins to be sampled before transferring data
SEPARATE	separate merged data
SEPARATEF	separate flagged merged data
SINEWAVE	generate sine waveforms
SKIP	delete selected blocks of data
SQRT	compute square roots
SQUAREWAVE	generate square waveforms
TAND	combine triggers with logical ‘and’
TCOLLATE	combine triggers producing a combined event stream
TFUNCTION1	calculate transfer functions from frequency domain data
TFUNCTION2	calculate transfer functions from cross-power spectra and auto-power spectra
TGEN	generate periodic triggers
THERMO	perform thermocouple linearization on data
TOGGLE	test trigger events for alternating `ON` and `OFF` events
TOGGWT	collect data between alternating `ON` and `OFF` events
TOR	combine triggers with logical 'or'
TRIANGLE	generate triangle waveforms
TRIGARM	allow a task to arm or disarm software triggers
TRIGRECV	recover encoded software trigger information
TRIGSCALE	modify a stream of trigger events
TRIGSEND	transfer trigger information to another DAP
TSTAMP	convert trigger assertions to time stamps
VARIANCE	compute the statistical variance of pipe data
WAIT	wait for a trigger event and transfer trigger data to a pipe
WAVEFORM	generate analog waveforms

7. Task Definition Using DAPL Expressions

DAPL expressions provide flexible means for performing arithmetic and bitwise operations on streams of data. While an expression statement might look like an "assignment statement" from various familiar programming languages, it is actually much more. Each expression defines a task that reads from pipes, input channel pipes, or variables, performs arithmetic and bitwise operations, and puts results into a pipe, output channel pipe or variable.

Expression Syntax

A DAPL expression consists of three parts: a "target" which is the destination for the computed results, an "assignment operator" represented by an equal sign, and an "expression" consisting of constants, names, and operators:

```
<target> = <expression>
```

Examples:

```
P3     = P1 + (P2 & $07F)
OPIPE0 = P5*P6 - P7
```

Target

`<target>` specifies the destination for computed results. The target must be a defined variable or pipe that can accept arithmetic data.

Expression

`<expression>` is a combination of operands and operators that specify the computations to perform.

Expression Operands

Operands are terms that provide data. They can be of the following types:

Named Constant: a shared value of numeric type defined by the **`CONSTANTS`** command. This value is "locked" once the DAPL configuration is started.

Explicit Constant: a numeric value specified directly as a term in the expression. Decimal, hexadecimal, or floating point constant notations can be used. The notation is unambiguous and is not overridden by the global **OPTIONS** DECIMAL=OFF mode that controls data formats.

Variable: a shared word, long, float, or double value residing in shared storage reserved by the **VARIABLES** command. Unlike a named constant, a variable value can be changed at any time by another task or the PC host. The expression task attempts to use the most current value of the variable for its computations.

Pipe: a stream of numeric values of word, long, float, or double type. A pipe operand names a stream of data rather than individual values. Any pipe that can provide numeric data can be used. The pipe can be a user-defined pipe, input channel pipe, or communications pipe. If no data are available from the pipe, the task suspends execution at that point, waiting for data to arrive.

When pipe operands are evaluated, the expression attempts to extract from the pipe as many values as it can, consistent with the buffering mode. When buffering is off, only one value is fetched at a time.

Because pipe operands refer to streams of data, not to individual values, each reference to the name of a pipe obtains access to the entire stream of data. It is as if there are multiple, separate and independent copies of the data stream. Consequently, the following two commands are equivalent:

```
P3 = P1 + P1 + P1
P3 = P1 * 3
```

Integer scalars can be expressed in a 32-bit hexadecimal notation. In general, it is a good idea to represent all 32 bits, explicitly showing sign extension bits, particularly when the values are to be treated as numeric (as opposed to bitwise) data.

Example: Select the high-order 8 bits from a 16-bit data value.

```
POUT = P1 &  $0000FF00
```

Expression Data Types

Inside of DAPL expressions, all values, whether taken from operands or computed, are classified into one of three internal data types:

- Fixed point
- Bitwise
- Floating point

Data from word or long fixed point pipes or variables are represented as fixed point internal data. Data from float or double pipes or variables are represented as floating point internal data.

Data types depend on the operators that are applied.

Expression Operators

Operators can be categorized as
- arithmetic operators
- bitwise operators
- shift operators
- negation operators
- grouping operators

Arithmetic operators

Arithmetic operators are infix operators that perform the usual arithmetic operations. They include:

addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`

Examples:

```
P1 + 100

RAW * SCALE1 / SCALE2
```

Arithmetic operations applied to bitwise or fixed point data types yield fixed point values. Arithmetic operations for which one or more of the values is floating point yield a floating point result.

Bitwise operators

Bitwise operators are infix operators that perform the usual Boolean operations of setting, clearing, and inverting patterns of bits. These operations include the following:

bitwise-and	&
bitwise-or	|
bitwise-xor	^

Examples:

```
P1 & $FF00

P1 | P2
```

Bitwise operations cannot be applied to floating point operands or floating point intermediate results. A bitwise operation applied to fixed or bitwise data types yields a bitwise result.

Shift operations

Shift operations are infix operators that shift patterns of bits left or right. The result of a shift is a bitwise value.

The first operand specifies the initial bit pattern and the second operand specifies the number of bit positions to shift. Shift operations cannot be applied to floating point operands or floating point intermediate results. Fixed point values specified as bit patterns are treated as if they were bitwise data. The number of bit positions to shift must be a fixed point number, or a bit pattern interpreted as a fixed point number, in the range 0-31.

The shift operations include the following:

shift left	<<
shift right	>>

Example:

```
P2 = (P1 >> 12) & $000F
```

A left shift has the same effect as multiplication by a power of two, except for the data type of the result. Vacated low-order bit positions are filled by zeroes.

The effect of a right shift is similar to a division by a power of two. The DAPL system replicates and propagates whatever bit value happens to be in the high-order (sign) bit position when the shift is applied. This behavior is common in PC-based compilers, where operands are numeric types rather than bit patterns.

To avoid problems with inconsistent treatment of high-order bits, it is recommended that the modified high order bits be considered indeterminate after a right shift, and either forced to known values or ignored.

If the number of positions to shift is negative or larger than 31, the shift is treated as an out-of-range condition. The result will be the same as if the initial bit pattern were shifted a very large number of positions. For the case of a right shift, this has the effect of setting all of the bits in the bit pattern to match the initial high-order bit. For all other cases the result is a zero bit pattern.

Negation

There is one negation operator, a minus sign preceeding an operand or subexpression. The result of negating a fixed point or bitwise expression is a fixed point value. The result of negating a floating point expression is a floating point value.

negation	-

Examples:

```
P1 = -P2
NEGSUM = -(A+B)
```

Grouping operators

Grouping can be used to control order of evaluation or just to make the evaluation sequence more clear. Terms inside of the grouping consist of expression operators and operands, and for this reason can be called *subexpressions*. Subexpressions can include nested groupings, but this nesting is restricted to 10 levels. There is no run-time speed penalty for grouping terms. The enclosing parentheses always occur in pairs, and if any open parenthesis is not balanced by a close parenthesis, the expression is not valid.

begin subexpression	(
end subexpression	)

Example:

```
POUT  =  (P1 + P2) * (P3 | P4)
```

A subexpression has the value and data type of the result computed inside the grouping. Because a subexpression has a value, a negation operator can be applied to a subexpression group.

Operator Precedence

Operator precedence determines the order in which operators are applied to operands. If operators have the same precedence, the operations are performed from left to right. However, if operators are at different precedence, the operators with higher precedence are performed first. The levels of precedence, from highest to lowest, are:

1. Evaluation of primitive operand terms and subexpressions
2. Negation
3. Multiplication and division
4. Addition and subtraction.
5. Bitwise operations and shifts

These same rules of precedence are applied (recursively) for evaluating terms bracketed by sub-expression parentheses.

Example 1:

```
P1 = P2+P3*P4|P5
```

The multiply operation has the highest precedence, so the intermediate result P3*P4 is computed first. The addition operator has next-highest precedence, so P2 is added to the intermediate result as the second operation. Bitwise-or has lowest precedence, and is performed last to yield a bitwise result.

Example 2:

```
P1 = P2 * -(P3+P4)
```

The subexpression is evaluated at highest precedence. The intermediate sum is then negated, the negation operation having higher precedence than the multiply operation.

Example 3:

```
P3 =  P1 & $01   +   P2 & $02
```

It is unlikely that this command will compute the desired result. Because the addition operation has higher precedence and is performed first, the expression is equivalent to the following, with bitwise operations performed left to right:

```
(P1)   &   ($01+P2)   &   ($02)
```

Example 4:

```
POUT  =  P1<<2   ^   P1>>2
```

It is unlikely that this command will compute the desired result. Because shift and other bitwise operations are equal precedence, the shift and exclusive-or operators are applied left to right. The command is equivalent to the following:

```
POUT  =  ((P1<<2) ^ P1)   >> 2
```

Buffering During Expression Evaluation

A DAPL expression defines a task in a processing procedure configuration. Because DAPL expressions operate on streams of data like other processing tasks, they are subject to the same tradeoffs between rapid response and efficient throughput. To respond to events as quickly as possible, it is necessary to push each value through the evaluation process just as soon as it appears. But doing this requires extra computing overhead. To evaluate large volumes of data quickly, it is better to collect data into buffers and process blocks of data rather than single values. But some amount of time delay occurs while the data accumulates to fill the buffers, leading to a delay in real-time response. There is a trade-off between response delay and processing efficiency.

As it starts running, each DAPL expression task will examine the current setting of the system `BUFFERING` option (see the **`OPTIONS`** command) to determine whether to buffer the data or to try to use the data immediately. With the option `BUFFERING=OFF`, the expression evaluator task pushes individual values through the sequence without buffering. With the option `BUFFERING=MEDIUM` or `BUFFERING=LARGE`, the expression evaluator sets up buffering storage and performs evaluation operations on data blocks.

With buffered data, the data arrival patterns in data pipes can be unpredictable, so the position in a data stream where a change in a variable takes effect can also be unpredictable.

- If there is a backlog of data in memory, the value of the variable could be more current than the data being processed. If the data are plotted as a function of sampling time, a variable value change could seem to appear at an implausibly early time. The plot, of course, doesn't show that the processing of the data was delayed, causing the data samples and the variable values to be "out of sync."
- When a variable value is combined with a data stream that arrives early, this can produce intermediate results that take effect when other data arrive somewhat later. The illusion is that the variable value changed late, or perhaps not at all. In

fact, there was just a full buffer of intermediate values that already included the old variable value.

Data Extraction

The data representation for fixed point, bitwise, and floating point data internal to DAPL expressions is very general. The final operation of the expression is to "extract" the results, and "convert" into an accessible form. There is no guarantee that the computed results will fit naturally into the target specified as the destination for the computed result. The expression task will do the best that it can to represent the final result accurately.

If the target of the expression is a variable, only one value can be stored. The value selected for storage is the last one to be computed, the "most current" value.

For both variables and pipes, the data conversion depends upon the data type of the computed result and the data type of the target structure.

- If the target location stores word data, the value of the result expression is reduced to a 16-bit quantity. Arithmetic values that are too large or small are bounded at the appropriate limits of their range. Floating point values are rounded to the nearest integer. Bitwise results drop the high order bits without changing the values of the remaining low-order bits.
- If the target location stores long data, the treatment is the same as word data except that the range limits are much larger.
- If the target location is float data type, a fixed-point value is rounded to the closest numeric value that the floating point notation can represent. The lowest 24 bits of a bitwise result are retained.
- If the target location is a double data type, the treatment is the same as for float type except that approximations or truncation are not necessary.

Other Notes on Expressions

Floating point faults (division by zero, overflow to infinity, etc.) result in IEEE "special numbers" such as NAN and +INF without generating a floating point exception.

With older models of Data Acquisition Processors and past versions of DAPL, shift operations were recommended as a subsitute for multiplication and division when the multipliers or divisors were powers of two. The advantages are less clear with CPU devices available on new generations of Data Acquisition Processors. In general, we recommend that you avoid clever programming tricks.

DAPL fixed point expressions will “saturate” in the event of fixed point overflow conditions. For example, if there is an attempt to add 1,000 to the value 2,147,483,640, the overflow will be detected and the reported result will be the maximum representable positive number, 2,147,483,647.

Be careful of multiplying sequences of large numbers in fixed point. Arithmetic operations that attempt to increase or decrease values too much can reach the saturation limits, and might not have the expected effect.

Fixed point division by zero in a DAPL expression is treated as a limiting case of division by a very small positive number. If the dividend was positive, the result is saturated to a maximum positive value. If the dividend was negative, the result is saturated to the largest negative number.

There is no complementation operator for bitwise data. To invert bits, use the exclusive-or bitwise operator. For example, to complement the low order four bits:

```
P2 = P1 ^ $000F
```

8. Voltages and Number Representations

The analog and digital inputs and outputs of the Data Acquisition Processor are voltages. The Data Acquisition Processor converts voltages to integers, performs computations on the resulting integers, and converts integers to voltages. This chapter explains how voltages and integers are related.

Analog Input Voltages

The input voltage for each analog input pin is buffered, amplified by the pin's gain factor, and fed to the analog-to-digital converter. The gain factor is between 1 and 500 and is specified independently for each input channel pipe. Each time the Data Acquisition Processor samples a pin, the analog-to-digital converter returns an integer between -32768 and +32767. The range of valid input voltages depends on the pin's gain and on the analog-to-digital converter range. Note that different Data Acquisition Processors have different ranges. Possible ranges with unity gain are 0 volts to +5 volts, -2.5 volts to +2.5 volts, -5 volts to +5 volts, and -10 volts to +10 volts. Voltage ranges that start at zero volts and go up to some positive voltage are called unipolar ranges while voltage ranges that span both negative and positive voltages are called bipolar ranges.

The Data Acquisition Processor Hardware Manual explains how to configure the voltage range of the Data Acquisition Processor.

Digital Input Voltages

Digital inputs are voltage signals with only two significant levels, low (0) and high (1). Data Acquisition Processor inputs follow the standard TTL specification that any voltage between ground and 0.8 volts is low, and any voltage between 2.0 volts and the supply voltage, approximately 5.0 volts, is high. Voltages between 0.8 volts and 2.0 volts are regarded as transition voltages, and may be sensed as either low or high.

Digital signals must originate from TTL-compatible components. Voltages applied to digital input pins must be between ground and the supply voltage, approximately 5.0 volts. The digital input latches may be damaged if this precaution is not observed.

There are pull-up resistors on all digital inputs. Because of these pull-up resistors, unused inputs appear as 1's.

Interpreting Integers as Analog Voltages

The readings that the Data Acquisition Processor receives from the analog-to-digital converter or sends to a digital-to-analog converter are referred to as conversion values. All conversion values are scaled so that a reading of zero represents zero volts, a reading of 32768 represents positive full scale, and a reading of -32768 represents negative full scale. For unipolar inputs, negative values are not used, so the lower range limit is 0.

The following gives a formula relating voltage, conversion value, and full scale. Let `X` represent a conversion value, and let `F` represent the full scale voltage — 2.5 volts, 5 volts, or 10 volts. The voltage `V` corresponding to the conversion value `X` is given by:

```
V = (X/32768) * F
```

For Data Acquisition Processors on bipolar ranges, input and output voltages can range from negative full scale to slightly below positive full scale. For Data Acquisition Processors on unipolar ranges, input and output voltages can range from 0 to slightly below positive full scale. The possible readings range from

- -32768 to +32752 for 12-bit Data Acquisition Processors on bipolar ranges
- 0 to +32760 for 12-bit Data Acquisition Processors on unipolar ranges
- -32768 to +32764 for 14-bit Data Acquisition Processors on bipolar ranges
- 0 to +32766 for 14-bit Data Acquisition Processors on unipolar ranges
- -32768 to +32767 for 16-bit Data Acquisition Processors on bipolar ranges

Only bipolar ranges are offered on 16-bit Data Acquisition Processors.

Binary Representation

Within the Data Acquisition Processor, conversion values are represented by 16-bit signed binary numbers. These are numbers of the form

```
xxxx xxxx xxxx xxxx
```

where each “x” represents a “0” or a “1”. The highest order (leftmost) bit always represents the sign. A number is positive if its highest order bit is 0 and negative if its highest order bit is 1.

12-bit Data Acquisition Processors

The 12-bit Data Acquisition Processors have 12-bit analog-to-digital converters and 12-bit digital-to-analog converters. In order to represent a 12-bit value as a 16-bit number, some of the bits are set to 0. For the bipolar ranges, the four rightmost bits are set to 0, so the conversion values are in the form

```
xxxx xxxx xxxx 0000
```

Possible conversion values range from -32768 to 32752. All are divisible by 16, and the increment between possible conversion values is 16. In hexadecimal form, the conversion values range from 8000 to 7FF0.

For the unipolar ranges, the leftmost bit is set to 0 so that the conversion values always appear as positive binary numbers. The conversion values then take the form

```
0xxx xxxx xxxx x000
```

Possible conversion values range from 0 to 32760. All are divisible by 8, and the increment between possible conversion values is 8. In hexadecimal form, the conversion values range from 0000 to 7FF8.

14-bit Data Acquisition Processors

The 14-bit Data Acquisition Processors have 14-bit analog-to-digital converters and 14-bit digital-to-analog converters. In order to represent a 14-bit value as a 16-bit number, some of the bits are set to 0. For the bipolar ranges, the two rightmost bits are set to 0, so the conversion values are in the form

```
xxxx xxxx xxxx xx00
```

Possible conversion values range from -32768 to 32764. All are divisible by 4, and the increment between possible conversion values is 4. In hexadecimal form, the conversion values range from 8000 to 7FFC.

For the unipolar ranges, the leftmost bit is set to 0 so that the conversion values always appear as positive binary numbers. The conversion values then take the form

```
0xxx xxxx xxxx xxx0
```

Possible conversion values range from 0 to 32766. All are divisible by 2, and the increment between possible conversion values is 2. In hexadecimal form, the conversion values range from 0000 to 7FFE.

16-bit Data Acquisition Processors

The 16-bit Data Acquisition Processors have 16-bit analog-to-digital converters and 16-bit digital-to-analog converters. All bits in the 16-bit binary representation are significant.

All Data Acquisition Processors are calibrated so 0000 0000 0000 0000 represents zero volts in all ranges. For the 12-bit Data Acquisition Processors, there are 4096 possible conversion values for each scale, so the highest conversion value is 4095 steps above the lowest. For the 16-bit Data Acquisition Processors, there are 65,536 possible conversion values for each scale, so the highest conversion value is 65,535 steps above the lowest. As a consequence, the highest conversion value for each scale is one step below the nominal full scale value.

A DAPL expression or a **SCALE** command can be used to scale the digitized values. The **FORMAT** command has an option to print values with decimal points. Together, these commands allow the Data Acquisition Processor to send data directly into programs that require data in engineering units. See the Applications Manual for an example of conversion from integers to engineering units.

Interpreting Readings as Binary Fractions

Conversion values can be considered as signed binary fractions by placing an implicit binary point after the leftmost bit of each value. This is equivalent to dividing each reading by 32768. In this representation, each reading is just a fraction of full scale. The input voltage V is the binary fraction reading multiplied by the full-scale voltage.

As shown here, the bit indicated by "`s`" is 0 for a positive sample or 1 for a negative sample. The bits indicated by "`x`" can take values 0 or 1. The bits indicated by "`0`" are

fixed at zero. The period represents the implied binary point. For unipolar voltages, the high order bit is always zero, indicating a positive value.

12-bit Data Acquisition Processors

As binary fractions, the readings for 12-bit bipolar voltages take the form

```
s.xxx xxxx xxxx 0000
```

Possible numbers in this form range from -1 to 32752/32768 = 0.9995. In a hexadecimal base, the values range from -$1.0000 to +$0.FFE0 and are represented as integers $8000 to $7FF0.

As binary fractions, the readings for 12-bit unipolar voltages take the form

```
0.xxx xxxx xxxx x000
```

Possible numbers in this form range from 0 to 32760/32768 = 0.9998. In a hexadecimal base, the values range from +$0.0000 to +$0.FFF0 and are represented as integers $0000 to $7FF8.

14-bit Data Acquisition Processors

As binary fractions, the readings for 14-bit bipolar voltages take the form

```
s.xxx xxxx xxxx xx00
```

Possible numbers in this form range from -1 to 32764/32768 = 0.99988. In a hexadecimal base, the values range from -$1.0000 to +$0.FFF8 and are represented as integers $8000 to $7FFC.

As binary fractions, the readings for 14-bit unipolar voltages take the form

```
0.xxx xxxx xxxx xxx0
```

Possible numbers in this form range from 0 to 32766/32768 = 0.99994. In a hexadecimal base, the values range from +$0.0000 to +$0.FFFC and are represented as integers $0000 to $7FFE.

16-bit Data Acquisition Processors

As binary fractions, the readings for 16-bit bipolar voltages take the form

```
s.xxx xxxx xxxx xxxx
```

Possible numbers in this form range from -1 to 32767/32768 = 0.99997. In a hexadecimal base, the values range from -$1.0000 to +$0.FFFE and are represented as integers $8000 to $7FFF.

Digital Readings

The digital input port of the Data Acquisition Processor has 16 bits. The pins of the digital input port always are read simultaneously by the Data Acquisition Processor. The resulting value may be interpreted either as forming one 16-bit digital input, or as 16 binary inputs.

A 16-bit binary value from the digital input port appears in the form

```
xxxx xxxx xxxx xxxx
```

This may be treated as an integer between -32768 and +32767. Note that the highest order bit determines the sign. If the highest order bit is 0 the integer is positive, and if the highest order bit is 1 the integer is negative.

Integers Used by DAPL

Most DAPL tasks deal with integer values in the range from -32768 to +32767. Thus the internal representation follows the format defined by the analog-to-digital converter. Integers in the range from -32768 to +32767 are called word integers. Each integer in this representation occupies two bytes of the Data Acquisition Processor buffer memory.

The range from -32768 to +32767 is too restrictive for some computations in DAPL, so DAPL has a “long integer” data type. Long integers range from -2,147,483,648 to +2,147,483,647. Each long integer occupies four bytes of the Data Acquisition Processor buffer memory. When there is a choice between word integers and long integers, use word integers unless extra precision is needed.

DAPL applications related to the fast Fourier transform also use complex integers. Each complex integer is represented implicitly by two ordinary integers.

Floating Point Types

The processor on the Data Acquisition Processor and the DAPL operating system provide support for floating point computations consistent with IEEE Std 754. Floating point is available for internal processing tasks. The standard "single real" data type is called `FLOAT` in the DAPL system. Each `FLOAT` value occupies four bytes of the Data Acquisition Processor buffer memory. The "double real" data type is called `DOUBLE` in the DAPL system. Each `DOUBLE` value occupies eight bytes of the Data Acquisition Processor buffer memory. The standard "extended real" type is not supported as a DAPL data storage type. Standard math library functions are supported to the precision of a `DOUBLE` data type. Where floating point facilities are not provided directly by the CPU hardware, the DAPL system includes floating point emulation. The emulation, because it consists of complex sequences of integer processing unit instructions, is slower than the hardware floating point unit (FPU) by a factor of roughly 500, so as a practical matter, most applications that need floating point computations also need Data Acquisition Processor models that provide a hardware FPU.

Conversions Between Integer Types

Sometimes a computation with integer values requires a higher precision format. Word integer values can be converted to a long integer format. To make this conversion, the original number value is stored in the lower 16-bit positions, and the bit in the sign bit position is replicated to fill the higher 16-bit positions. This process is called "sign extension." Most sign extension conversions are automatic. For example, if the value of a word data pipe is placed into a long data pipe by a DAPL expression, sign extension is applied.

A long integer value can be reduced in precision without affecting the value if the 16 extension bits all match the sign bit position of the lower two bytes. A long integer value not in the form of a sign-extended 16-bit value is beyond the representable range of a word integer. If a long integer number that is not word representable is assigned to a word integer element, the value actually stored will be the largest positive or negative number representable by the word integer, according to the sign of the original 32-bit value. This range-limiting operation is called "saturation." For example, if the **SCALE** command computes an intermediate long value that exceeds the range of a word output data pipe, the output value is saturated at a range limit for a word output pipe.

Hexadecimal Notations and Integers

The DAPL system provides a hexadecimal notation for access to individual bits in a binary number representation. This feature is particularly useful for encoding or decoding information that is transferred through the digital data ports. The hexadecimal numbers $0000 through $FFFF can be used to represent word integer values. The hexadecimal numbers $00000000 through $FFFFFFFF can be used to represent long integer values.

Hexadecimal number do not have a natural representation for negative numbers, but data stored in the DAPL system have a signed interpretation. Whether the bit pattern is interpreted as positive or negative depends on the value of the bit in the sign bit position. If the bits are numbered so that the last, lowest order bit is numbered as bit 0, then word values have a sign bit in bit position 15, and long values have a sign bit in bit position 31. If a hexadecimal notation does not specify some of the higher-order bit positions, those bit positions default to zeroes.

Problems can arise because the sign bit position depends on the number of bits in the representation. That means, hexadecimal notations are not necessarily unique, and must be used with care. For example, the hexadecimal notation $FFF0 is equal to 65520 when it is used to initialize a 32-bit constant value, because only 16 of the 32 bits are specified, and the sign bit position is among the 16 higher-order bits that default to zero. But when used to assign a value to a word constant, this same $FFF0 notation yields a value of -16, because the leading bit is a 1, giving a negative interpretation.

For DAPL expressions, all explicit integer constants are presumed to be 32-bit values, and this applies to hexadecimal as well as integer notations. Consequently, eight hexadecimal digits must be specified to represent a negative number in DAPL expressions using a hexadecimal notation. For example, if the notation $FFF0 is intended to represent a value -16, this notation must be sign extended to $FFFFFFF0 in the DAPL expression command line.

For DAPL system commands that define scalar values, the sign bit position for integers is determined by the data type specified. For the case of long integer type, the interpretation of hex notations is exactly the same as in DAPL expressions. For the case of word integer type, only hex notations $0000 through $FFFF are meaningful. Any bits beyond the first 16 cannot be stored, consequently, any additional nonzero bits are diagnosed as an out-of-range condition. For example, $0FFF0 would be considered the same as $FFF0, with the leading zero digit not significant, but $FFFF0 is a range error.

For task parameters, the hex notation is interpreted differently because there is no prior information about whether the constant value should be considered a word or long type. If the hexadecimal notation contains four or fewer hex digits, it is presumed that the notation is intended to define a word value, and the notation is interpreted in the manner of a 16-bit word value. However, if the hexadecimal notation contains five or more hex digits, it is presumed that the notation is intended to define a long value, and the notation is interpreted in the manner of a 32 bit value. Consequently, $FFF0 would be interpreted as a word value -16, while $0FFF0 would be interpreted as a long value +65520.

9. Data Transfer

Communication pipes (com pipes) are first-in-first-out buffers for communication between a Data Acquisition Processor and its host PC. Com pipes allow text or binary communication, and transfer data on the PC bus.

This chapter summarizes the communication pipes and the tasks through which DAPL sends and receives data.

Standard Com Pipes

The default configuration for a Data Acquisition Processor has four standard com pipes: `$SYSIN`, `$SYSOUT`, `$BININ`, and `$BINOUT`. `$SYSIN` and `$SYSOUT` are text com pipes for reading data from and writing data to the PC. All commands to the DAPL interpreter are read from `$SYSIN`; all status and error messages are sent to `$SYSOUT`.

`$BININ` and `$BINOUT` are binary com pipes for reading data from and writing data to the PC.

Sending Text to the PC

The **`PRINT`** and **`FORMAT`** commands are used for sending text data to the PC. Both commands format data into ASCII characters before sending the data to the PC. A **`PRINT`** task sends all raw data from the input channel pipes to the PC without any processing. A **`FORMAT`** task sends computed data to the PC. A **`FORMAT`** task can send data from pipes or variables, and also can send line counts, constants, and strings.

If several **`FORMAT`** tasks are active at one time, each **`FORMAT`** task can send a different identifying string. This lets programs in the PC determine which **`FORMAT`** task sent each line of data.

Sending Binary Data to the PC

Most DAPL commands can send binary data to the PC. The simplest way to send binary data to the PC is with the **BPRINT** command. **BPRINT** sends all raw data from the input channel pipes directly to the PC.

Binary data also can be sent to the PC by using $BINOUT as the output pipe of any task. Because binary data have no implicit identifying symbols, in most applications only one task can write to $BINOUT.

If data streams from several pipes are to be sent to the PC, it is necessary to merge the data streams. DAPL provides several merging commands for this purpose. If data enter several pipes at the same rate, a **MERGE** task with output pipe $BINOUT can be used to combine the data. If data enter the pipes at different rates, the command **MERGEF** can be used. A **MERGEF** task adds an identifying flag to each binary value. While this doubles the communication overhead, it allows a program in the PC to identify the source of each binary value. If data enter the pipes at different but proportional rates to each other, the command **NMERGE** can be used.

The commands **BMERGE** and **BMERGEF** are more efficient blocked versions of **MERGE** and **MERGEF**. A **BMERGE** or **BMERGEF** task reads blocks of data from each of several pipes and writes the blocks to its output pipe. The output pipe usually is $BINOUT. **BMERGEF** adds an identifying flag to each block of data. **BMERGEF** appends one flag per block, so it is more efficient than **MERGEF**.

Reading Text from the PC

The DAPL command interpreter receives all data sent to a Data Acquisition Processor through `$SYSIN`. This com pipe also can be used to put data into pipes with the **`FILL`** command, or to set the values of variables with the **`LET`** command.

Reading Binary Data from the PC

Binary data can be sent from the PC to the Data Acquisition Processor through the com pipe `$BININ.` If the PC sends multiplexed data, a **`SEPARATE`** task can be used to break out the data streams in the Data Acquisition Processor. If each data stream has a different rate, the PC can append flags to the data sent to the Data Acquisition Processor and use **`SEPARATEF`** task to break out the data streams.

Additional Com Pipes

In addition to the default com pipes `$SYSIN`, `$SYSOUT`, `$BININ`, and `$BINOUT`, other com pipes can be defined. This allows the Data Acquisition Processor to have several binary and text communication pipes transmitting and receiving simultaneously. The additional pipes are configured in the DAPcell control panel application. It constructs the required communication channels and configures them automatically. Communication pipes can also be configured by software applications, using features of the DAPIO32 programming interface.

10. Processor and Memory Allocation

This chapter describes how the DAPL operating system manages shared CPU and memory resources.

Multitasking

As a multitasking operating system, DAPL is responsible for allocation of processor resources among all the tasks that are active at any time. DAPL is responsible for giving priority to certain critical tasks that are necessary for error-free data acquisition. Once reliable system operation is assured, processing tasks defined by task definition commands are allowed to perform all other required processing.

The DAPL operating system is responsible for switching repeatedly among tasks. This means that DAPL maintains a run-time environment for each active task. When a task is active and is running, the environment is found in the processor's registers and stack. When a task is active, but is not running, the run-time environment is saved in memory. When DAPL switches between tasks, one task's environment is saved to memory and another task's environment is recalled from memory.

DAPL switches between tasks for either of two reasons:

1. a task uses up its maximum CPU time allocation, or
2. a task has no data to process and explicitly requests a task switch.

Under the default options assumed when the DAPL system is initialized, the maximum time allocation allowed per task is typically 2000 µs. This setting can be adjusted using the `QUANTUM` option. A setting of 200 µs is typical when low response latency is required.

When DAPL switches between tasks, the operating system is responsible for selecting the task to activate. Under the `SCHEDULING=FIXED` option, DAPL uses a round robin scheduling algorithm. Under the `SCHEDULING=ADAPTIVE` option, DAPL uses an adaptive algorithm that schedules some tasks more often and some tasks less often, as required by the actual data flow. In the limiting case, a task that does not require much time is scheduled one fifth as often as a task that requires a lot of CPU time.

Because task scheduling depends on the unpredictable arrival of data, there is no guaranteed relationship between the order in which the DAPL command interpreter processes task definition commands and the order in which DAPL schedules the

commands. This should be considered in interpreting the output of a DAPL application.

Note: See **OPTIONS** in the command reference for information on setting scheduling options.

For information about optimizing processor performance see Chapter 11.

Interleaving of Output

When several tasks share an output device (usually a text or binary com pipe), the multitasking nature of the DAPL operating system becomes apparent. An example of this is an application that has several **FORMAT** tasks active at one time. In this sort of application, it normally happens that one **FORMAT** statement sends several lines of data before using up its time allocation, and then another **FORMAT** statement sends several lines of data. There is no way to predict the order in which the Data Acquisition Processor sends its data. In order to distinguish the data from different **FORMAT** tasks, each **FORMAT** task can send a string to the PC as part of each line. This allows a program in the PC to determine the source of the data for each line of text.

Note: A **FORMAT** task sends only full lines; DAPL prevents interleaving parts of two or more lines of output.

When sending binary data from a Data Acquisition Processor to the PC, an identifying number must be used rather than an identifying text string. **MERGE**, **MERGEF**, **BMERGE**, and **BMERGEF** allow interleaving of binary data from several sources without loss of information.

Memory Allocation

The DAPL operating system has efficient algorithms for allocating RAM memory for all of the data structures in DAPL. Some structures are allocated once during their definition phase and de-allocated when their life cycles end. Others grow and shrink dynamically as required by the flow of data. A memory allocation fails only when there is no contiguous memory in the RAM to satisfy the request.

16-bit Custom Command Stack Memory Allocation

16-bit custom commands created with older versions of the Developer's Toolkit for DAPL require space in a special region of memory called heap memory. The size of heap memory expands dynamically, up to a maximum of 64K bytes. The actual amount of available heap memory may vary among different Data Acquisition Processor models and among different DAPL versions. To determine the exact size, run the command **DISPLAY** HMEM.

A typical custom command requires about 1.5K bytes of stack memory. The size of heap memory available on any Data Acquisition Processor model is sufficient for all but the largest applications. If the heap memory capacity is exceeded, the following error message is printed:

```
*** Error 2222: out of 16-bit custom command stack memory
```

This typically occurs if a DAPL application has more than 30 custom commands running simultaneously.

The best way to eliminate heap memory problems is to convert the custom command to 32-bit custom module form. 32-bit tasks are not subject to stack or dynamic memory size limitations.

11. Optimizing Processor Performance

This chapter provides suggestions for getting the maximum performance from the Data Acquisition Processor.

Reducing Processor Load

DAPL 2000-compatible Data Acquisition Processors have high-speed processing power, optimized software, and a large RAM buffer. In some applications, the Data Acquisition Processor is pushed to its limits; in others the Data Acquisition Processor spends most of its time waiting for data. Dynamic performance of the Data Acquisition Processor depends upon the input data stream, the number of tasks, the types of tasks, communications overhead, CPU clock speed, host PC speed, and other factors.

The Data Acquisition Processor loses a small amount of time each time it moves a value into a pipe or takes a value out of a pipe. Operations that process large amounts of data take more time than operations that process small amounts of data. If large amounts of data are reduced to smaller amounts early, that saves processing power. If an application requires averaging, for example, it is best to perform the averaging on the raw data so that further processing is performed on a reduced number of points.

In most cases it is faster to process and reduce data in the Data Acquisition Processor than it is to transmit the data to the PC and then process and reduce the data in the PC.

Digital Signal Processing

Digital signal processing is possible with all Data Acquisition Processor models, but it is necessary to take into account the limitations of the processor. **`FIRFILTER`**, **`FFT`**, and other digital signal processing tasks are highly optimized, but are computationally intensive. To speed these tasks, try to limit the number of computations that the Data Acquisition Processor must perform. When using a low pass digital filter, for example, consider computing averages of small blocks before applying the filter. Averaging reduces the number of taps required, as well as reducing the number of data points. When computing FFTs, try to reduce the number of points required. If the data are to be plotted on the PC's screen, for example, it usually is possible to limit the size of the transform according to the screen resolution.

Communication Formats

To speed communication from the Data Acquisition Processor to the PC, use binary format instead of ASCII format. This eliminates the time consuming format conversion from binary to ASCII. Binary format also requires transmission of fewer bytes from the Data Acquisition Processor to the PC.

Channel Pipe Efficiency

The configuration of input and output channel pipes affects the efficiency of tasks that read and write channel pipe data. The following list shows possible channel pipe configurations in decreasing order of efficiency:

1. task with a channel pipe list of all channel pipes
2. task with a single channel pipe
3. task with a channel pipe list of some, but not all, channel pipes

Scheduling Options

The scheduling control options `BUFFERING`, `SCHEDULING`, and `QUANTUM` provide additional means for speed optimization. Most tasks are more efficient when they work on larger blocks of data, and when they are allowed to run to completion rather than being interrupted frequently. The default values of the scheduling control parameters provide moderate sized memory buffers and a moderate sized scheduling quantum appropriate for moderate buffer sizes. See the **`OPTIONS`** command reference for more information.

Note The scheduling options for most efficient processing are not necessarily the best to obtain a quick real-time response. See Chapter 13 for more details.

Streaming Data to the PC

Some applications require high-speed data acquisition without real-time data processing. The **BPRINT** and **COPY** commands stream raw data to the PC in binary format. Microstar Laboratories DAPlog Plus program logs data to disk or RAM disk at high speed in real-time.

Trigger Performance

Tasks that involve triggers are especially sensitive to their input data streams. A **WAIT** task, for example, throws out data until a trigger occurs; then it transfers a block of data from one pipe to another. Discarding blocks of unwanted data is very efficient, but identifying and moving data blocks when a trigger is asserted takes more computing overhead than an ordinary data transfer. If trigger events are frequent, it is more efficient to have a processing task that analyzes a continuous data stream, rather than using a trigger. On the other hand, if trigger events occur rarely (for example, less than once per 100 samples), triggering is typically more efficient.

High-Speed Triggering

The Data Acquisition Processor allows hardware triggering up to the maximum input sampling rate. By using a **COUNT** command with hardware triggering, it is possible to acquire large blocks of data without risk of overflow.

Software triggering is more flexible than hardware triggering and also allows capture of pretrigger data. Software triggering can perform up to the maximum speed of the Data Acquisition Processor, but limits the amount of CPU capacity available for other processing. When using software triggering at high data rates, the task that asserts the trigger may need to process only a part of the raw data, and can skip unnecessary processing to save time. This is true even though the **WAIT** command can wait for data at the maximum speed of the Data Acquisition Processor.

The following command list illustrates a typical high-speed triggering application. In this application, the Data Acquisition Processor samples one pin at 200,000 samples/second, and transfers a block of data, including pretrigger data, each time the input signal passes through a specified region. The **WAIT** task can process data at the maximum speed of the Data Acquisition Processor, but the **LIMIT** task is limited to a slower rate. To increase the performance of the Data Acquisition Processor, the input signal is read into six input channel pipes. The **LIMIT** task is configured to read just one of the input channel pipes, so it processes one sixth of the raw data. The **WAIT** reads data from a list of input channel pipes, so it processes all the raw data.

```
TRIGGER T
PIPE P1
IDEF A 6
  SET IPIPE0 D0
  SET IPIPE1 D0
  SET IPIPE2 D0
  SET IPIPE3 D0
  SET IPIPE4 D0
  SET IPIPE5 D0
  TIME 5
  END
PDEF B
  LIMIT (IPIPE0,INSIDE,100,200,T)
  WAIT  (IPIPE(0,1,2,3,4,5),T,100,100,P1)
  END
```

Note: In this example, the trigger event must span at least six sample times to guarantee recognition.

Benchmarking an Application

For fast applications, it is useful to perform a benchmark to find how much processing capability is available. Benchmarks can determine if there is a comfortable overhead of processing power for the application, or if the input rate is too high for the processing being done. Benchmarks also aid in determining how to optimize an application.

12. Overflow and Underflow

When sampling occurs faster than data can be processed and transmitted to the host, buffer overflow eventually must occur. When this happens, the Data Acquisition Processor halts gracefully. Sampling stops without loss of data, and the Data Acquisition Processor continues processing the valid acquired data.

Even at the highest speed, the Data Acquisition Processor correctly buffers the acquired data in its onboard memory. An overflow of buffer memory never results in any loss or corruption of buffered data.

Overflow can be prevented by specifying a **COUNT** option, using a hardware trigger, reducing the sampling rate, or making processing tasks more efficient.

When overflow occurs, input sampling stops. The Data Acquisition Processor continues normal processing of the data stored in memory.

Overflow usually occurs at the input channel pipes. If the Data Acquisition Processor is unable to remove data from input channel pipes fast enough, DAPL expands the input channel pipes. If there continues to be too much data to remove from the input channel pipes, the pipes expand until they fill buffer memory.

DAPL automatically discards data from input channel pipes that do not have any tasks reading from them. Unused input channel pipes usually don't contribute to memory overflow.

Overflow Messages

The Data Acquisition Processor determines the sample count at which overflow occurs. The Data Acquisition Processor response to overflow is determined by the `OVERFLOWQ` option. If this option is set to `OFF`, the following overflow message is sent to the host PC:

```
*** Warning 1530: channel pipe overflow at sample #xxxxx
```

If the `OVERFLOWQ` option is set to `ON`, the overflow message is suppressed, so that it is not inserted into the data stream. At any time, the PC can send the command:

```
DISPLAY OVERFLOWQ
```

The Data Acquisition Processor responds by sending a line containing a single 32-bit integer. If this number is zero, overflow has not occurred. Otherwise, the number indicates the sample number at which overflow occurred.

The sample number in the overflow message is the analog-to-digital converter's sample count at overflow. If overflow occurs after sampling each pin of a six channel pipe input procedure 100,000 times, for example, the overflow sample number is 600,000.

Preventing Overflow

The first step in preventing overflow is to determine where data values are backing up. This can be accomplished by sampling until just before overflow occurs and issuing a **DISPLAY PIPES** command. This command prints the names of all the user-defined pipes, as well as the number of data values in each pipe. Check this list for pipes that contain large amounts of data — typically over 1000 values.

If no user-defined pipe contains excessive data, overflow is occurring in the input channel pipes. This means that the tasks reading from input channel pipes cannot keep up with the sampling rate. Solutions include reducing the sampling rate, using **SKIP** to reduce the amount of data processed, using a hardware or software trigger to extract relevant data, or performing more efficient processing in processing procedures.

If a pipe contains many data values, data backup probably is occurring at the task that reads from the pipe. To prevent buffer overflow from a task, it is best to do as much data reduction as possible in the tasks that read from input channel pipes and to minimize the number of times data are moved in and out of pipes.

Another overflow analysis strategy is to use the **STATISTICS** command to determine which tasks are using large amounts of processor time.

The **COUNT** command can prevent overflow by specifying the number of values to be sampled. Input sampling stops when the sample count is satisfied. Note that overflow still can occur if the count is larger than available memory. In this case it is necessary to test the application to verify that tasks are able to process the extra input data before memory overflow.

In rare cases, channel pipes that have no tasks reading from them can contribute to channel pipe overflow. Data in these unused channel pipes are temporarily stored in memory. The data from unused channel pipes are discarded as data from used channel pipes are read. If processing is delayed, unneeded values use up part of the available storage capacity.

Underflow Messages

When output procedure updates occur faster than the Data Acquisition Processor can place data into output channel pipes, channel pipe underflow eventually must occur. When underflow occurs, output updating stops.

The Data Acquisition Processor response to underflow is determined by the `UNDERFLOWQ` option. If this option is set to `OFF`, the following underflow message is sent to the host PC:

```
*** Warning 1531: channel pipe underflow at sample #xxxxx
```

If the `UNDERFLOWQ` option is set to `ON`, the underflow message is suppressed so that it is not inserted into the data stream to the PC. At any time, the PC can send the command:

```
DISPLAY UNDERFLOWQ
```

The Data Acquisition Processor responds by sending a line containing a single 32-bit integer. If this number is zero, underflow has not occurred. Otherwise, the number indicates the sample number at which underflow occurred.

The sample number in the underflow message is the output procedure's update count at the time of underflow. If underflow occurs after updating each channel pipe of a two-channel pipe output procedure 100,000 times, for example, the underflow sample number is 200,000.

Preventing Underflow

If output channel pipe data are repetitive, a cyclical output procedure should be used. Cyclical output procedures are configured using the **CYCLE** command. Cyclical output procedures never underflow, even at the highest update rate of the Data Acquisition Processor.

The maximum update rate of a noncyclical output procedure is limited by the rate at which values are placed into output channel pipes. This depends on the speed of tasks used to send data to the output channel pipes. In some instances, increasing an output procedure's initial startup delay allows faster update rates. An output procedure's initial startup delay is increased using the **OUTPUTWAIT** command.

Other options are available for avoiding underflow. The `UPDATE BURST` option of an output procedure allows output updating to stop when no data are available and automatically restart when more data are ready. The **COUNT** command also controls underflow—**COUNT** forces an output procedure to stop output updating after a specified number of updates.

13. Low Latency Operation

A Data Acquisition Processor acquires data and processes data concurrently. Because many tasks share one processor, there is a delay from the time an input is sampled until a task can act upon the sample data. This delay is called latency.

There is a tradeoff between throughput efficiency and latency. To optimize the processing of large amounts of data, the most effective strategy is to dedicate as much CPU power as possible to computation, and as little as possible to other system activities such as task scheduling. To optimize the speed at which a task can respond to a real-time event, the most effective processing strategy is to divide tasks into very small time intervals, or scheduling quanta, and switch tasks frequently so that each task has an opportunity to respond. Switching tasks more frequently increases system overhead, reducing the volume of data that can be processed.

The two goals of high throughput and fast response are clearly in conflict. Each application should evaluate whether its goal is fast processing or minimal latency, and select options accordingly. For example, most disk logging applications capture and transfer data at very high speeds, but they are not sensitive to delays of a second or more. In contrast, delays of a couple of milliseconds in a closed loop control application could cause the controlled system to oscillate in a chaotic fashion.

The DAPL operating system provides three system options for controlling the tradeoff between latency and efficiency. These options are: `BUFFERING`, `SCHEDULING` and `QUANTUM`. See the **`OPTIONS`** command for more information on setting system options.

Buffering Control

When optimizing for throughput, each processing task will maintain possession of the CPU for longer periods of time. Because each task runs more efficiently but less often, data will tend to collect into larger blocks. Larger buffers can also be used for moving the data.

On the other hand, when optimizing for fast response, each processing task will maintain possession of the CPU for shorter periods of time, so that fewer samples accumulate.

The `BUFFERING` option specifies the strategy that DAPL should use for moving data into and out of pipes. For most applications, `MEDIUM` is a good choice, providing good efficiency for moving moderate amounts of data into and out of tasks. Some small gains in processing efficiency are available by changing this option to `LARGE`. For systems optimized for fastest response, there is no time for data to accumulate, and `BUFFERING=OFF` can be specified. This informs the DAPL system that it is not necessary to wait for blocks of data to accumulate.

Another consideration is the `MAXSIZE` parameter on the **`PIPES`** command. Pipes will allocate memory as required, up to the specified `MAXSIZE`. This parameter can be set lower for tasks that receive or generate small blocks of data, and larger for tasks that receive or generate large blocks of data.

Task Scheduling Control

The SCHEDULING and QUANTUM options affect the multitasking strategy employed by the DAPL system.

The SCHEDULING option allows applications to control the task switching strategy. DAPL provides two algorithms for task switching: fixed (also known as round-robin) and adaptive.

With a fixed task switching algorithm, DAPL gives each task the same amount of CPU time. In most time-critical applications, this is a small amount of time, but sufficient to generate a required response to a real-time event. A task that does not need the entire time quantum can release unused CPU capacity for other tasks to use, without penalty.

With the adaptive algorithm, DAPL observes the amount of CPU time consumed by each task. Those tasks that use greater amounts of time are scheduled more often. Those that use less time are scheduled less often. The effect of the adaptive strategy is that tasks are given opportunities to execute in proportion to the work that they need to do to maintain data flow through the system. Equalizing the data flow among tasks causes data to accumulate into relatively uniform blocks that can be processed efficiently. While this strategy works well for equalizing data flow, it makes scheduling of any one task less predictable. For example, a task that processes sporadic real-time events will seldom have much data to process, hence it is scheduled less often, and its responses are correspondingly delayed. Adaptive scheduling can also have adverse effects when processing is not steady and continuous.

The QUANTUM option sets the rate at which the CPU switches among tasks without affecting the task scheduling algorithm or the buffering strategy. The quantum can be set to any desired number between 100 µs and 5000 µs. Most real-time applications will set this number small so that no single task can delay the response time too much. In other cases, a real-time application might choose a longer quantum. For example, suppose that a real-time system has tasks to detect sporadic events, select data associated with those events, perform a small FFT transform on the collected data, and generate a real-time response on the basis of each transform result. The transform is most efficient if allowed to run to completion without interruption by other tasks. The QUANTUM option can be adjusted to allow completion of the transform without interruption by task switching. Other tasks have relatively little to do, so they release the CPU quickly regardless of the QUANTUM option.

A careful selection of BUFFERING, SCHEDULING and QUANTUM options allows an application to create an environment that best satisfies both its latency and efficiency requirements.

The following options would allow higher data volumes and shorter sampling intervals at the expense of larger buffering delays.

```
OPTIONS  BUFFERING=LARGE, SCHEDULING=ADAPTIVE
OPTIONS  QUANTUM=4000
```

This is optimized for acquisition applications with moderate processing requirements, aimed primarily at efficient throughput.

The following options would provide shorter real-time delays at the expense of data throughput capacity.

```
OPTIONS  BUFFERING=OFF, SCHEDULING=FIXED, QUANTUM=200
```

This is optimized for individual samples or very small data blocks, guaranteed periodic scheduling of all tasks regardless of CPU activity history, and rapid task switching for quick response.

At power up, the options for the Data Acquisition Processor are set to a default compromise that works well for most applications.

```
OPTIONS  BUFFERING=MEDIUM, SCHEDULING=FIXED, QUANTUM=2000
```

Evaluating Task Latency

The **STATISTICS** command is useful for evaluating expected task latency. This command will show all active tasks in the system and the amount of time they consume. Under a fixed scheduling strategy, an absolute bound can be placed on the scheduling quanta, and thus a bound on the latency of real-time response can be estimated.

For systems with so-called "soft" real time requirements, the absolute bound is not very useful, because actual system latency rarely approaches the computed bound. The **STATISTICS** command will report the CPU utilization of each active task, and the latency of the entire scheduling sequence. This information is useful for estimating expected latency, which is typically much better than the worst-case bound.

Low Latency Commands

Some DAPL commands have special logic that detects the **BUFFERING=OFF** option and selects an appropriate algorithm specialized for that case. The **AVERAGE** and **SKIP** commands fall into this category. Other DAPL commands such as **LCOPY** and **SCAN** guarantee a low-latency behavior under any system options.

Many process control programs take the following form:

1. Read input data from all input channel pipes.
2. Process the data for the set.
3. Wait until another complete set of data is available, then repeat.

SCAN is an efficient command for implementing this sort of control program. **SCAN** normally is used with a input channel pipe list. It forms a block of data from all input channel pipes in the channel pipe list before placing any data in its output pipe. When a channel list is completed, the data block is released immediately for other tasks to process. Since **SCAN** releases a data block as quickly as possible, it is more responsive than a **COPY** command.

Using Custom Modules to Reduce Latency

Some latency is inherent in any multitasking operating system. A task can only respond to an input when the processor is executing the task. Also, there is system overhead in switching among tasks. One way to minimize latency is to reduce the number of tasks active at any time. In critical applications, all of the required processing can be implemented in one custom task. Because the operating system's background tasks do not require much of the processor's time, the custom command will be running most of the time. This approach yields the lowest possible latencies.

14. DAPL Software Triggering

Software triggering is a unique capability of the DAPL operating system. Software triggering allows applications to select data blocks of interest, ignoring other data. In many applications, software triggering can replace complicated electronic triggering circuits. Software triggering can do some things easily that are difficult or impossible to do any other way.

Consider for example a protective relaying application. Opening a circuit breaker to drop a large three-phase motor load can be very expensive, but failing to do so could damage the motor, which is even worse! Phase, timing and sensitivity settings on the protective relay are important, but optimizing these settings requires measurement of actual operating conditions. A Data Acquisition Processor can collect data each time a system disturbance occurs, so that later analysis can verify the proper balance between safety and economical operation. However, some of the data of interest occur *before* relay operation. In other words, by the time a hardware logic signal is available to initiate data collection, it is already too late to capture some of the required measurements.

Software triggering solves this problem by monitoring the data continuously. If nothing happens, and the relay does not activate, extraneous measurements are discarded automatically. However, if the relay does operate, the required measurements are extracted from memory.

The continuous data capture and data management cause some additional processing overhead, but the DAPL operating system is optimized to do these operations with extreme efficiency. Besides, in most applications, there is plenty of capacity in the Data Acquisition Processor's onboard CPU. Why not use it?

Software triggering determines when to trigger by analyzing a data signal. Powerful analysis techniques can be applied, including digital filtering, operating state logic, and region-of-interest selection. The logic can include information from more than one source. Time-sequence analysis can also be performed, to select only a few relevant events for further analysis. Operations such as these are difficult or impossible to do with hardware triggering circuits.

Defining Software Triggers

To use software triggers, three DAPL elements work in combination: a software trigger element to coordinate triggering action, a processing task that generates trigger events, and a processing task that responds to trigger events.

The **TRIGGERS** command defines a software trigger element. For example, the following command might be used to define one trigger that responds to pressure measurements and another trigger that responds to temperature measurements:

```
TRIGGERS    TPRESS, TTEMP
```

After a software trigger is defined, it is available to processing tasks.

Each DAPL processing configuration that uses software triggering has a processing task that generates triggering events and one or more tasks that respond to these events. Most applications use commands built into the DAPL operating system to generate trigger events. Specialized applications can use custom modules for generating trigger events, responding to trigger events, or both.

The DAPL operating system provides these commands that generate trigger events:

- **CHANGE**
- **DLIMIT**
- **LIMIT**
- **LOGIC**
- **PCASSERT**
- **PEAK**
- **TGEN**
- **TOGGLE**

The DAPL operating system provides these commands that respond to software trigger events:

- **INTEGRATE**
- **FREQUENCY**
- **TSTAMP**
- **WAIT**
- **TOGGWT**

There are other tasks that convert one or more streams of trigger events into a new, modified stream. The DAPL operating system provides the following:

- **NTH**
- **TAND**
- **TOR**
- **TRIGSCALE**
- **TCOLLATE**

Three special commands coordinate software triggering applications.

- **TRIGSEND**
- **TRIGRECV**
- **TRIGARM**

Applying Software Triggers

Most applications use software triggering for selecting blocks of data. This is illustrated in the following continuation of the protective relaying example.

Suppose that voltages are monitored on three power phases at 60 Hz. The protective relay has two signals that can be monitored. A high speed 'pickup' signal reports when the relay detection circuits are active. A delayed circuit breaker control signal activated by the relay causes a power circuit breaker to operate. The DAPL input sampling configuration to monitor these five signals might look like the following:

```
IDEFINE  voltages  5
  SET  IP0  s0    // phase1
  SET  IP1  s1    // phase2
  SET  IP2  s2    // phase3
  SET  IP3  s3    // breaker activation signal
  SET  IP4  s4    // relay pickup signal
  TIME   16.75    // about 200 samples / 60 Hz cycle
END
```

The following commands define a trigger and begin the processing configuration that will detect trigger events. The relay pickup and circuit breaker control voltages are measured on a 0 to 5 volt scale, which is digitized in the range 0 to 32767. Pickup is indicated by a voltage reading of 3.0 volts (digitized as 19660) or more, and when this occurs, a trigger event is generated. Subsequent trigger events are allowed after the control voltage reading drops below 1.0 volt (digitized as 6553):

```
TRIGGERS   Tbreak

PDEFINE   capture
LIMIT(IP4,  INSIDE,19660,32767,  Tbreak, \
            INSIDE,6553,32767)
...
```

Capturing voltage and breaker control data starts two power cycles before the relay pickup event. Assume that relay action takes up to five power cycles. At that point, circuit breaker operation begins. Circuit breaker action takes three more power cycles. Following breaker operation, collect two more power cycles of data for completeness. The total is twelve cycles of data to be recorded on four channels, two cycles before, and ten cycles after relay pickup.

For each event and each phase, 200 samples per cycle are recorded. That means, 1600 samples are retained before the relay pickup event, and 8000 samples after. The

following commands continue the processing configuration definition. The **WAIT** command retains the data associated with each event, transferring the data to the PC for storage on a disk drive:

```
WAIT (IP(0,1,2,3), Tbreak, 1600, 8000, $BINOUT)
END
```

How Software Triggering Works

A task that generates trigger events is associated with a data source, usually a stream of data in a pipe. When these data are captured at uniform sampling intervals, as in the case of input channels, there is a direct correspondence between the arrival of samples and the passage of time. Numbers representing the positions of data in a data stream are therefore called timestamps.

As data samples arrive, the trigger-generating task counts them. When a sample satisfies the triggering conditions, the sample number for that sample is placed into the trigger. Think of the trigger as a kind of pipe, except that it contains timestamp information instead of sampled data.

Trigger-reading tasks are also associated with a data stream, which may be a different data stream than the one that the trigger writer scans. Samples are counted there as well. When the trigger-reading task receives an event timestamp from the trigger pipe, it looks for data at that position in its data stream.

Equalizing Data Rates

Samples scanned by the trigger reader and writer must appear at the same data rate. The most common reasons for different data rates are:

- data processing reduces the volume of data. For example, an **AVERAGE** command that averages input data in groups of ten also reduces the data rate by a factor of ten.
- multiple data channels. For the protective relaying example, one data channel is scanned for triggering conditions, but data are taken from four channels. This multiplies the data amount by a factor of four.

If the data rates do not match, the timestamps for the two streams do not correspond, and the software triggering will produce meaningless results.

One way to avoid data rate problems is to avoid commands that affect data rates. The **AVERAGE** command changes the data rate because it produces one output value for each block of values it reads, but **RAVERAGE** produces one output result for each input value, leaving the data rate unchanged.

This approach has drawbacks, however. Performing too many computations on a high-rate data stream can use up too much CPU capacity, forcing the application to operate at lower sampling rates.

A second option is to compensate for the different data rates. In the power relaying example, trigger events are determined by analyzing the data from one input channel pipe:

```
LIMIT(IP4, INSIDE, 19660, 32767, Tbreak, \
           INSIDE, 6553, 32767)
```

This trigger is asserted based on scanning data in a single channel, `IP4`. The following **WAIT** command retains data from four data channels using the input channel list `IP(0,1,2,3)`, which has 4 times as much data as the pipe `IP4`:

```
WAIT (IP(0,1,2,3), Tbreak, 1600, 8000, $BINOUT)
```

Failure does not occur, however. The **WAIT** command is specially constructed to detect input channel lists and apply rate adjustments for this case. No special action is necessary. This allows a trigger writing task to trigger on one input channel while the trigger reading task takes the corresponding data from any combination of input channels.

Suppose that the samples from the four data channels are first copied into a separate pipe using the following commands:

```
COPY (IP(0,1,2,3), P4)
WAIT (P4, Tbreak, 1600, 8000, $BINOUT)  // ERROR!
```

This configuration will fail. The **WAIT** command receives exactly the same data as before, but now there is a data rate problem. The **WAIT** command cannot know that data from four input channels are multiplexed in one data pipe.

Let's make the situation even worse. Suppose that the data are noisy, and in an attempt to reduce the level of the noise the voltage measurements are averaged in groups of five using a **BAVERAGE** command.

```
BAVERAGE (IP(0,1,2,3), 4, 5, Pavg5)
```

Now the data rate is increased by a factor of four because of the multiple channels, but immediately reduced by a factor of five because of the data processing.

The **TRIGSCALE** command can adjust timestamp values to account for rate differences. It can compensate for multiple channels and data rate changes. A multiplier factor of four accounts for the four channels, and a division factor of five accounts for the data processing reduction. After the averaging operation in groups of five, there are forty samples for each power cycle in each channel. The following is the modified DAPL configuration:

```
TRIGGERS   Tbreak,Tscaled
PIPES      Pavg5

PDEFINE    capture
  BAVERAGE  (IP(0,1,2,3), 4,5, Pavg5)
  LIMIT(IP4,  INSIDE,19660,32767,  Tbreak, \
              INSIDE,6553,32767)
  TRIGSCALE (Tbreak, 0,4,5, Tscaled)
  WAIT (Pavg5,  Tscaled,  320,1600, $BINOUT)
END
```

Starting and Stopping Triggers

Triggers act much like data pipes when accessed in more than one processing procedure. When a processing procedure is started, the DAPL system determines the number of trigger parameter references by task definition commands for that procedure. One of these references must be a trigger writing task, and the rest must be trigger reader tasks. The trigger readers do not have to be in the same procedure as the trigger writer, but all active readers for one trigger must be in the same processing procedure. The trigger begins continuous operation, with dynamic allocation and release of memory, when all reader and writer tasks for the trigger are started.

An important difference between a trigger and a data pipe is that a trigger does not retain past history when all readers and writers stop using it. Suppose there are two processing configurations, named `A` and `B`, with a task in procedure `A` writing trigger information and a task in `B` reading it. When the following sequence is executed, the reading tasks in group `B` will see no trigger events:

```
START A
STOP  A
START B
```

This occurs because no readers or writers remain active when procedure group `A` stops, so the triggers discard all old information.

On the other hand, when the following is executed, the tasks in group `B` will see the trigger events:

```
START A
START B
STOP  A
```

Note that continuous operation is never quite achieved in this example. The only time that all reader and writer tasks are simultaneously active is for the tiny interval after the readers start and before the writer stops. Hence, operation of the trigger has a finite capacity and will eventually terminate.

In the previous example, the following sequence would not be allowed:

```
START A
START B
STOP  A
START A
```

After any tasks using a trigger are stopped, all tasks accessing the trigger must be stopped to clear the trigger. After that, new tasks can use it.

Triggering Modes

Examples in other parts of this chapter concentrate on data capture. There are also applications that are less concerned about detecting and measuring special events, but more concerned about limiting the amount of data processed. For these applications, trigger operating modes are especially useful.

Data display requirements, for example, are very different from the requirements for data capture. It can often be assumed that the data are already captured and available; the problem is, which parts are needed for display? Some of the special requirements:

- Too much data transfer activity on the PC bus can interfere with other processing. Data transfer activity might need to be limited.
- Graphics displays are very slow. Time must be allowed to complete the display once a data block is accepted.
- Time must be allowed for the user to see and perhaps respond to the display.
- The display might need an occasional refresh even when nothing important occurs.

Trigger modes are very useful for coordinating data displays and process control applications. Trigger operating modes modify the way that trigger events are asserted, and can also generate events artificially. Operating modes act as filters, accepting some events, suppressing others. Trigger properties adjust the behaviors of the operating modes.

The operating modes are as follows:

- `NATIVE` Respond to all events

The `NATIVE` mode applies no filtering actions, and the trigger does not use any of the trigger properties. This mode is optimized for maximum speed when other triggering features are not needed.

- `NORMAL` Triggered display operation

The `NORMAL` mode uses the `HOLDOFF` property. This mode simulates normal mode operation of an oscilloscope, in which a display sweep must be completed before responding to another trigger event.

- `DEFERRED` Triggered displays for clustered events

This mode is the same as `NORMAL` mode, except that, instead of ignoring an event that occurs during the `HOLDOFF` interval, the event is delayed until just after the `HOLDOFF`

interval. This mode is useful when events tend to arrive in clusters rather than as isolated incidents.

- MANUAL Respond to single events

This mode is similar to hardware triggering using **HTRIGGER** ONESHOT. Processing is the same as NORMAL mode until an event occurs. The trigger responds to only this event, and then sets its GATE property to DISARMED. The trigger will not assert again until the GATE property is reassigned an ARMED property. See the discussion of the ARMED and DISARMED properties below.

- AUTO Triggered displays with self-timer

This mode is similar to NORMAL mode, except that artificial events are inserted at regular intervals when no events occur otherwise. This simulates the automatic triggering mode of an oscilloscope. The number of samples between artificial events is specified by the CYCLE property of the trigger. This mode is useful for applications where a data display must be refreshed periodically even if no events occur. Note that if the cycle is too small, real events can be buried in a large number of artificially generated events.

The operating modes use the following trigger properties to configure their operation:

- GATE Asynchronous enable and disable

Any trigger except native mode can be asynchronously enabled or disabled by assigning a value to the GATE property. When the GATE property is set to DISARMED, all trigger events are ignored until the property is set to ARMED. An initial value can be set when the trigger is defined. The value can be changed later using the **TRIGARM** command, or using the **EDIT** command. The exact sample at which the asynchronous arming or disarming takes effect is unpredictable, because it is not associated with a data event. All triggering modes except NATIVE mode respond to the GATE property. Default is ARMED.

- HOLDOFF Temporary disable after each event

This property specifies a number of samples during which no new assertions are accepted into the trigger pipe after asserting a trigger event. This simulates the holdoff operation of an oscilloscope, in which a display sweep is completed before responding to a new trigger event. A non-zero holdoff guarantees a time separation between consecutive events. The holdoff is applied both to real and artificial events. This property is most useful for `NORMAL` and `DEFERRED` operating modes, but can be used with all modes except `NATIVE`. Default is zero.

- `STARTUP` Temporary disable at initial startup

This property specifies an interval similar to `HOLDOFF`, except that events are ignored if they occur during the specified number of initial samples. This property is useful for systems that require a settling time before measurements can begin. Default is zero.

- `CYCLE` Automatic interval for artificial events

This property sets the number of samples between artificially generated events for the `AUTO` mode.

This is a lot of options, but in most cases it will be clear which is the best operating mode. Given the operating mode, choosing appropriate property values is usually very straightforward. Examples in the next section show some typical combinations of trigger modes and properties.

Applying Trigger Operating Modes

Oscilloscope Emulation Application

The PC must display data for events that occur frequently but not at precisely defined intervals. Triggering is used for two purposes: to extract a useful portion of the available data, and to "stabilize" the position of the data in a graphical display window. It is necessary to limit the update rate, so that the screen display is not chaotic.

For this application, `NORMAL` mode is selected. `NORMAL` mode uses a `HOLDOFF` property. When a block of data is selected for display, there follows a delay interval (number of samples) during which no additional trigger events are accepted. This number can match the data block size, or it can be longer to provide an extra delay.

Suppose that the PC application displays blocks of 500 samples. The display should show data for the special events only, and each display should remain for at least two seconds. For a data stream sampled at 50 microsecond intervals, a two-second `HOLDOFF` interval corresponds to 40000 samples. Configure the trigger as follows:

```
TRIGGERS  Tscope  MODE=NORMAL HOLDOFF=40000
```

Process Monitoring Application

For this application, data are again displayed, but special events are relatively rare. In fact, they are undesirable — they mean product defects. On the other hand, no defects means that there is little of interest to see in the data. Rather than leave the display screen empty, the display is occasionally refreshed with current data, interesting or not.

Suppose that display updates are required about once every five seconds. However, if a special event occurs, these data should be displayed immediately, but no more than two screen updates per second. Assume that a sample is taken every 100 microseconds, so that a five second delay corresponds to 50000 samples, and a 1/2 second delay corresponds to 5000 samples. Use a `CYCLE` property to set up the refresh interval, and a `HOLDOFF` property to enforce the two-per-second limit. Configure the trigger as follows:

```
TRIGGERS  TMonitor  MODE=AUTO CYCLE=50000 HOLDOFF=5000
```

Event Counting Application

For this application, product defects must be detected and counted using information from a sensor data stream. The defects show up as a disturbance. Very simple triggering can be used to detect disturbances and eliminate data that obviously contain nothing of interest. On the other hand, simple triggering is not able to distinguish a defect from an unrelated disturbance. To analyze the data, and recognize the actual defects, portions of the signal both before and after a possible defect must be retained. The `NORMAL` mode is suitable when defects occur in isolation. But if defects occur in clusters, the `NORMAL` mode will lose some of the context for an event near the end of a data block. So, select the `DEFERRED` mode.

Suppose that detecting a defect requires a data block with 40 points before and 88 points after each event. Configure a **`WAIT`** command to capture this data block. Set up the trigger in `DEFERRED` mode, with the property `HOLDOFF`=128, which covers both the before-event and after-event samples. Configure the trigger as follows:

```
TRIGGERS TOutlier MODE=DEFERRED HOLDOFF=128
...
WAIT ( PDATA, TOutlier, 40, 88, $BINOUT )
```

Destructive Tests and One-Shot Events

Destructive tests are discontinuous — after one test piece is stressed to failure, it must be scrapped and the next piece mounted. When a test piece is ready, there is one test, and one data block collected. Data collection must not be allowed until the next test is ready.

One way to do this is to start and stop the application repeatedly, but the `MANUAL` triggering mode is easier.

Suppose for example that each test involves loading a test piece until it fractures. For this application, start a data acquisition configuration using a trigger operating in `MANUAL` mode, with the `GATE=DISARMED` property. The trigger will not respond to anything because it is disarmed. Once the test is ready, enable the `GATE=ARMED` property by sending a nonzero number to the **`TRIGARM`** command through a data pipe. Actual data collection begins when the trigger is asserted, for example, after a non-zero force is measured. Once data collection begins, the `MANUAL` mode trigger changes its `GATE` property to `DISARMED`. The trigger will not respond to another event until `GATE=ARMED` is set again.

A configuration using the **TRIGARM** command to control the GATE property is as follows:

```
TRIGGERS TDestuct  MODE=MANUAL GATE=DISARMED
PIPE     PEnable
   ...

// Processing command to control trigger GATE
TRIGARM ( PEnable, TDestruct )
   ...
START
```

When it is time to run the next experiment, activate the trigger. To do this, put a nonzero value into the pipe monitored by the **TRIGARM** command:

```
// Send a command to arm the trigger
FILL      PEnable  1
// Trigger is now armed
   ...
```

Timestamp-Modifying Commands

Sometimes triggering is required when a combination of events occurs. The **TAND** and **TOR** commands allow combining trigger events from multiple sources to produce a new, composite trigger event.

For the protective relaying example, relaying events of interest might be only those where there is a large voltage imbalance. When a voltage imbalance occurs, voltage peaks are outside their normal range. Data are recorded if breaker operation *and* large voltage disturbance *both* occur within the same power cycle.

Six software triggers and three extra pipes are defined:

```
PIPES     PA1, PA2, PA3
TRIGGERS  Tph1, Tph2, Tph3
TRIGGERS  Tpeak, Tpickup, Tcombined
```

Suppose that 22000 is the nominal digitized peak voltage, so a peak outside the range 17600 to 26400 is more than 20% offset from nominal. The following processing commands detect the voltage imbalances:

```
ABS  (IP0, PA1)
ABS  (IP1, PA2)
ABS  (IP2, PA3)
PEAK (PA1, 1, Tph1, OUTSIDE,17600,26400)
PEAK (PA2, 1, Tph2, OUTSIDE,17600,26400)
PEAK (PA3, 1, Tph3, OUTSIDE,17600,26400)
```

This analysis produces three separate trigger streams, one for each phase. The **TOR** command combines these streams into a single event stream that represents voltage disturbance on *any* of the three phases:

```
TOR  (Tph1, Tph2, Tph3, Tpeak)
```

With 200 samples per cycle, any voltage disturbance event occurring within 200 cycles of a circuit breaker event is of interest. These are found using the **TAND** command.

```
TAND (Tpeak, Tpickup, Tcombined, 200)
```

Triggers and Independent ON/OFF Events

The applications so far capture data in fixed time intervals. In the power relaying example, a circuit-breaker event completes in twelve power cycles, so a fixed-size data block is appropriate. For other applications, the amount of data may not be known in advance.

For example, transient disturbances can induce torsional oscillations in the main shaft of rotating machinery. If these oscillations are large enough, they can lead to mechanical failure. By monitoring oscillation events, a cumulative assessment of damage can be made, and preventative maintenance scheduled.

The problem is that damping of the oscillations depends on unpredictable external conditions such as load characteristics and the presence of voltage compensation devices. The oscillations might damp out quickly or sustain for a dangerously long time. There is no way to know in advance whether small or large amounts of data are necessary.

Toggled trigger operation uses an `ON` condition to initiate data acquisition and a second `OFF` condition to terminate it. In the torsional oscillation example, mechanical strain measurements can be analyzed continuously by a digital filter tuned to the dominant oscillatory modes of the machine. If the filter's output reaches a sufficient level, an `ON` event occurs, and data acquisition begins. Once the filtered oscillations drop to insignificance, an `OFF` event occurs, and data acquisition is terminated. Analysis and data logging are performed off-line by the PC host. The point is that triggering is controlled by two events, rather than just one.

DAPL provides three special commands to support toggled trigger operation.

- **`TOGGLE`**
- **`TOGGWT`**
- **`TCOLLATE`**

The **`TOGGLE`** command detects `ON` and `OFF` events, enforcing a strict alternating protocol. The **`TOGGWT`** command takes data from an input stream under control of the events asserted by the **`TOGGLE`** command. The **`TCOLLATE`** command provides an alternative means for generating an `ON/OFF` event stream.

For the torsional monitoring example, the signal from a custom-designed filtering task is used to detect oscillations. The details of this filter are not discussed here. The maximum frequency at which damage can occur is presumed to be about 75 Hz, so a minimum sampling frequency of about 150 Hz is necessary for successful filtering.

Assume that 20x oversampling is used, but the digital filtering decimates by a factor of ten. This leaves a factor of ten difference between the data rates of the filtered data and the raw strain data. Presume that an engineering analysis has determined that a filter output of more than 4000 on a scale of 0 to 32767 indicates potential damage, and that a filter output of less than 2000 indicates that danger is past.

The following DAPL configuration can be used to continuously monitor the strain data:

```
IDEFINE  samp  1
  SET  IP0  s0    // phase1
  TIME   333.30   // 150 Hz x 20 oversample
END
```

The following DAPL configuration performs the processing. It uses a custom command `CFILT` to filter the signal and the **`TOGGLE`** command to signal ON and OFF events depending on the output level of `CFILT`. The **`TRIGSCALE`** command compensates for the factor of 10 data reduction applied during digital filtering. The **`TOGGWT`** task then sends the selected strain data to the PC for logging and analysis.

```
PIPE      P1
TRIGGERS  TOGGLE, SCTOGGLE

PDEFINE  detect
  CFILT (IP0, P1)  // decimates by 10
  TOGGLE(P1, INSIDE, 4000,32767, \
             OUTSIDE, 2000,32767, TOGGLE )
  TRIGSCALE(TOGGLE,0,10,0,SCTOGGLE)
  TOGGWT(IP0, SCTOGGLE, $BINOUT)
END
```

Triggering with Multiple-Data Acquisition Processors

The **TRIGSEND** and **TRIGRECV** commands are useful in applications where there are many data channels and acquisition must be coordinated among multiple Data Acquisition Processors. For example, assume that it is necessary to sample 15 analog channels simultaneously, at 250 kHz. Data capture must begin when a triggering event is detected on a 16th channel. The 250 kHz rate is too fast for simultaneous sampling expansion accessory cards. A DAP 5400a/627 can capture up to eight independent data channels simultaneously, so two DAP 5400a/627 boards configured in a master and slave connection can meet the sampling requirements. One Data Acquisition Processor reads eight data channels, and the other reads seven data channels and the synchronizing signal. However, this leaves a problem. One of the Data Acquisition Processors does not see the data from the triggering channel. How can that Data Acquisition Processor know when to retain its data?

The **TRIGSEND** command encodes triggering information and sends it through communication pipes to other Data Acquisition Processor boards. The **TRIGRECV** command on the receiving board decodes the triggering information, writing it into triggers where it can be accessed to control data acquisition. This provides the means for exchange of triggering information.

The command used to generate the trigger signal is not important, as long as it is able to sustain continuous operation. For a DAP 5400a/627, any of the triggering commands provided by the DAPL operating system can do this easily. A **LIMIT** command is shown as the trigger-generating command for this example. Assume that the board configured as the **MASTER** detects triggering events.

Each Data Acquisition Processor must be configured to access communication pipes for transferring triggering information. Extra communication pipes for this purpose can be set up using the Accel32 control panel application.

- Select the `Browser` tab
- In the graphical device tree window, expand the display for the two Data Acquisition Processors.
- Right click on the `Compipes` element.
- In the dialog box, select `Create`.

In this manner, create a `Cp2Out` pipe for the master Data Acquisition Processor, and a `Cp2In` pipe for the slave. Select the `long` data type for each. Alternatively, applications can create communications pipes using features of the Accel32

programming interface, instead of using a configuration set up by the Accel32 control panel application.

An application program on the PC will receive the captured data. It is presumed that this same application is also available to connect the master-to-PC and PC-to-slave communications. The application simply copies all of the data it receives from the `Tsend` pipe into the `Treceive` pipe.

Now the two Data Acquisition Processors can be configured for input sampling. Since a DAP 5400a/627 samples channel groups of 8 channels, each Data Acquisition Processor is configured to sample only one channel group. The master Data Acquisition Processor is configured as follows:

```
IDEF A
  GROUPS  1
  SET  IP(0..7)  SPG0
  TIME 4
  MASTER
END
```

The slave board is configured similarly, except that the **MASTER** command is replaced by **SLAVE**.

The boards must also be configured for processing. The configuration is almost the same, except that the master board detects and sends trigger events to the slave, while the slave receives and responds to the events. Note that the slaves capture 8 data channels, while the master board captures only 7, so the data blocks transferred to the PC application from the slave are larger. If this is a problem, the trigger channel could be used as a source of padding data on the master board, to make the blocks the same size.

Processing on the master Data Acquisition Processor:

```
TRIGGER  TXF

PDEF B
  // Monitor one channel and generate trigger events
  LIMIT(IP7,INSIDE,LOWLIM,HILIM,TXF)
  // Notify slave of events and progress
  TRIGSEND(TXF, 1000, Cp2Out)
  // Capture 7 data channels
  WAIT (IP(0..6),TXF,0,7000,$BINOUT)
END
```

Processing on the slave Data Acquisition Processor:

```
TRIGGER  TXF

PDEF B
  // Receive trigger events
  TRIGRECV(Cp2In,TXF)
  // Capture 8 data channels
  WAIT (IP(0..7),TXF,0,8000,$BINOUT)
END
```

A slave board configuration is completed by interaction with the master board. To prepare for this, the slave board must be started first. The slave will not do anything until the master is ready.

Start the master Data Acquisition Processor last, to complete the initialization. Both boards will begin sampling and buffering data simultaneously, but will not retain any data until the triggering conditions on the master board are satisfied.

Asynchronous Events and PCASSERT

PC applications run at bottom priority. A PC application can proceed when computing resources are available to it. When exactly this will occur is unpredictable and unmeasurable. The only thing the application knows is that it is running *now* and needs data *now*. Software triggering provides a means of selecting data on an as-needed basis.

The difficulty is, when the PC asks for data *now*, what exactly does that mean? The **PCASSERT** command provides an answer. It maintains information about the status of a data stream, as current as possible. When a request arrives, the **PCASSERT** command uses the status information to artificially manufacture a trigger event. This event is then used to select a block of data for the PC.

When there is a single stream of samples, **PCASSERT** can use the system hardware sample status to derive a sample number. For example, suppose that the PC needs a block of 400 samples from a single channel. The PC signals the **PCASSERT** command by placing a number into the binary input pipe. When it receives the request, the **PCASSERT** command then determines an appropriate timestamp and asserts an event in trigger `TRIGPC`.

```
PCASSERT ($BININ, TRIGPC)
WAIT (IP0,TRIGPC,200,200,$BINOUT)
```

No reference stream is provided, so the **PCASSERT** command looks up the current sample count from the hardware status, and uses that to generate the event timestamp. The **WAIT** command responds to the event, takes 200 pre-trigger and 200 post-trigger samples, and sends them immediately to the PC.

This works because samples are taken from a single data channel. What about the case when there are multiple channels? The system sample count, which includes all samples from all channels, is wrong for taking samples from individual channels.

There are several ways to get around this. One easy way is to use the optional third parameter of the **PCASSERT** command to specify a data reduction factor. If there are four input channels, for example, the rate in any data channel is 1/4 of the net system rate. Thus, we can specify

```
PCASSERT ($BININ,TRIGPC,4)
WAIT (IP(0,1),TRIGPC,200,200,$BINOUT)
```

Recall that the **WAIT** command is aware of the number of input data channels in its input list. Once **PCASSERT** asserts a meaningful sample number for a single channel, the **WAIT** command can read from any number of channels.

Another way is to allow the **PCASSERT** to monitor one of the data channels.

```
PCASSERT ($BININ, TRIGPC, IP0)
```

This is usually not necessary for input channel data. But suppose the **WAIT** command takes blocks of processed data, which might result from another triggering process. Now the system sampler count has no useful relationship to the number of samples available in the data pipe.

For this situation, we can allow **PCASSERT** to monitor any appropriate data stream to maintain its reference count. Suppose that pipe `ANYPIPE` contains an arbitrary data stream. The following commands monitor and extract current data from this pipe:

```
PCASSERT ($BININ, TRIGPC, ANYPIPE)
WAIT (ANYPIPE,TRIGPC,200,200,$BINOUT)
```

The following example illustrates fetching blocks of 256-point FFT spectrum data to a PC application. This example uses an alternate means of signaling the **PCASSERT** command.

Suppose that **FFT** computations occur continuously, faster than the PC can use all of the data.

```
FFT (5, 9, 0, IP0, SPECTRA)
```

The PC requests a spectrum block by setting variable `VREQ` to a nonzero value using a `LET` command:

```
LET VREQ=1
```

The **PCASSERT** command monitors the number of samples available in the `SPECTRA` pipe.

```
PCASSERT(VREQ, TRIGPC, SPECTRA)
```

The data appear in blocks of 256 samples, but this means that **PCASSERT** will always detect the last sample in a data block. We want the beginning of a block, not the end. We correct this problem by modifying the event timestamps, using the **TRIGSCALE** command as described earlier in this chapter.

```
TRIGSCALE(TRIGPC, 0, 256, 256, TRIGSP)
```

This operation produces trigger timestamps in trigger `TRIGSP` that indicate the starting locations of completed spectrum blocks in storage. The **WAIT** command takes each block beginning at the sample indicated. Data that are not requested are silently discarded.

```
WAIT(SPECTRA, TRIGSP, 0, 256, $BINOUT)
```

This technique of pre-computing values and having them ready to go upon request provides extremely fast response.

15. Digital Filtering

Digital filtering removes unwanted frequency components from digital data. This chapter describes digital filtering commands available in DAPL.

Average and Running Average

AVERAGE and **RAVERAGE** implement two of the simplest digital filters. **AVERAGE** reads blocks of 'n' data values and returns the averages of the blocks. **RAVERAGE** maintains a moving block of 'n' data values, and returns averages of the moving block. **RAVERAGE** starts by reading in enough data values to fill one block and returning one average value. Then, it repeats a sequence of throwing out the oldest value in the block, reading a new value into the block, and returning the average of the block.

It is important to note that **AVERAGE** reduces the data rate, where **RAVERAGE** does not. **AVERAGE** returns one value for each 'n' values it reads. After initially filling the block it maintains, **RAVERAGE** returns one value for each value it reads.

AVERAGE and **RAVERAGE** implement simple lowpass filters. These commands should be considered for some applications, especially for reducing broad-band random noise. Additional digital filtering commands provided by DAPL implement specialized frequency-selective filters.

Finite Impulse Response Filters

The digital filtering commands **FIRFILTER** and **FIRLOWPASS** implement finite impulse response (FIR) filters. A finite impulse response filter is determined by a vector v of filter coefficients. The filter output corresponding to a block of data is calculated by multiplying each entry in the block of data by the corresponding entry in the vector v and adding the products. This means that the block of data must have the same length as the length of the vector.

When a digital filtering task starts to process a stream of data, the task first begins by reading in enough values to produce one sum of products. In a typical filtering application, **FIRFILTER**, like **RAVERAGE**, maintains a block of data. Each time **FIRFILTER** reads one data value, it removes the oldest data value from its block of data and appends the new value to the block. It then calculates one output value. Thus, after a startup sequence, **FIRFILTER** returns one filtered value for each input value.

FIRFILTER and **FIRLOWPASS** can operate either like **RAVERAGE**, generating one result for each input sample received, or like **AVERAGE**, reducing the output data rate by discarding some of the filtered data. Reducing the amount of data is appropriate for many applications in which an input is oversampled and then passed through a lowpass or bandpass filter. After filtering removes the high frequencies, fewer samples are needed to accurately represent the resulting signal, and there is no need to retain all of the filtered data.

FIRFILTER has a parameter n called “decimation” that specifies data reduction. After each calculation, **FIRFILTER** reads in n-1 values without calculating. Then, after reading the n-th value, **FIRFILTER** calculates another value.

FIRFILTER also has optional “take” and “skip” parameters that allow blocks of data to be alternately retained or rejected, retaining complete signal information locally but reducing the overall volume of data.

FIRLOWPASS is a variant of **FIRFILTER**. **FIRLOWPASS** allows values of 2 through 12 for the decimation factor, and provides predefined symmetric filter vectors appropriate for lowpass filtering using these levels of decimation.

Generating Filter Coefficients

Filter coefficients may be calculated from the frequency spectrum of an ideal filter. This procedure is implemented in the program FGEN from Microstar Laboratories.

Window Vectors

Most ideal filter characteristics have a filter vector with an infinite number of terms. Approximating the infinite filter vector with a filter vector having a finite number of coefficients leads to approximation errors. All FIR filters suffer from this to some degree. Effects related to the finite filter length are decreased by multiplying the coefficient values by a window function. This typically is a function whose values approach zero near the edges of the coefficient block. FGEN allows a window as part of a filter specification.

Phase Response and Time Delay

All of the filters designed by FGEN have coefficients that are symmetric around the middle coefficient. A symmetric FIR filter having `2M+1` coefficients has the property that the output values are delayed by `M` samples. This delay is sometimes interpreted in the frequency domain as a phase shift. A pure time delay causes an apparent phase shift at each signal frequency proportional to the frequency; for this reason symmetric filters are sometimes called linear-phase filters. For purposes of analysis, the phase-shift point of view is very important, but in the sampled-data world, the time-delay interpretation is more immediately useful.

The time delay is of critical importance, for example, when analyzing peaks in a filtered signal for purposes of triggering, For such applications, a 'phase correction' parameter is provided by the **`FIRFILTER`** command. The correction parameter can be set to the value `M`, or equivalently, the value `-1` can be specified and the **`FIRFILTER`** command will calculate the correct value to use. **`FIRFILTER`** makes the correction by adding extra samples to the output stream. After this correction, features in the filtered data stream will correspond in time to features of the original unfiltered data stream.

This synchronization of the input and output data streams should not be confused with real-time response. `2M+1` samples are still required before the first output value can be computed, and the first computed output corresponds to filter term `M+1`. There is a real-time delay of `M` samples between the most current sample taken and the most recent filter output generated. This delay cannot be avoided when using a symmetric filter.

It should be noted that the **`FIRFILTER`** command is not restricted to using symmetric filters. Filters designed using other techniques may have other delay characteristics and could require a different time shift correction.

16. Fast Fourier Transform

The Fourier transform is a mathematical operation that converts data from the time domain to the frequency domain. Using the Fourier transform, complex signals are represented in terms of simple signals; each of the simple signals is a “pure” oscillatory component, either a complex exponential, or a cosine or sine.

The discrete Fourier transform (DFT) is a mathematical operation that approximates the Fourier transform for blocks of sampled data. The frequencies of the simple signals in a DFT are harmonics of one frequency, called the fundamental frequency. The DFT is appropriate for the data derived by sampling a continuous signal with an analog-to-digital converter. The fast Fourier transform (FFT) is an extremely efficient algorithm for calculating the DFT.

The DFT has two forms, which are inverse operations. The forward transform goes from time domain data (a sequence of samples over time) to frequency domain data (a sequence of frequency harmonics spanning a frequency band); the inverse transform goes in the other direction, from frequency domain data to time domain data. Because DAPL is used primarily for computations on acquired data, the forward transform is used more often than the inverse transform.

Even with real input data, the Fourier transform produces complex-valued output data. In many cases the interesting information in the output data is contained in the amplitude, the amplitude and phase, or the square of the amplitude. The square of the amplitude is called power, because in many applications it can be interpreted as the power at a specified frequency.

FFT Commands

DAPL provides **FFT** commands that combine a number of mathematical operations into a convenient package. The operations include window pre-processing, the actual transform, and post-transform data conversions. Most applications involve signals containing various degrees of random noise; the effects of random noise typically dominate any “noise” from numerical roundoff. The FFT algorithms require a block size that is a power of 2. The two **FFT** commands accept block sizes from 4 to 16384.

The FFT algorithms require pre-computed tables of coefficients. When FFT sizes are large, these tables become large. To avoid unnecessary construction of FFT coefficient tables, these tables are not computed until an FFT task is defined. A small FFT can work with a large table, but a large FFT cannot work with a small table. If FFT sizes are mixed, define the tasks with a large FFT size first.

FFT Modes

The **FFT** command provides selected combinations of input, transform, and output processing operations that cover most application requirements. These combinations are called modes.

The seven modes of **FFT** are:

0. forward transform, real input data, complex output data
1. forward transform, complex input data, complex output data
2. inverse transform, complex input data, real output data
3. inverse transform, complex input data, complex output data
4. forward transform, real input data, power spectrum output data
5. forward transform, real input data, amplitude spectrum output data
6. forward transform, real input data, amplitude and phase output data

When the forward fast Fourier transform is applied to real data, half of the output values are redundant. For blocks of size `N`, for example, the (`N`-`n`)-th term is equal to the complex conjugate of the `n`-th term, and the 0-th and the `N`/2-th terms are real. Because of this symmetry, modes 4, 5, and 6 return output blocks of half of the size of the input blocks. Power and magnitude computations take the symmetry into account, and combine the effects of reflected terms.

Window Vectors

The underlying assumption of an FFT is that samples in a data block represent one period of a periodic signal. Often, this is not actually the case, and a transform is applied to a data block extracted from a continuous data stream. In this case, the computed FFT exhibits both the frequency components present in the data and artificial frequency components caused by isolating the data block. It is possible to minimize the data blocking effects by multiplying the values in the input data block, term-by-term, by a vector of coefficients called a window vector. The coefficients in a window vector usually decay to zero at the ends of the block.

Using a window has some drawbacks. Information near the ends of the block is reduced or lost. Statistical interpretation of the transform result is less clear, because errors in the original data are weighted rather than uniform. A window vector generally changes the output spectrum, altering the zero-frequency component, and smoothing peaks in the frequency domain so that they are somewhat broader and less distinct.

DAPL provides the five most common non-parametric window types: `Rectangular`, `Hanning`, `Hamming`, `Bartlett` and `Blackman`. The `Rectangular` window has all coefficients equal to 1.0; this window is equivalent to applying no window operation.

The other windows are given by:

```
1.Hanning(k)  = 0.5 - 0.5 cos(2pk/N)
2.Hamming(k)  = 0.54 - 0.46 cos(2pk/N)
3.Bartlett(k) = 2k/N       for k < N/2,
                (2 - 2k/N) for k >= N/2
4.Blackman(k) = 0.42 - 0.50 cos(2pk/N) + 0.08 cos(4pk/N)
```

where `N` is the block size.

All of the FFT modes accept a window. A window is usually meaningful only for forward transforms. A predefined window vector is specified in the task definition by setting the window parameter to a numeric constant in the range 0 to 4, corresponding to the window types.

Customized windows can also be specified. For these windows, the coefficients are specified by a DAPL **`VECTOR`**, and the name of the vector rather than a numeric constant is specified in the task definition parameter list.

Scaling in the FFT

The easiest way to describe the scaling of the various FFT modes is to give the results of applying the FFT to a sine wave of the maximum amplitude 10000, at the fundamental frequency or at any harmonic frequency less than half of the sampling frequency. The transform of this sine wave has two components, one at the frequency of the sine wave and one at the sampling frequency less the frequency of the sine wave.

FFT modes 0 and 1 are scaled so that each component of the output has magnitude 10000/2. Mode 4 is scaled so that the output is the square of the RMS value corresponding to the amplitude of the input; the output has magnitude (10000*10000)/2. Mode 5 is scaled so that the amplitude of the output is the RMS value corresponding to the amplitude of the input; the output has magnitude 10000/sqrt(2). Mode 6 returns the amplitude along with the phase angle.

Modes 2 and 3 are inverse FFT modes. These modes are scaled so that applying the FFT, followed by the inverse FFT, returns the original data.

The scaling used by DAPL is different from the conventions used in many (but not all) DSP textbooks. Most books represent a forward DFT as an unscaled sum of terms, and a reverse DFT as a similar sum of terms scaled by an a factor of 1/`N`. The 1/`N` multiplier derives from the transform theory, and must be applied to either the forward direction or the reverse direction transform for the transform pair to restore the original data. However, there is no essential reason for applying the 1/`N` factor to the reverse transform. In fact, there are distinct advantages to applying it to the forward transform. A DFT computation involves summing a large number of terms, and this summation is subject to arithmetic overflow conditions. Overflow leads to loss of information about frequency peaks. In most cases, this is the information that is of the most value. Applying the 1/`N` factor during the forward transform yields the well-scaled transform previously described, without exposure to overflow and without loss of information at frequency peaks.

Representing Sampled Data

The input for the fast Fourier transform is a block of samples of a time-dependent signal u(t). For the fast Fourier transform, values in the input data must be sampled at equally spaced times. Denote the time between samples by τ; then the sampling frequency is $1/\tau$. If time zero denotes the time at which the first sample is taken, then the k-th sample is taken at time

$$t_{[k]} = k\tau.$$

The samples form a block

$$z = (z_{[0]}, z_{[1]} \ldots, z_{[N-1]}),$$

where $z_{[k]} = u(t_{[k]})$ denotes the k-th sample of the signal u(t).

If the block length is denoted by N, the time per block is

$$T = N\tau.$$

The Fourier transform represents the sampled signal u(t) in terms of signals that are periodic in t with period $T = N\tau$. The fundamental frequency F is given by

$$\begin{aligned} F &= 1/T \\ &= (1/N)(1/\tau) \end{aligned}$$

The n-th harmonic has frequency $f_{[n]}$, where

$$\begin{aligned} f_{[n]} &= nF \\ &= n/T \\ &= (n/N)(1/\tau). \end{aligned}$$

The 0-th harmonic is a special case; its frequency is $f_{[0]} = 0$. This corresponds to a constant term whose size is proportional to the average value of the sampled signal u(t).

When sampling a real-valued signal at an input pin at 1024 samples per second and calculating a 2048-point fast Fourier transform, for example, the fast Fourier transform returns information about input frequencies up to 512 Hz in steps of 1/2 Hz. Non-redundant frequency information is contained in FFT output components 0 through 1023. Component 0 represents the DC component of the input signal, component 1 represents the (1/2048)*1024 Hz = 0.5 Hz component of the input

signal, component 2 represents the (2/2048)*1024 Hz = 1.0 Hz component of the input signal, etc.

Nyquist Frequency

The Nyquist frequency is $f_{[N/2]}$, half the sampling frequency. The n-th frequency and the (N-n)-th frequency are symmetrically placed with respect to the Nyquist frequency.

In a typical example, the Data Acquisition Processor takes samples at 100,000 samples per second and computes 1024-point FFTs. Then the sampling time τ equals 10 microseconds, $N = 1024$, and $T = 0.01024$ seconds. The Nyquist frequency is 50 kHz, and the frequencies corresponding to the FFT output are 0, 97.7 Hz, 195.3 Hz, …, 99.9 kHz.

Through the phenomenon of aliasing, the FFT cannot distinguish frequencies above the Nyquist frequency from frequencies below the Nyquist frequency. Frequencies above the Nyquist frequency should be considered equivalent to their symmetric frequencies below the Nyquist frequency. Because of aliasing, the components at frequencies

$$f_{[(N/2)+1]},\ f_{[(N/2)+2]},\ \ldots,\ f_{[N-1]}$$

appear to have frequencies

$$f_{[(N/2)-1]},\ f_{[(N/2)-2]},\ \ldots,\ f_{[1]}$$

To eliminate aliasing, the input signal for the FFT must be filtered to remove all components at or above the Nyquist frequency. Frequencies above half of the sampling frequency must be removed.

There are two common approaches to eliminating unwanted high frequencies. The first uses hardware anti-aliasing filters. The second uses digital filters. Digital filtering may be used when the noise spectrum is sufficiently band-limited so that it is possible to represent both the desired frequencies and the high frequency noise by sampling at a rate much higher than required by the FFT. (This is called oversampling). A lowpass digital filter (such as a DAPL **FIRLOWPASS** task) eliminates all high frequencies above the FFT's Nyquist frequency and then decimates the data to the sampling rate required by the FFT.

Representing Sample Data with Complex Exponentials

The remainder of this chapter explains the use and interpretation of the Data Acquisition Processor FFT commands. Some of the mathematical details should be skipped on a first reading.

For sampled data, the Fourier transform represents a sample block z in terms of periodic functions of sample number k. The standard representations use either complex exponentials or cosines and sines. For computational purposes, the most useful form of the Fourier representation involves complex numbers and complex exponentials. This representation eliminates many trigonometric identities; it also simplifies phase computations.

In terms of the sample time $t = t_{[k]}$, the complex exponentials are given by

$$E_{[n]}(t) = \cos(2\pi(nt/N\tau)) + i \sin(2\pi(nt/N\tau))$$

In terms of the sample number k, the complex exponentials are given by

$$E_{[n]}(k) = \cos(2\pi nk/N) + i \sin(2\pi nk/N)$$

The Fourier transform represents the sample block z in terms of the special blocks given by

$$E_{[n]} = (E_{[n]}(0), E_{[n]}(1), \ldots , E_{[n]}(N-1))$$

Because of the periodicity of the sines and cosines, there are only N distinct blocks E_n. For N = 4, for example, the 4 distinct blocks are:

$$E_{[0]} = (1,\ 1,\ 1,\ 1)$$
$$E_{[1]} = (1,\ i,\ -1,\ -i)$$
$$E_{[2]} = (1,\ -1,\ 1,\ -1)$$
$$E_{[3]} = (1,\ -i,\ -1,\ i)$$

Notice that $E_{[3]}$ is the complex conjugate of $E_{[1]}$. More generally, $E_{[N-n]}$ is the complex conjugate of $E_{[N]}$ for all n.

A short computation may clarify the Fourier transform representation. Consider a signal u(t), which is sampled four times, yielding the samples 1000, 3000, 4000, and 1000 at time 0, τ, 2τ, and 3τ. The Fourier transform represents u(t) as

$$u(t) = a_{[0]} E_{[0]}(t) + a_{[1]} E_{[1]}(t) + a_{[2]} E_{[2]}(t) + a_{[3]} E_{[3]}(t)$$

The coefficients $a_{[0]},\ldots, a_{[3]}$ can be determined from the values `u(τ)` at times 0, τ, 2τ, and 3τ. This leads to the equations

```
(1000, 3000, 4000, 2000) =  a0 (1,  1,  1,  1)
                         +  a1 (1,  i, -1, -i)
                         +  a2 (1, -1,  1, -1)
                         +  a3 (1, -i, -1,  i)
```

A short matrix calculation gives

```
a0 =  2500
a1 = -750 - i 250
a2 =  0
a3 = -750 + i 250
```

After some algebraic simplifications, this leads to the following representations:

```
u(t) =  2500 - 1500 cos(2πt/4t) + 500 sin(2πt/4t)
u(kt) = 2500 - 1500 cos(2πk/4) + 500 sin(2πk/4)
```

Care is required in interpreting this result, as the formula for the signal `u(t)` gives `u(t)` exactly at only the four discrete values 0, τ, 2τ, and 3τ.

Representing Sampled Data with Cosines and Sines

In the previous computation, the imaginary terms canceled out to give a real representation of a real signal in terms of cosines and sines. The cancellation must occur whenever the Fourier transform is applied to real data, so there is an alternate Fourier representation in terms of cosines and sines. In terms of the sample time $t = t_{[k]}$, the Fourier transform represents a signal in terms of

$$c_{[n]}(t) = \cos(2\pi(nt/N\tau))$$
$$s_{[n]}(t) = \sin(2\pi(nt/N\tau))$$

In terms of k, the corresponding discrete values are

$$C_{[n]}(k) = \cos(2\pi nk/N)$$
$$S_{[n]}(k) = \sin(2\pi nk/N)$$

Because the cosines and sines are periodic, only certain values of n give distinct data blocks. For example, for N = 4, there are 4 distinct standard data blocks:

$$C_{[0]} = (1,\ 1,\ 1,\ 1)$$
$$C_{[1]} = (1,\ 0,\ -1,\ 0)$$
$$C_{[2]} = (1,\ -1,\ 1,\ -1)$$
$$S_{[1]} = (0,\ 1,\ 0,\ -1)$$

and for N = 8, there are 8 distinct blocks:

$$C_{[0]} = (1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1)$$
$$C_{[1]} = (1,\ r,\ 0,\ -r,\ -1,\ -r,\ 0,\ r)$$
$$C_{[2]} = (1,\ 0,\ -1,\ 0,\ 1,\ 0,\ -1,\ 0)$$
$$C_{[3]} = (1,\ -r,\ 0,\ r,\ -1,\ r,\ 0,\ -r)$$
$$C_{[4]} = (1,\ -1,\ 1,\ -1,\ 1,\ -1,\ 1,\ -1)$$
$$S_{[1]} = (0,\ r,\ 1,\ r,\ 0,\ -r,\ -1,\ -r)$$
$$S_{[2]} = (0,\ 1,\ 0,\ -1,\ 0,\ 1,\ 0,\ -1)$$
$$S_{[3]} = (0,\ r,\ -1,\ r,\ 0,\ -r,\ 1,\ -r)$$

where r represents $\sin(\pi/4) = \cos(\pi/4) = 0.707\ldots$

Symmetry Around the Nyquist Frequency

Because k and n always are integers, and the sines and cosines are periodic, the following formulas hold:

$$E_{[N-n]}(k) = \cos(2\pi(N-n)k/N) + i \sin(2\pi(N-n)k/N)$$
$$= \cos(2\pi nk/N) - i \sin(2\pi nk/N)$$
$$= E_{[-n]}(k)$$

for all n and k. This means that the complex exponentials at frequencies that are symmetric about the Nyquist frequency are complex conjugates. In particular, all of the components at frequencies above the Nyquist frequency can be represented as sines and cosines at frequencies below the Nyquist frequency. The representation of frequency components above the Nyquist frequency in terms of frequency components below the Nyquist frequency is called aliasing.

From the previous formulas,

$$E_{[n]}(k) + E_{[-n]}(k) = 2 \cos(2\pi nk/N)$$
$$E_{[n]}(k) - E_{[-n]}(k) = 2i \sin(2\pi nk/N)$$

Using these formulas, real data are represented later in this chapter in series of sines and cosines with real coefficients.

Interpreting the FFT

The FFT acts on blocks of N complex values

$$z_{[k]} = x_{[k]} + i\, y_{[k]}$$

where k = 0, 1, … , N-1. The FFT returns blocks of N complex coefficients

$$X_{[n]} + i\, Y_{[n]}$$

where n = 0, 1, … , N-1. The k-th term in the original block of data equals the sum from n = 0 to N-1 of the terms

$$Z_{[n]}(k) = (X_{[n]} + i\, Y_{[n]})\, E_{[n]}(k)$$

up to computational accuracy. Because the complex exponentials are periodic in k, the complex coefficients can be interpreted as the frequency components of the original data.

Note: If the original data are multiplied by a window function before the FFT is computed, the FFT gives a representation of the modified data, rather than a representation of the original data.

Interpreting the FFT for Real Data

If the input data for the FFT are real, then the coefficients satisfy the symmetry relationship

$$X_{[n]}(k) = X_{[N-n]}(k)$$
$$Y_{[n]}(k) = -Y_{[N-n]}(k)$$

for all k. From these formulas, the n-th coefficient and the (N-n)-th coefficient are complex conjugates, and the 0-th and N/2-th terms are real. The n-th term of the Fourier series is

$$(X_{[n]} + i\ Y_{[n]})\ E_{[n]}(k)$$

and the (N-n)-th term of the Fourier series is

$$(X_{[N-n]} + i\ Y_{[N-n]})\ E_{[N-n]}(k)$$

Adding these terms gives

$$2\ X_{[n]}\ \cos(2\pi nk/N) - 2\ Y_{[n]}\ \sin(2\pi nk/N)$$

This gives the representation of the original data in sines and cosines. Note that the 0-th term and the N/2-th term are real and do not have symmetric terms. The 0-th term is the average of the original data block, and should not be multiplied by 2. The N/2-th term corresponds to the Nyquist frequency. For sampled input data, this term should be very close to zero.

In a sense, half of the computations are unnecessary when an FFT is applied to real-valued data because half of the computed results are redundant. For real-valued data, DAPL takes advantage of symmetry properties to compute an FFT of size N in approximately the same amount of time as an FFT of size N/2. The modified algorithm is applied automatically for real-valued data. Further attempts to obtain speed advantages using symmetry properties will not improve performance, and are not advised.

Errors in the FFT

The FFT must be interpreted with care. Errors are introduced by sampling at discrete times, by sampling for only a finite interval of time, by rounding, and by truncation.

An FFT is a multistage algorithm performed using fixed-point arithmetic. Even though great care is applied to maintaining the accuracy of intermediate results, the smallest rounding errors can propagate from stage to stage, affecting low-order bits in many locations of the final result. The larger the FFT, the more likely that errors will accumulate. Do not expect transforms to be exact to the very last bit.

The algorithms used to compute transforms for real-valued data require a special final stage that collects and reconstructs symmetric transform results. This process is subject to rounding error, as is any other computation, and can introduce errors into the low-order bit.

Accumulation of errors is particularly apparent in the reverse transforms. The 1/`N` factor of the forward transform tends to hide most of the truncation and rounding error, but the reverse transform does not have a 1/`N` factor. As a rule of thumb, if the size parameter for the FFT is `m`, the last `m`/2 bits in a reverse transform are noisy. For example, in a 256 point transform, `m`=8, so do not attach significance to differences less than 16. The error in the reverse transform usually is well modeled by "white noise." Averaging can sometimes cancel some of the noise, yielding additional significant bits.

Scaling effects are much more evident in the reverse transform. Because the reverse transform is not scaled by 1/`N` like the forward transform, an arbitrary frequency spectrum is likely to produce saturated time-domain peaks. An inverse transform applied to data derived from a forward transform is less likely to be affected by saturation because the 1/`N` factor is already in effect. On the other hand, because of the 1/`N` factor, the forward transform loses some of the information originally present in the low order bits of the original samples, and the difference appears as various small non-random artifacts in the reverse transform.

Error accumulation, truncation, and scaling considerations always apply to some degree, depending on the number representation. If speed is less important than preserving precision, consider using floating point data types. Floating point data types have automatic internal scaling that is very effective for avoiding precision-related error accumulation. Better precision does not remove any noise present in the original input signal, however, so do not confuse precision with accuracy.

Even if the Data Acquisition Processor receives a periodic analog signal at its input pins, the sampled data typically are not periodic with period equal to the block length of the FFT. Windowing compensates for data blocking errors, but introduces other errors. Windowing artificially makes the data look periodic, but the transform of the windowed data may differ substantially from the transform of the original data.

An FFT analyzes the frequency content of a signal in terms of harmonics of the fundamental frequency. If the signal contains a pure oscillatory wave at one of the harmonic frequencies, only one complex term of the resulting FFT is nonzero. If the original signal is a pure oscillatory wave, but does not coincide exactly with one of the harmonics in the transform, the signal appears "smeared" into neighboring locations, as if the signal were not pure. This phenomenon is known as "leakage." Do not interpret a nonzero value in a spectrum as meaning that a signal of precisely that frequency is actually present in the original signal. Note that the presence of a signal that is not a harmonic of the fundamental frequency also implies that the FFT block does not exactly represent one period of the original waveform, hence leakage and windowing are related. Windowing often reduces the effects of leakage, but alters the frequency spectrum in different ways.

Noise is present in most real-world data. The act of digitally sampling the signal immediately introduces some amount of error. To the extent that this error is truly random and has constant statistical properties over time, noise tends to appear as a uniform noise band, or 'fuzz', in the FFT spectrum. This noise affects every value in the spectrum, though it is often more apparent where the spectrum is flattened. Most real phenomena will stand out clearly from the random noise.

Remember that arithmetic errors such as truncation and rounding also appear as noise in the computed transform.

Section II. Reference

17. DAPL Commands

This chapter provides detailed descriptions for all DAPL commands. See the Applications Manual for application examples.

Some commands are available only for certain hardware models. Others have variant forms that are specific to certain hardware models. Look in the descriptions for each individual command for information about hardware dependencies, or check the separate “Features of DAPL dependent on DAP model” document.

DAPL allows certain specific abbreviations for its system commands, the ones that execute immediately and are most likely to be typed in “live.” The abbreviated forms are shown in each relevant command.

The following syntax notation is used to describe DAPL commands:

- Parameters representing numbers or symbol names are enclosed in angle brackets <>.
- Optional parameters are enclosed in square brackets [].
- If a parameter or sequence of parameters can be repeated, it is followed by a star *.
- If several possible command alternatives exist, the alternatives are separated by vertical bars |. The vertical bar should be read as “or.”
- Other notations and explicit keywords are shown as literal text.

The following page uses a fictitious sample command `EXAMPLE` to display the format used to describe the DAPL 2000 commands.

EXAMPLE

A brief command description appears here.

Note: EXAMPLE is a fictitious command used to illustrate how the DAPL 2000 commands are described in this manual. In this sample, the abbreviated form EX may be used instead of EXAMPLE.

EXAMPLE *(<param1>, <param2>)*

EX *(<param1>, <param2>)*

Parameters

<param1>

Description of this parameter.
DATA TYPE FOR PARAM1

<param2>

Output data pipe.
WORD PIPE

Description

This section provides a comprehensive description of what the command does, and the roles served by parameters *<param1>* and *<param2>*.

Example

```
EXAMPLE (P1,P2)
```

The command is illustrated with an instance shown exactly as it would be delivered to the DAPL system, with a detailed description of the specific options selected, input data, and output results.

See Also

Related commands

ABS

Define a task that computes the absolute value of data values.

ABS *(<in_pipe>, <out_pipe>)*

Parameters

<in_pipe>
Source data pipe.
`WORD PIPE | LONG PIPE`

<out_pipe>
Output data pipe.
`WORD PIPE | LONG PIPE`

Description

ABS reads data from *<in_pipe>*, computes absolute values, and places the results in *<out_pipe>*.

Example

```
ABS (P1, P2)
```

Read data from pipe `P1` and place the absolute values into pipe `P2`.

See Also

CABS

ALARM

Define a task that detects values within a region and sets an output bit of the digital output port.

ALARM *(<in_pipe>, <region>, <output_bit>, [<reset>])*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<region>
The region of detection for values.
`REGION`

<output_bit>
A number representing the bit of the digital output port.
`WORD VARIABLE | WORD CONSTANT`

<reset>
A nonnegative integer specifying a time in milliseconds
`WORD CONSTANT`

Description

ALARM reads data values from *<in_pipe>* and determines whether a value is inside *<region>*. If a value inside *<region>* is detected, bit *<output_bit>* of the digital output port is set to one.

Bit 0 is the least significant bit and bit 15 is the most significant bit. *<output_bit>* is a number in the range from 0 to the maximum digital output number supported by the Data Acquisition Processor. When digital output expansion is used, *<output_bit>* is in the range 0 to 15 for digital port `B0`, in the range 16 to 31 for digital expansion port `B1`, etc.

If *<reset>* is present, the digital output bit is reset at least *<reset>* milliseconds after the data values leave the region. The exact time depends on system latency. If *<reset>* is omitted, the output bit is not reset when the data values leave the region and the bit remains set.

During the *<reset>* time, an **ALARM** task flushes data from *<in_pipe>*.

Example

```
ALARM (P1, OUTSIDE,0,100, 5)
```

If a value from pipe `P1` is outside the range 0 to 100, bit 5 of the digital output port is turned `ON`.

See Also

DIGITALOUT, **LIMIT**

AVERAGE

Define a task that computes the arithmetic mean values for groups of samples.

AVERAGE *(<in_pipe>, <count>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE, LONG PIPE, FLOAT PIPE, DOUBLE PIPE`

<count>
The number of samples in each group.
`WORD CONSTANT, LONG CONSTANT`

<out_pipe>
Output pipe for the average values.
`WORD PIPE, LONG PIPE, FLOAT PIPE, DOUBLE PIPE`

Description

AVERAGE computes the arithmetic mean for groups of *<count>* samples. Groups of samples are received from *<in_pipe>* and results are sent to *<out_pipe>*. **AVERAGE** is useful for data compression, noise reduction, and computing statistics from measurements.

The internal summation uses a 64-bit representation. This is large enough that ordinary computations cannot produce an overflow condition. For `DOUBLE` floating point data with exceptionally large values, an intermediate summation exceeding the range representable by the 64-bit `DOUBLE` type results in special numbers `+INF` or `-INF`.

Examples

```
AVERAGE (IPIPE0, 100, P1)
```

Average groups of 100 values from input channel pipe `0` and send the averages to pipe `P1`.

```
AVERAGE (PF4, 4, PFAVG)
```

Average groups of 4 floating point values from pipe `PF4` and send the averages to floating point pipe `PFAVG`.

See Also

BAVERAGE, FIRFILTER, RAVERAGE

BAVERAGE

Define a task that computes multiple arithmetic means for multiplexed data.

BAVERAGE *(<in_pipe>, <n>, <m>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<n>
The number of values in a block.
`WORD CONSTANT`

<m>
The number of blocks of data to be averaged.
`WORD CONSTANT`

<out_pipe>
Output pipe for averaged data blocks.
`WORD PIPE | LONG PIPE`

Description

<n> and *<m>* are positive nonzero integers. *<n>* indicates the number of values in the block. **BAVERAGE** reads *<m>* blocks, averages corresponding points in the blocks, and puts one block of averages into *<out_pipe>*. For every *<n>*<m>* values read, *<n>* values are put into *<out_pipe>*. The maximum size of *<n>* is 8192.

Some common applications of **BAVERAGE** are

- reducing noise in repetitive waveforms,
- averaging data from multiple channels in parallel,
- and reducing noise in FFT power spectra.

Example

```
BAVERAGE (P1, 100, 5, P2)
```

Read 5 blocks of 100 values, average corresponding values in the blocks, and send the 100 averages to pipe `P2`.

See Also

RAVERAGE, **WAIT**

BINTEGRATE

Define a task that uses the trapezoidal method to compute the block integral of data.

BINTEGRATE *(<in_pipe>, <out_pipe>, <blocksize>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<out_pipe>
Output pipe for the value of the integral.
`WORD PIPE | LONG PIPE`

<blocksize>
A number that represents the blocksize.
`WORD CONSTANT`

Description

BINTEGRATE computes the block integral of data in *<in_pipe>* by the trapezoidal method. After each value is used in the integration, the value of the integral is sent to *<out_pipe>*. Each group of *<blocksize>* values is processed independently. The value of the integral is reset after each block.

The value of the integral is computed as half of the first value from *<in_pipe>* plus half of the most recent value from *<in_pipe>*, plus the sum of the other values from *<in_pipe>*. The first data value is consumed from each data block to initialize the integration. To maintain equal input and output data rates, the value zero, corresponding to the lower limit of integration, is placed in *<out_pipe>*, followed by *<blocksize>* - 1 running integral values.

If the integral ever exceeds a signed 31-bit number, **BINTEGRATE** produces erroneous results. In other words, the absolute value of the integral must never exceed approximately 1 billion.

Example

```
BINTEGRATE (P1, P2, 100)
```

Integrate over groups of 100 values from pipe `P1` and reset the integral value to zero after every 100 values.

See Also

INTEGRATE

BMERGE

Define a task that merges equal-sized blocks of similar data.

BMERGE *(<in_pipe_0>, ... , <in_pipe_n-1>, <blocksize>, <out_pipe>)*

Parameters

<in_pipe_0>, ... <in_pipe_n-1>
Input data pipes.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<blocksize>
A number specifying the length of the data blocks.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>
Output pipe for merged data blocks.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

BMERGE reads blocks of length *<blocksize>* sequentially from pipes *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>*, and writes the blocks to *<out_pipe>*. For data that occurs naturally in blocks, such as the outputs of **WAIT** or **FFT** tasks, **BMERGE** is more efficient than **MERGE**.

The data types must be the same for all input and output pipes. The maximum value of *<blocksize>* is 8192.

BMERGE should be used only when *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>*, are filled at the same rate. If different volumes of data arrive in different pipes, data will backlog in the pipes having higher data traffic, causing processing to stall when a pipe has no capacity to accept more data. The **BMERGEF** command should be used when data block arrivals are unbalanced or unpredictable.

Example

```
BMERGE (P1, P2, P3, P4, 1024, $BINOUT)
```

Read blocks of 1024 `WORD` data values sequentially from pipes `P1`, `P2`, `P3`, and `P4`, and send the blocks to the PC through `$BINOUT`.

See Also

BMERGEF, **MERGE**, **MERGEF**, **NMERGE**, **SEPARATE**, **SEPARATEF**

BMERGEF

Define a task that merges blocks of data, adding a flag before each block.

BMERGEF *(<in_pipe_0>, ... , <in_pipe_n-1>, <blocksize>, <out_pipe>)*

Parameters

<in_pipe_0>
First input data pipe.
`WORD PIPE | LONG PIPE`

<in_pipe_n-1>
Last input data pipe.
`WORD PIPE | LONG PIPE`

<blocksize>
A number that represents the length of the data blocks.
`WORD CONSTANT`

<out_pipe>
Output pipe for merged blocks of data.
`WORD PIPE | LONG PIPE`

Description

BMERGEF merges blocks of data from *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>* to *<out_pipe>*, adding an identifying flag before each block. For blocked data, such as the outputs of **WAIT** or **FFT** tasks, **BMERGEF** is more efficient than **MERGEF**.

BMERGEF scans through *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>* sequentially. If a pipe contains at least *<blocksize>* values, **BMERGEF** writes an identifying flag to *<out_pipe>*, then reads *<blocksize>* values and writes those values to *<out_pipe>*. The identifying flag is a number from 0 to *n-1*.

When **BMERGEF** sends data to `$BINOUT`, a PC program should read the first item to determine the source pipe and then read *<blocksize>* values. The size of *<blocksize>* should not exceed 8192 for word pipes and 4096 for long pipes.

Example

```
BMERGEF (P1, P2, P3, P4, 1024, $BINOUT)
```

Read blocks of 1024 data values from pipes `P1`, `P2`, `P3`, and `P4`. As blocks are available, add identifying flags, and send the blocks to the PC.

See Also

BMERGE, **MERGE**, **MERGEF**, **SEPARATEF**

BPRINT

Define a task that sends raw data to the PC in binary format.

BPRINT [(*<in_pipe>*)]

Parameters

<in_pipe>

Optional input data pipe.
WORD PIPE, LONG PIPE, FLOAT PIPE, DOUBLE PIPE

Description

BPRINT sends data to the PC through communication pipe $BINOUT. The command **BPRINT** is used primarily when raw sample data must be transferred to the PC without processing.

When no *<in_pipe>* is specified, **BPRINT** transfers data for all channels sampled by the currently active input sampling configuration. If only a subset of the data channels are needed, *<in_pipe>* can specify an input channel pipe with a limited channel range.

Data of any type can be transferred in this manner, but the $BINOUT pipe must be configured to have the same data type as *<in_pipe>* . For the case of raw sample data, the $BINOUT pipe requires no configuration changes.

Examples

```
BPRINT
```

Transfer all data sampled by the currently active **IDEFINE** configuration to the PC in binary format.

```
BPRINT(IPIPES(0..5))
```

Transfer to the PC the data from the first six input channels sampled by the currently active **IDEFINE** configuration.

See Also

FORMAT, **MERGE**, **PRINT**

CABS

Define a task that computes the sum of the squares of complex data values.

CABS *(<in_pipe1>, <in_pipe2>, <out_pipe> [, <val>])*

Parameters

<in_pipe1>
First input data pipe for real-valued terms.
`WORD PIPE`

<in_pipe2>
Second input data pipe for imaginary-valued terms.
`WORD PIPE`

<out_pipe>
Output data pipe.
`WORD PIPE`

<val>
A value for scaling the sum of the squares.
`WORD CONSTANT`

Description

CABS is an abbreviation for Complex ABsolute value Square. This is different from the complex absolute value function without squaring found in most standard mathematical libraries. If *<in_pipe1>* and *<in_pipe2>* contain the real and imaginary parts of a complex number, *<out_pipe>* contains the scaled square of the amplitude of the complex number.

CABS reads one word value from *<in_pipe1>* and one word value from *<in_pipe2>*. The sum of the squares is placed in *<out_pipe>*. If the *<val>* parameter is specified, the sum of squares is divided by *<val>* before being sent to *<out_pipe>*. This can be used to prevent the sum of squares from overflowing a 16-bit number. *<val>* must be positive.

Note: Mode 4 of **FFT** is more efficient than mode 1 of **FFT** followed by **CABS**.

Example

```
CABS (P1,P2,P3,1000)
```

Read complex values from pipes `P1` and `P2`, compute the sum of squares, divide by 1000, and place the result in pipe `P3`.

See Also

ABS, **FFT**, **POLAR**

CALIBRATE

Calibrate the Data Acquisition Processor input sampling hardware.

CALIBRATE *[<option>]*

Parameter

<option>

Optional keyword to indicate storage mode for calibration results. May be `STORE` or `NOSTORE`.

Description

For Data Acquisition Processor models that support self-calibration, the **`CALIBRATE`** system command issues a DAPL **`RESET`** and then initiates a Data Acquisition Processor hardware calibration session. The calibration session may last a few seconds. While the calibration is in progress, the DAPL interpreter is suspended until the calibration is complete.

When the DAPL operating system is loaded, calibration values are loaded from the onboard nonvolatile memory. If you want to store a new set of calibration values, specify the option `STORE` on the **`CALIBRATE`** command line. This stores calibration values to onboard nonvolatile memory. After calibration, the Data Acquisition Processor will use the computed range and offset adjustments until the next time the Data Acquisition Processor is powered up, or the next time that the DAPL operating system is loaded. If you do not specify any option, the default option is `NOSTORE`. Calibration values will not be stored in onboard nonvolatile memory.

As shipped from the factory, calibration values are already stored on the onboard nonvolatile memory.

The onboard nonvolatile memory allows a limited number of write cycles, on the order of a million. For repeated calibrations, use the `NOSTORE` option.

Example

```
CALIBRATE NOSTORE
```

Perform a self-calibration process, leaving calibration data in active memory but not updating the calibration data image stored in nonvolatile memory.

See Also

RESET

CHANGE

Define a task that scans input data for changes.

CHANGE *(<in_pipe>, <trigger> [, <delta>])*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<trigger>
The trigger that is asserted when a change is detected.
`TRIGGER`

<delta>
An optional number that represents the minimum value of the change.
`WORD CONSTANT`

Description

CHANGE scans input data for changes in numeric value. Every time a change is detected, *<trigger>* is asserted. If the optional numeric parameter, *<delta>*, is present, the absolute value of the change must be greater than *<delta>* before *<trigger>* is asserted.

The default value for *<delta>* is zero, meaning that any change is detected.

Examples

```
CHANGE (IPIPE5, T1)
```

Assert trigger `T1` when the data from input channel pipe 5 change.

```
CHANGE (P1, T2, 1000)
```

Assert trigger `T2` when consecutive data values in pipe `P1` differ by more than 1000.

See Also

DLIMIT, **LIMIT**, **LOGIC**

CHANNELS

Define the number of input channels in an input sampling configuration.

CHANNELS *<nchannels>*

Parameters

<nchannels>

The number of input channels to receive data.
WORD CONSTANT

Description

The **CHANNELS** command configures the number of input channels that will receive input samples for a Data Acquisition Processor that samples individual signal pins sequentially. An **IDEFINE** configuration needs to know the number of channels before other configuration information can be processed, so the **CHANNELS** command should appear as one of the first commands following the **IDEFINE** command.

Ordinarily, all channels are assigned to signal pins by specifying *<nchannels>* number of **SET** commands. Channels reserved by the **CHANNELS** command are sampled and occupy memory whether or not an external signal pin is assigned by a **SET** command.

The **GROUPS** command is similar to the **CHANNELS** command, except that the **GROUPS** command configures sampling for Data Acquisition Processor models that sample multiple signal pins simultaneously.

Examples

```
IDEFINE INP4
   CHANNELS  4
   SET IP0  D0
   SET IP1  D1
   SET IP2  D0
   SET IP3  D2
   TIME 25
END
```

Configure the input sampling to capture data for 4 channels, on a Data Acquisition Processor model that samples signal pins individually. The total number of input data channels is 4. Two channels sample the same signal source.

See Also

SET, **GROUPS**, **IDEFINE**

CLCLOCKING

Set the channel list clocking mode of an input or output configuration.

CLCLOCKING *<switch>*

Parameters

<switch>
A keyword, either `ON` or `OFF`.

Description

CLCLOCKING sets the channel list clocking mode of an input configuration or an output configuration. *<switch>* is `ON` or `OFF`. If the mode is `ON`, a positive clock edge initiates a sampling sequence for a complete channel list. If the mode is `OFF`, **CLCLOCKING** converts a single channel or channel group on the positive edge of the clock. The default value is `ON`. See the Data Acquisition Processor hardware documentation for more information about channel list clocking.

Note: For Data Acquisition Processor models that have simultaneous sampling, all four samples in an input channel group are sampled simultaneously.

Example

```
CLCLOCKING OFF
```

Turn channel list clocking off.

See Also

CLOCK, **HTRIGGER**

CLOCK

Select the source of the input or output configuration sample clock.

CLOCK *<source>*

Description

CLOCK specifies the clock source used by an input configuration or an output configuration. *<source>* is INTERNAL or EXTERNAL. The default *<source>* is INTERNAL. The clock options are described in the hardware documentation.

Example

```
CLOCK EXTERNAL
```

Specify that the clock source is the external clock input pin.

See Also

CLCLOCKING, **HTRIGGER**

COMPRESS

Define a task that encodes data in which changes occur infrequently.

COMPRESS *(<in_pipe>, <n> [, <threshold>]*, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<n>
A positive constant, less than or equal to 16, specifying the number of data streams read from *<in_pipe>*.
`WORD CONSTANT`

<threshold>
A vector, single number or sequence of numbers that represents the threshold values for reporting changes.
`WORD VECTOR | WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for encoded data blocks.
`WORD PIPE`

Description

COMPRESS encodes sample data in which changes occur infrequently. **COMPRESS** can monitor a single data stream or a multiplexed data stream such as an input channel pipe list. **COMPRESS** takes data from *<in_pipe>* one value at a time, and compares that value to the corresponding value previously reported for the same channel. If the absolute difference is greater than or equal to the *<threshold>* parameter value for that channel, a data block containing the new value is placed into *<out_pipe>*. Otherwise the data value is discarded. Output data blocks are always generated to report the first value from each input channel.

The first parameter *<in_pipe>* provides the data. The first parameter typically is an input channel pipe with a channel list, but it can be a user-defined data pipe with a single data channel, or any data pipe with multiplexed `WORD` data in the manner of the **`MERGE`** command.

The second parameter *<n>* is a positive constant, less than or equal to 1024, specifying the number of data channels to read from *<in_pipe>*. If *<in_pipe>* is an input channel pipe list, *<n>* must be equal to the number of channels in the input channel pipe list. For a single channel pipe or user-defined pipe with a data stream that is not multiplexed, *<n>* must be 1. The data streams from the input pipe are numbered 0 through *<n>*-1.

The threshold levels can be specified by a single `WORD` threshold value to be applied to every channel, a vector of `WORD` values with one term in the vector for each input channel, or a sequence of `WORD` constant or variable values with one value for each input channel. Threshold values are specified in sequence, corresponding to increasing channel numbers. Specifying a list of constant or variable values limits the number of channels that the **`COMPRESS`** command can process to 16 or fewer. Each *<threshold>* value must be nonnegative.

Each change report written to *<out_pipe>* contains three values. The first value is a 16-bit integer from 0 to *<n>*-1, specifying the channel for which the level change exceeded the threshold value. A zero indicates the first channel, a 1 indicates the second channel, and so on. The second value is the new `WORD` data value for the indicated channel. The last field is a 32-bit unsigned integer specifying the sample number (timestamp) of the new value. The timestamp starts from 0 and increases by one count for each group of *<n>* sample values read from the input data stream.

Examples

```
COMPRESS (P1, 1, 100, $binout)
```

Send a notification to the PC through the `$binout` pipe each time the level in pipe `P1` changes by more than 100.

```
VECTOR  THRESH  WORD = (64, 64, 1000)
COMPRESS (IPIPES(0,1,2), 3, 64, 64, 1000, P1)
COMPRESS (IPIPES(0,1,2), 3, THRESH, P1)
```

Examine the data from an input channel pipe with three channels, and send a change information block to pipe `P1` each time that the value from input channel 0 or input channel 1 changes by more than 64 or that the value from input channel 2 changes by more than 1000.

See Also

AVERAGE, **LIMIT**, **MERGE**

CONSTANTS

Define a shared constant data element.

CONSTANTS *<name> <type> = <value>*
*[, <name> <type> = <value>]**

CONST *<name> <type> = <value>*
*[, <name> <type> = <value>]**

Parameters

<name>
An assigned name.

<type>
Keyword for data type of the new constant symbol.
`WORD | LONG | FLOAT | DOUBLE`

<value>
The value assigned to the constant.

`WORD CONSTANT`	`WORD VARIABLE`
`LONG CONSTANT`	`LONG VARIABLE`
`FLOAT CONSTANT`	`FLOAT VARIABLE`
`DOUBLE CONSTANT`	`DOUBLE VARIABLE`

Description

CONSTANTS creates symbols for representing numbers that have a consistent meaning in a DAPL configuration. This command can be used to associate an intuitive name with a number used repeatedly in a DAPL command list, allowing all occurrences of the number to be reconfigured by changing only the **CONSTANTS** declaration.

The assigned *<type>* determines the numeric format and precision of the constant. The *<value>* specifier assigns a value to the named constant. Floating point variables and constants cannot be used to assign a value to a `WORD` or `LONG` constant, but otherwise, any constant or variable is acceptable if it provides a value in the representable range.

Unlike **VARIABLES**, for which values shared by tasks can change at any time, the values of **CONSTANTS** do not change while a DAPL configuration runs. While no configurations are active, a constant symbol value can be reconfigured using the **LET**

command. However, this must be done carefully. A constant symbol is evaluated when items using it are initialized. For tasks, the evaluation occurs at task creation, as the configuration starts. For other system elements, the evaluation occurs when the configuration is downloaded to the DAPL system. See the **LET** command for more information.

Note: For compatibility with earlier DAPL system versions, this command will also accept declarations that do not specify a data type. In this case, a `WORD` or `LONG` data type is selected, based on an analysis of the initializer value. The old command form does not support floating point types, and hexadecimal expressions could be interpreted in a manner inconsistent with other DAPL commands. Use of the old command notation should be avoided.

Example

```
CONSTANT NCHANNELS WORD = 12
```

Define the symbol `NCHANNELS` to have the 16-bit value 12.

See Also

VARIABLES, **LET**, **SDISPLAY**

COPY

Define a task that transfers data from an input pipe to one or more output pipes.

COPY *(<in_pipe>, <out_pipe_1>, ... , <out_pipe_n>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe_1> ... <out_pipe_n>
Output pipes for copied data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

COPY transfers each value from *<in_pipe>* to one or more output pipes. As many as 64 output pipes are allowed. Each output pipe receives an independent copy of the original data stream. There are no conversions applied to the data, hence the data types of parameters *<in_pipe>*, *<out_pipe_1>* … *<out_pipe_n>* must all match.

The **COPY** command applies various buffering strategies to transfer as much data as possible with the least overhead. This buffering can result in some increased latency in configurations requiring fast real-time response.

Using **COPY** to generate an independent data stream allows different processing configurations to share the same data. This gets around the restriction that all tasks reading from a pipe must reside in the same processing procedure.

Example

```
COPY (P1, P2, P3, P4)
```

Copy each value from pipe `P1` to pipes `P2`, `P3`, and `P4`.

See Also

COPYVEC, **LCOPY**, **MERGE**

COPYVEC

Define a task that continuously copies data from a vector to a pipe.

COPYVEC *(<vector>, <out_pipe>)*

Parameters

<vector>
Input vector.
`WORD VECTOR | LONG VECTOR`

<out_pipe>
Output data pipe.
`WORD PIPE | LONG PIPE`

Description

COPYVEC continuously copies data from a vector to a pipe. This is used to generate repetitive waveforms for testing or for synchronous output. If the source vector is a long vector, the output pipe must be a long pipe. If the source vector is a word vector, the output pipe can be either a word pipe or a long pipe. Word data are sign extended before being sent to a long output pipe.

Example

```
VECTOR V=(1,2,3,4)
COPYVEC  (V, P)
```

Continuously place copies of the values 1, 2, 3, 4 into pipe `P`.

See Also

COPY, **VECTOR**

CORRELATE

Define a task that computes cross correlation between blocks of data using spectral methods.

CORRELATE *(<p1>, <p2>, <mode>, <n>, <lead>, <lag>, <window>, <p3>)*

Parameters

<p1>
Input data pipe for the first block of samples.
`WORD PIPE`

<p2>
Input data pipe for the second block of samples.
`WORD PIPE`

<mode>
A parameter that is reserved for future expansion, and currently must be zero.
`WORD CONSTANT`

<n>
A parameter specifying the number of degrees of freedom for the correlation estimate, and related to data block length.
`WORD CONSTANT`

<lead>
A parameter specifying the maximum time shift in the position direction, positive number of samples.
`WORD CONSTANT`

<lag>
A parameter specifying the maximum time shift in the negative direction, positive number of samples.
`WORD CONSTANT`

<window>
A parameter that specifies a window operation.
`WORD VECTOR` | window ID number

<p3>
Output pipe for correlation values.
`WORD PIPE`

Description

CORRELATE computes cross correlation between blocks of data from *<p1>* and *<p2>* using spectral methods. Correlation is a way of measuring a degree of similarity between two data streams.

In general, to compute a value of correlation at a time shift of T samples, a block of consecutive values are taken from *<p1>* and an equal sized block of consecutive values are taken from *<p2>*, where the samples in the block from *<p2>* begin T sampling intervals after the block from *<p1>*. If T is positive, the data from *<p2>* is said to lag the data from *<p1>*, and if T is negative, the data from *<p2>* is said to lead the data from *<p1>*. Corresponding pairs of values from the two data blocks are multiplied, and the resulting terms are averaged.

A range for the time shift T is specified by the *<lead>* and *<lag>* parameters, which are both non-negative. The time shift T is adjusted over the range from *<lead>* to *<lag>*, and correlation is computed at each shift. For each block of data from *<p1>*, *<lead>* + *<lag>* + 1 computed correlation values are placed in *<p3>*, starting with the correlation for the greatest lead and continuing to the correlation for the greatest lag.

For computational efficiency, **CORRELATE** is calculated using a fast Fourier transform. For each computation, *<lead>* + *<lag>* + *<n>* values are read from *<p1>* and *<lead>* + *<lag>* + *<n>* values are read from *<p2>*. Because the computation uses the fast Fourier transform, *<lead>* + *<lag>* + *<n>* must be a power of two and must not be larger than the maximum **FFT** size allowed. Also, *<lead>* + *<lag>* must not exceed *<n>*. A larger value of *<n>* can improve statistical significance when important data are spread relatively uniformly through the data blocks, but introduces unnecessary noise into the computations when important data occur in small localized groups.

<mode> is reserved for future expansion, and currently must be zero.

<window> specifies a window operation. The **CORRELATE** command accepts the same pre-defined window types as the **FFT** command. A custom window may also be defined using a **VECTOR**, except that the binary fractions specified in this vector must be 32-bit binary fractions. See the description of the **FFT** command for more information about Fourier transforms and window operations.

A common application of **CORRELATE** is finding the propagation time of a signal through a system. In this case *<p1>* represents the input to the system, and *<p2>* represents the output from the system. The parameter *<lead>* can be set to zero, because the signal cannot arrive before it is sent. Then the lag time at which the correlation is highest represents the time for a signal to propagate from input to output of the system.

Example

```
CORRELATE (P1, P2, 0, 257, 127, 128, 0, P3)
```

Calculate correlations between blocks of 512 points taken from pipes `P1` and `P2`, producing output blocks of 256 points, with the first 127 results for lead, one result for no lead or lag, and the last 128 results for lag. 257+127+128 = 512, so 512 points are consumed from each of pipes `P1` and `P2` to produce each group of 256 output values. Apply no window and place the results in pipe `P3`.

See Also

FFT

COSINEWAVE

Define a task that generates cosine wave data.

COSINEWAVE *(<amplitude>, <period>, <out_pipe>*
[, <mod_type>, <mod1>[, <mod2>]])

Parameters

<amplitude>
A value that is one half of the peak to peak range of the output.
`WORD CONSTANT | WORD VARIABLE`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for cosine wave data.
`WORD PIPE`

<mod_type>
A value that selects amplitude and/or frequency modulation of the output wave.
`WORD CONSTANT`

<mod1>
Pipe for first modulation signal.
`WORD PIPE`

<mod2>
Pipe for second modulation signal.
`WORD PIPE`

Description

COSINEWAVE generates cosine wave data and places the data in *<out_pipe>*. *<period>* is the number of sample values in each wave cycle. The *<amplitude>* is one half the peak to peak distance of the output wave and has a maximum value of 32767.

Note: The **COSINEWAVE** is identical to **SINEWAVE** except for the phase of the signal.

Three optional modulation parameters may be specified. *<mod_type>* selects amplitude and/or frequency modulation of the output wave. The value of *<mod_type>* must be one of the following:

1. amplitude modulation controlled by the data in *<mod1>*
2. frequency modulation controlled by the data in *<mod1>*
3. amplitude and frequency modulation controlled by the data in *<mod1>* and *<mod2>*, respectively

<mod1> and *<mod2>* are pipes. One value is read from the pipe(s) for each value output by **COSINEWAVE**. Modulation values are interpreted as signed binary fractions; they are multiplied by the base amplitude or frequency to obtain the effective amplitude or frequency.

An alternative method for changing the amplitude or frequency of **COSINEWAVE** during execution uses a DAPL variable as the *<amplitude>* or *<period>* parameter of **COSINEWAVE**. This variable can be changed during execution using a **LET** command. This is efficient, but cannot adjust the amplitude or frequency continuously, and changes are detected and applied asynchronously.

Example

```
COSINEWAVE (1000, 100, P2)
```

Generate a cosine wave with values ranging from -1000 to 1000, with a period of 100 samples.

See Also

SAWTOOTH, **SINEWAVE**, **SQUAREWAVE**, **TRIANGLE**, **WAVEFORM**

COUNT

Establish a fixed data block length for an input or output configuration.

COUNT *<sample_count>*

Parameters

<sample_count>

An integer that specifies the number of sampling or updating operations.
`WORD CONSTANT | LONG CONSTANT`

Description

A **COUNT** command in an input configuration definition sets the number of input samples the input configuration acquires. A **COUNT** command in an output configuration definition sets the number of output updates that the output configuration provides. After the specified number of operations, the input or output configuration suspends operation.

<sample_count> specifies the number of samples or updates in a block. Divide *<sample_count>* by the number of input or output channel pipes to obtain the number of times each pin is sampled or updated. *<sample_count>* must be an integral multiple of the number of channel pipes, and must be at least 2. For Data Acquisition Processor models that provide simultaneous sampling of multiple signal lines, the *<sample_count>* must be an integral multiple of the number of channel groups times the sampling group size.

Example

```
COUNT 100000
```

When the input configuration is started, the Data Acquisition Processor acquires 100000 samples and then stops.

See Also

CYCLE, **HTRIGGER**

CROSSPOWER

Define a task that computes a crosspower spectrum for blocks of data.

CROSSPOWER *(<p1>, <p2>, <mode>, <m>, <window>, <p3>, <p4> [, <p5>])*

Parameters

<p1>
An input pipe that represents the input signal from the device under test.
`WORD PIPE`

<p2>
An input pipe that represents the output signal from the device under test.
`WORD PIPE`

<mode>
A parameter that is reserved for future expansion, and must be set to zero.
`WORD CONSTANT`

<m>
A value that represents the size of the transform.
`WORD CONSTANT`

<window>
A value that represents the window vector specification.
`WORD CONSTANT | LONG VECTOR`

<p3>
A output pipe that contains the real part of the crosspower spectrum.
`LONG PIPE`

<p4>
A output pipe that contains the imaginary part of the crosspower spectrum.
`LONG PIPE`

<p5>
An optional pipe that contains the autopower spectrum of *<p1>*.
`LONG PIPE`

Description

CROSSPOWER computes a crosspower spectrum and autopower spectrum for blocks of data. At each frequency, the crosspower spectrum of *<p1>* and *<p2>* is computed

by multiplying the complex conjugate of the value of the FFT of *<p1>* and the value of the FFT of *<p2>*. At each frequency, the autopower spectrum of *<p1>* is computed by multiplying the complex conjugate of the value of the FFT of *<p1>* and the value of the FFT of *<p1>*. The autopower spectrum is just the power spectrum of *<p1>*; this also can be calculated using **FFT**.

Pipes *<p1>* and *<p2>* are inputs to **CROSSPOWER** and typically represent the input signal and the output signal from a device under test. Pipes *<p3>* and *<p4>* are outputs of **CROSSPOWER** and contain the real and imaginary parts of the crosspower spectrum of *<p1>* and *<p2>*. *<p5>* is an optional pipe. This is an output from **CROSSPOWER** that contains the autopower spectrum of *<p1>*. The autopower spectrum always is real.

<mode> is reserved, and must be set to zero.

<m> and *<window>* have the same meaning as for the **FFT** command. *<m>* is the size of the transform. *<window>* is the window vector specification. See the **FFT** command for details. A custom window vector may be specified, but the binary fractions must be in 32-bit binary fraction.

CROSSPOWER is used in conjunction with **TFUNCTION2** for calculating the transfer function of a "black box." The advantage of using crosspower spectrum and autopower spectrum is that these may be averaged to reduce noise.

Example

```
CROSSPOWER (P1, P2, 0, 10, 0, P3, P4, P5)
```

Calculate crosspower spectrum and autopower spectrum of `P1` and `P2` on blocks of 1024 points with a rectangular window. Returns the real and imaginary parts of the crosspower spectrum in `P3` and `P4`, and the autopower spectrum of `P1` in `P5`.

See Also

FFT

CTCOUNT

Define a task that extends a 16-bit event count to 32 bits.

CTCOUNT *(<in_pipe>, <out_pipe>)*

Parameters

<in_pipe>
Input pipe for word data.
`WORD PIPE`

<out_pipe>
Output pipe for long data.
`WORD PIPE | LONG PIPE`

Description

CTCOUNT is used with the Counter Timer Board to convert the 16-bit count provided by a Counter Timer Board into a 32-bit count. This allows DAPL to provide 32-bit counter capabilities using 16-bit counter hardware.

CTCOUNT converts word data to long data, assuming that the input values represent the low-order words of a sequence of non-decreasing long word values. For correct operation, the input to the Counter Timer Board is limited to a frequency with less than 65535 pulses in every period of length `T`, where `T` is the time between successive acquisitions of a counter/timer input.

If *<out_pipe>* is a word pipe, **CTCOUNT** is equivalent to **COPY**.

Example

```
CTCOUNT (IPIPE5, P1)
```

Read data from a Counter Timer Board from input channel pipe 5, convert it to long word counts, and write the results to long pipe `P1.`

See Also

CTRATE

CTRATE

Define a task that computes the arrival rate of timed events.

CTRATE *(<in_pipe>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<out_pipe>
Output data pipe
`WORD PIPE | LONG PIPE`

Description

CTRATE is used with the Counter Timer Board to convert the 16-bit count provided by the Counter Timer Board into differences that represent rate or frequency data.

CTRATE calculates differences of successive event counts, assuming that the input values represent the low-order words of a sequence of non-decreasing long word values. For correct operation, the input to the Counter Timer Board is limited to a frequency that allows the Data Acquisition Processor to determine the correct 32-bit count. This means that the input must have less than 32767 or 65535 pulses in every period of length `T`, depending on whether pipe *<out_pipe>* is a word pipe or a long pipe, where `T` is the time between successive acquisitions of a counter/timer input.

Example

```
CTRATE (IPIPE5, P1)
```

Read data from a Counter Timer Board from input channel pipe 5, convert to frequency data, compute the number of events in each block, and write the results to word pipe `P1.`

See Also

CTCOUNT

CYCLE

Specify that an output configuration generates a repetitive pattern.

CYCLE *<n>*

Parameters

<n>

A value that specifies the number of output values sent to each output channel pipe before repeating.

`WORD CONSTANT | LONG CONSTANT`

Description

CYCLE specifies that an output configuration defines a repetitive pattern. *<n>* specifies the number of output values sent to each output channel pipe before repeating.

<n> can take any value greater than 0 up to the maximum memory space available.

The units of **CYCLE** are different from the units of other output configurations commands, such as **COUNT**. **CYCLE** specifies the number of channel lists processed while **COUNT** specifies the number of channels processed.

Important: A task that writes to an output channel pipe will suspend when the number of samples required for an output **CYCLE** values have been written to the pipe.

Example

```
CYCLE 1024
```

Specify that output waveforms repeat after 1024 values.

See Also

WAVEFORM

DACOUT

Define a task that writes data asynchronously to a digital-to-analog converter.

DACOUT *(<source>, <dac_number>)*

Parameters

<source>

Source pipe or variable.

`WORD PIPE | WORD VARIABLE`

<dac_number>

A value that represents the selected digital-to-analog converter.

`WORD CONSTANT | WORD VARIABLE`

Description

DACOUT reads from *<source>* and writes to a digital-to-analog converter. *<dac_number>* selects a digital-to-analog converter. Valid numbers for onboard DACs are zero and one. If analog output expansion hardware is connected, *<dac_number>* is two for analog output expansion port 0, three for output expansion port 1, and so on.

DACOUT updates digital-to-analog converters asynchronously, so updates do not occur at evenly spaced intervals. Synchronous output updating is provided by output configurations. Note that **DACOUT** must not be used on an output that is actively being used for synchronous output.

Note: The voltages on the digital-to-analog converters remain unchanged after **STOP** and **RESET** commands.

Examples

```
DACOUT (P1, 0)
```

Read values from pipe `P1` and send these values to `DAC 0`.

```
DACOUT (V, 1)
```

Send the value of `V` to `DAC 1` repeatedly so that the output voltage of `DAC 1` tracks the value of `V`.

See Also

DIGITALOUT, **ODEFINE**

DECIBEL

Define a task that converts data into decibel units.

DECIBEL *(<in_pipe>, <reference>, <scale>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<reference>
An input value corresponding to 0 dB.
`WORD CONSTANT | LONG CONSTANT`

<scale>
An integer that represents the scale factor for the output.
`WORD CONSTANT`

<out_pipe>
Output pipe for decibel data.
`WORD PIPE`

Description

DECIBEL converts data from *<in_pipe>* to decibel units. *<reference>* is the input value corresponding to 0 dB. In many applications, this is 32767 for word pipes and 2147483647 for long pipes.

<scale> is a scale factor for the output. *<scale>* must be an integer between 1 and 100, and normally is 1, 10, or 100. With *<scale>* equal to 100, for example, 3.25 dB is output as the number 325. *<scale>* can be used with the decimal point specification of **FORMAT** to produce formatted decibel output with zero, one, or two decimal places of accuracy.

The output of a **DECIBEL** task is given by

```
20 * <scale> * log10 (X/<reference>),
```

where `X` is the value of the input. Note that the output range over all possible inputs and all possible decibel reference values is approximately -186.6 to +186.6 decibels.

Even when *<scale>* equals 100, the computed output always fits into a word output pipe.

The decibel computation is defined only for positive input values. The **DECIBEL** command generates a special output value of -32768 for a negative or zero input value.

Example

```
DECIBEL (P1, 32767, 100, P2)
```

Read values from pipe `P1`, convert data to decibel units (a value of 32767 is 0 dB), multiply decibel units by 100, and write results to pipe `P2`.

DELTA

Define a task that computes differences between successive data values.

DELTA *(<in_pipe>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<out_pipe>
Output pipe for difference data.
`WORD PIPE | LONG PIPE`

Description

DELTA reads from *<in_pipe>*, computes the forward differences of the data, and puts the result values into *<out_pipe>*. The difference is computed by subtracting the previous value from *<in_pipe>* from the current value from *<in_pipe>*. One difference value is generated for each data value read from *<in_pipe>*, with the exception of the first value.

Differences can be considered a low order approximation for a derivative, under suitable scaling to account for the length of the sampling intervals.

Note: Differences between corresponding values in two separate data streams can be computed using DAPL expressions.

Example

```
DELTA (P1,P2)
```

Read data from `P1`, compute the forward difference, and place the result in `P2`.

See Also

INTEGRATE

DEXPAND

Define a task that encodes multiple channels for synchronized output expansion.

DEXPAND *(<in_pipe>, <output_vector>, <out_pipe> [, <type>])*

Parameters

<in_pipe>
Input word pipe.
WORD PIPE

<output_vector>
A vector containing a list of the output pins to which data should be sent.
VECTOR

<out_pipe>
Output channel pipe.
WORD PIPE

<type>
A parameter specifying the type of output expansion board.
WORD CONSTANT

Description

DEXPAND encodes data and address information for transfer to a digital or analog expansion board through the Data Acquisition Processor digital port. *<output_vector>* is a vector containing a list of the expanded output ports to which data should be sent. *<in_pipe>* is a word pipe that contains data to be sent. Data must appear in the order of the list in *<output_vector>*. For each data value read from *<in_pipe>*, four encoded words specifying the output pin number and the data are written to *<out_pipe>*, which is typically an output channel pipe assigned to digital output port B0.

The *<output_vector>* specifies a list of output ports. The port numbers must be within the range 0 through 63 as supported by the expansion boards. See the expansion board manual for more information about port addressing.

The *<type>* parameter specifies the type of output expansion board, either digital or analog. A value of 0 specifies a digital output expansion board. A value of 1 specifies an analog output expansion board. **DEXPAND** responds to the **OPTIONS** BPOUTPUT setting by encoding analog data streams differently for unipolar and

bipolar operation. If the parameter is omitted, both digital and analog output expansion cards will work correctly under the default `BPOUTPUT=ON` option. For safety, the recommended practice is to always specify the *<type>* parameter.

DEXPAND is used only for synchronous output expansion. The **OUTPORT** command configures asynchronous output expansion. Asynchronous output to the digital output port is not available when **DEXPAND** is used.

Note: The encoding generates a data stream in groups of four WORD values. It is possible to stop an output configuration in the middle of a four word output expansion sequence. If another output configuration then is started, the first value written to the expanded output port may be incorrect.

Example

```
OPTIONS  BPOUTPUT=ON
...
DEXPAND(P1, (4, 5, 6, 7), OPIPE0, 1)
```

Prepare multiplexed data from pipe P1 for synchronized analog updating. Encode the data to send it through the digital port to converter ports 4, 5, 6, and 7 on the expansion card. Transfer the data to the Data Acquisition Processor's digital connector through output channel pipe OPIPE0. Specify that the expansion is analog so that the encoding properly takes the bipolar output mode into account.

See Also

DIGITALOUT, **DACOUT**, **ODEFINE**, **OPTIONS**, **OUTPORT**

DIGITALOUT

Define a task that sends data asynchronously to a digital output port.

DIGITALOUT *(<source>, <port_number>)*

Parameters

<source>
Source pipe or variable.
`WORD PIPE | WORD VARIABLE`

<port_number>
The digital output port.
`WORD CONSTANT | WORD VARIABLE`

Description

DIGITALOUT reads data from *<source>* and sends the data to a digital output port. *<port_number>* specifies the digital output port. This parameter should be set to zero to access a Data Acquisition Processor onboard digital output port. If digital output expansion hardware is connected, *<port_number>* may be greater than zero.

DIGITALOUT updates the digital output port asynchronously, so updates do not occur at evenly spaced intervals. Synchronous output is provided by output configurations. Note that **DIGITALOUT** must not be used when synchronous digital output is active.

Note: The values of the digital outputs remain unchanged after **STOP** and **RESET** commands.

Examples

```
DIGITALOUT (P2,0)
```

Read data from pipe `P2` and send the data to the digital output port.

```
DIGITALOUT (V, 0)
```

send the value of word variable `V` to the digital output port repeatedly, so that the output of the digital output port tracks the value of `V`.

See Also

DACOUT, **ODEFINE**

DISPLAY

Display selected system information.

DISPLAY *<item>*

DISP *<item>*

D *<item>*

Parameters

<item>

One of the following keywords:

ALLSYMBOLS	**COMMANDS**	**CPIPES**	**DVARIANT**
EMSG	**ENUM**	**HMEMORY**	**ICOUNT**
MEMORY	**OCOUNT**	**OEMID**	**OPTIONS**
OUTPORT	**OVERFLOWQ**	**PIPES**	**PROCEDURES**
SYMBOLS	**TMEMORY**	**TRIGGERS**	**UNDERFLOWQ**
VARIABLES	**VECTORS**	**WMSG**	**WNUM**

Description

`DISPLAY` prints system information by formatting a brief report and transmitting it through the `$SYSOUT` communication pipe. The *<item>* parameter selects the information to display. Some item names have abbreviations, typically two or three characters.

The following list provides additional information about each of the display options.

`ALLSYMBOLS`

`DISPLAY ALLSYMBOLS` lists information about all user-defined and reserved system names, such as the names assigned to variables and pipes.

`COMMANDS`

`DISPLAY COMMANDS` lists information about the custom commands that have been loaded.

`CPIPES`

`DISPLAY CPIPES` lists information about user-defined and pre-defined communication channel pipes.

DVARIANT

`DISPLAY DVARIANT` displays the variant of the DAPL software system. Most software systems will display `DAPL2000/STANDARD`. Special OEM configurations will display `DAPL2000/KERNEL`.

EMSG

`DISPLAY EMSG` displays the last error message. An empty line is generated if there are no errors recorded in the system error queue. Displaying a software error message clears the error queue, but displaying hardware error messages does not clear it.

ENUM

`DISPLAY ENUM` prints a number indicating whether a system error has occurred since the last `DISPLAY ENUM` command. If the number is zero, no error has occurred. If the number is nonzero, an error has occurred. Nonzero numbers are error codes, which are described in Chapter 18. Displaying the value of `ENUM` resets the error flag to zero. The occurrence of a buffer overflow or underflow does not affect the error flag.

HMEMORY

`DISPLAY HMEMORY` displays the number of bytes of shared heap memory and total available data memory.

ICOUNT

`DISPLAY ICOUNT` prints the current input configuration sample count.

MEMORY

`DISPLAY MEMORY` prints the number of bytes currently in use and the total memory available.

OCOUNT

`DISPLAY OCOUNT` prints the current output configuration update count.

OEMID

`DISPLAY OEMID` displays an OEM identification number. This option is used only for custom OEM versions of DAPL.

OPTIONS

`DISPLAY OPTIONS` prints the state of all of the configuration options selectable using the **`OPTIONS`** command. The options are displayed in a variable number of lines, as a sequence of expressions in the form “keyword=value” separated by a variable number of blanks. The keywords can appear in any order.

OUTPORT

`DISPLAY OUTPORT` prints the current configuration of the output ports.

`OVERFLOWQ`

`DISPLAY OVERFLOWQ` indicates whether a sampling overflow has occurred. The command prints a long integer. If the number is zero, no loss of data has occurred. If the number is nonzero, it is the sample number of the first sample that was lost when the internal buffers overflowed. The number is the sample count from the input configuration most recently started by a **`START`** command. See Chapter 12.

`PIPES`

`DISPLAY PIPES` prints the status of all the defined pipes including each pipe's type and number of items currently stored in the pipe. Note that if several tasks read from a pipe, the number of entries indicates the number of samples that have not yet been processed by all tasks reading data from the pipe.

`PROCEDURES`

`DISPLAY PROCEDURES` lists the names, type, and activity of all input, output, and processing configurations that are defined.

`SYMBOLS`

`DISPLAY SYMBOLS` prints all user-defined names known to the system.

`TMEMORY`

`DISPLAY TMEMORY` (terse memory) gives a condensed version of the **`DISPLAY MEMORY`** information. This command prints a single integer representing the percentage of memory currently in use. If this number is close to 100, almost all available memory is allocated for buffer and data sample storage.

`TRIGGERS`

`DISPLAY TRIGGERS` displays the names, operating modes, and properties of all defined software triggers.

`UNDERFLOWQ`

`DISPLAY UNDERFLOWQ` indicates whether an updating underflow has occurred. The command prints a long integer. If the number is zero, underflow has not occurred. If the number is nonzero, the number is the sample number at which underflow occurred. The number is the update count from the output configuration most recently started by a **`START`** command. See Chapter 12.

`VARIABLES`

`DISPLAY VARIABLES` displays the names, data types and current values of all shared variables.

`VECTORS`

`DISPLAY VECTORS` displays the names, data types and lengths of user-defined vectors.

`WMSG`

`DISPLAY WMSG` prints the last warning message. The message is cleared after it is displayed.

`WNUM`

`DISPLAY WNUM` prints a number indicating whether a system warning has occurred since the last `DISPLAY WNUM` command. If the number is zero, no warning has occurred. If the number is nonzero, a warning has occurred. Nonzero numbers are warning codes, which are described in Chapter 18. Displaying the value of `WNUM` resets the warning flag to zero. The occurrence of a buffer overflow or underflow does not affect the warning flag.

See Also

SDISPLAY

DLIMIT

Define a task that detects data change events.

DLIMIT *(<in_pipe>, <region1>, <trigger> [,<region2>])*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<region1>
A specification for the triggering condition.
`REGION`.

<trigger>
The trigger that is asserted whenever a slope difference satisfying *<region1>* is found.
`TRIGGER`

<region2>
An optional parameter that provides trigger hysteresis.
`REGION`

Description

DLIMIT computes differences between consecutive data values. When a difference satisfies the *<region1>* condition, *<trigger>* is asserted.

<region2> is an optional region specification that provides trigger hysteresis. After the trigger is asserted, no subsequent trigger events are reported while differences between consecutive values satisfy the *<region2>* condition.

Specifying *<region2>* prevents multiple triggering on single events when a signal change covers more than one sample; this is required in most applications. *<region1>* usually is the same as *<region2>*, although this is not required. See Chapter 14 for more details about regions.

The two limit values for a `REGION` can be variable. The location within a data stream where change to a `REGION` variable takes effect is indeterminate because variable changes are not synchronized with task processing. The variable change can appear to be shifted either forward or backward in time by an unpredictable number of sample positions.

Examples

```
DLIMIT (IP5, INSIDE,100,200, T2, INSIDE,100,200)
```

Scan input pipe 5 for differences between 100 and 200; once a trigger is asserted, no triggers are reasserted until a difference not between 100 and 200 is detected.

```
DLIMIT (P1, OUTSIDE -20,20, T1, OUTSIDE -20,20)
```

Scan `P1` for differences less than -20 or greater than 20; after a trigger is asserted, the trigger is not reasserted until a difference less than the range -20 to 20 is detected.

See Also

CHANGE, **LIMIT**, **LOGIC**, **PEAK**

EDIT

Modify input configurations and output configurations.

EDIT *<proc_name>* *<proc_command>*

ED *<proc_name>* *<proc_command>*

Modify com pipes.

EDIT *<cpipe_name>* *<cpipe_parameters>*

ED *<cpipe_name>* *<cpipe_parameters>*

Parameters

<proc_name>
The name of the input or output configuration to change.

<proc_command>
A configuration command as it would appear in an input configuration or output configuration.

<cpipe_name>
The name of the communications pipe to change.

<cpipe_parameters>
Communications pipe configuration parameters as they would appear in a **CPIPE** command.

Description

EDIT modifies an input configuration, output configuration, or communications pipe.

For input or output configurations, the *<proc_command>* has the same form as a configuration command in the original input or output configuration. For input configurations, *<proc_command>* can be a **SET**, **TIME** or **COUNT** command. For output configurations, *<proc_command>* can be a **SET**, **TIME**, **COUNT**, **CYCLE** or **OUTPUTWAIT** command. See the command references pages for information about these individual command types.

For a communications pipe, *<cpipe_parameters>* can be `BLOCKING = <num>` or `WIDTH=LONG|WORD|BYTE|FLOAT`. See the **CPIPE** command for more information about these specifications.

The **EDIT** command can only change a configuration or com pipe when it is inactive. In particular, the data type of a communications pipe can be changed only when the pipe is empty.

Examples

```
EDIT INPR SET IP4 S3 10
```

Change input pipe 4 of configuration `INPR` to input `S3` with a gain of 10.

```
EDIT A TIME 1000
```

Change the update time of output configuration `A` to 1 millisecond per update.

```
EDIT $BINOUT WIDTH=LONG
```

Change the width of `$BINOUT` to long for sending long values to the PC.

See Also

CPIPE, **ERASE**, **IDEFINE**, **ODEFINE**, **RESET**

EMPTY

Flush all data from one or more pipes.

EMPTY *<pipe_name> [, <pipe_name>]**

Parameters

<pipe_name>

The pipe from which data are flushed.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

EMPTY flushes all data from one or more pipes.

Only user-defined pipes and output communications pipes can be emptied. Input communications pipes like `$SYSIN` and `$BININ` cannot be emptied.

Example

```
EMPTY P1,P2
```

Flush pipes `P1` and `P2`.

See Also

ERASE, **RESET**, **STOP**

END

Terminate an input, output, or processing configuration definition.

END

Description

The **END** command terminates a group of statements that define an input sampling, output updating, or processing configuration.

Example

```
PDEFINE A
 ...
END
```

End the definition mode for defining the processing procedure A.

See Also

IDEFINE, **ODEFINE**, **PDEFINE**, **STOP**

ERASE

Delete user-defined symbols.

ERASE *<symbol> [, <symbol>]**

Parameters

<symbol>

User-defined symbol or list of symbols.

Description

ERASE deletes a user-defined symbol or list of symbols so that the symbol or symbols can be redefined. **ERASE** accepts any named symbols except those for communication pipes. When a symbol is deleted, it is removed from memory.

Erasing the symbol for a downloaded command makes the command unavailable. Erasing the module containing one or more downloaded commands removes all commands within that module from memory.

Example

```
ERASE TCOM
```

Remove the trigger pipe named `TCOM`.

See Also

RESET, **EDIT**

EXTRACT

Define a task that extracts a single bit from input data.

EXTRACT *(<in_pipe>, <bit_num>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<bit_num>
A value that selects the bit to extract.
`WORD CONSTANT`

<out_pipe>
Output pipe for the extracted data bit.
`WORD PIPE`

Description

EXTRACT extracts a single bit from the input data. *<bit_num>* selects the bit to extract. Bit 0 is the least significant bit and bit 15 is the most significant bit. Zero or one is placed in *<out_pipe>*, depending whether the extracted bit is zero or one.

<input_pipe> typically contains data from the digital input port.

Similar results can be obtained using a DAPL expression — shift the contents of the input pipe right by *<bit_num>* values and 'AND' the result with the number 1. For example, the following DAPL expression extracts the fifth binary bit from data in pipe P1:

```
P2 = (P1 >> 4) & 1
```

Example

```
EXTRACT (IPIPE5, 7, P1)
```

Read data from input channel pipe 5, extract bit 7 of from each value, and write the result to pipe P1.

FFT

Define a task that calculates fast Fourier transforms of blocks of data.

FFT *(<mode>, <m>, <window>, <pipeinR> [, <pipeinI>] ,*
<pipeoutR> [, <pipeoutI>])

Parameters

<mode>

A number in the range 0 to 6 that selects the operating mode of the transform.
`WORD CONSTANT`

<m>

A number that determines the size of a block of input data.
`WORD CONSTANT`

<window>

A constant or a vector specifying the window operation used by the transform.
`WORD CONSTANT |`
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<pipeinR>,<pipeinI>,<pipeoutR>,<pipeoutI>,

Data input and output pipes, according to the operating mode.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

An **FFT** task calculates a Discrete Fourier Transform of data blocks using a Fast Fourier Transform algorithm.

The internal precision of the calculations depends on the type of input data provided:

- *16-bit fixed point (WORD)* 32 bit internal
- *32-bit fixed point (LONG)* 64 bit internal
- *32-bit floating point (FLOAT)* 64 bit internal
- *64-bit floating point (DOUBLE)* 64 bit internal

The floating point unit (FPU) of the processor is used extensively for `LONG`, `FLOAT` and `DOUBLE` data types. Most applications that need transforms for these data types will need a Data Acquisition Processor model with hardware FPU support. The `FLOAT` and `DOUBLE` data types use the same internal representation, and apply the

same methods, so they differ only in the representation and transfer of their input and output data.

The *<mode>* parameter determines the manner in which the command operates: data requirements for input and output pipes, transform direction, and the post-transform processing applied to the output data. The first four operating modes provide transforms without post-processing operations. The transforms can be applied in the forward direction (time domain to frequency domain) or in the reverse direction (frequency domain to time domain). The differences affect the phase and scaling of terms.

The following table summarizes these modes and their requirements for input and output data pipes.

FFT Modes without Data Post-Processing

Description	*<pipeinR>*	*<pipeinI>*	*<pipeoutR>*	*<pipeoutI>*
Mode 0 (Forward Real in, complex out)	WORD LONG FLOAT DOUBLE		WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE
Mode 1 (Forward Complex in, complex out)	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE
Mode 2 (Reverse Complex in, real out)	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	
Mode 3 (Reverse Complex in, complex out)	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE

The input data are provided in one or two data pipes, *<pipeinR>* and *<pipeinI>*. For most applications, the data derive from measurements of real processes, so only the real-valued data from pipe *<pipeinR>* are needed. For some other applications, *<pipeinI>* is needed to provide imaginary or quadrature terms. Omit parameter *<pipeinI>* when it is not needed.

These modes produce complex output for either real or complex input, with the resulting data blocks equal in length to the input data blocks. However, it is possible that the resulting imaginary terms contain no information when certain data symmetries are present in the complex input data. For this special case, the imaginary output terms can be suppressed with no loss of information using *<mode>=2*. Omit parameter *<pipeoutI>* from the parameter list when it is not needed.

The remaining three operating modes apply post-processing operations that combine terms to make analysis easier. Processed results are returned, rather than the raw transform data. The post-processing operations are meaningful only for the case of forward transforms of real valued inputs, so the parameter *<pipeinI>* is not used. Transforms of real data result in output blocks with symmetry properties such that no new information appears in the second half of the data. To save time and storage, the redundant data are omitted during post processing, and the processed output blocks are half as long as the original input blocks. The output processing options are:

- Magnitude. Terms for each frequency are squared and summed, then the square root of this sum is taken. The magnitude is represented at the same precision as the original data.
- Magnitude and angle. The magnitude term is the same as above. In addition, the ratio of the real and imaginary terms is analyzed to determine the phase angle associated with the frequency.
- Power. Basically, the same thing as magnitude but leaving off the square root operation. Until support for 64 bit integer data is available, special care must be taken to scale `LONG` data so that results are representable in a 32-bit fixed point form.

FFT Modes with Data Post-Processing

Description	*<pipeinR>* *<pipeinI>* (signal)	*<pipeoutR>* (power or magnitude)	*<pipeoutI>* (angle)
Mode 4 (Forward, real in Output = power density)	WORD LONG FLOAT DOUBLE	LONG LONG * FLOAT DOUBLE	
Mode 5 (Forward, real in Output = magnitude)	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	
Mode 6 (Forward, real in Outputs = magnitude and angle)	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE	WORD LONG FLOAT DOUBLE

* Use caution to avoid saturated large values.

For all operating modes, parameter *<m>* indirectly specifies the size of the input data blocks. The lengths of the data blocks are a power of 2, and the actual size is 2 to the power of *<m>* as summarized below.

<m>	*block size*
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384

The *<window>* parameter is a constant or a vector specifying the window operation used by the transform. A window operation is represented by a vector that multiplies

input data term-by-term. For the most common window types, a code number can be entered instead of a vector, and the corresponding vector is constructed automatically. The code numbers for predefined window types are:

`0`	do not use a window (equivalently, use a Rectangular window).
`1`	use a von Hann window.
`2`	use a Hamming window.
`3`	use a Bartlett window.
`4`	use a Blackman window.

When the *<window>* parameter specifies a named `VECTOR` that explicitly defines a window vector, that vector can be coded in any of the supported data types for any input data type. The data types will be converted internally to the appropriate type for run-time processing. For `WORD` window terms, the upper bound integer value 32768 represents the real number 1.0, and smaller integer values proportionally represent fractions. `LONG` window terms are similar, except that the upper bound integer value is 2147483647. For floating point data types, the upper bound value is 1.0, and fractions are represented in a natural notation.

A window is typically used with forward transforms and real-valued data, to reduce stray effects introduced by breaking a continuous sequence of data samples into discrete blocks. Each window type has different effects on spectral resolution, magnitudes of narrow-band peaks, and local power densities. If these distortions are important, they must be taken into account when analyzing the resulting spectrum.

To avoid aliasing, the sampling rate must be chosen so that frequencies of all relevant phenomena are below the Nyquist frequency at term *<n>* / 2 . The spectrum of a real-valued input signal has some special symmetry properties. The term at location 0 is always real-valued, and represents a constant offset (zero frequency) component. For all other terms, the term at *<n>* / 2 + k is equal to the term at *<n>* / 2 - k except for the sign on the imaginary part. In other words, the terms beyond the Nyquist frequency provide no additional spectral information and can be ignored, which is why the post-processing modes return only half-length blocks.

The symmetry phenomenon can lead to some surprises. For example, suppose the input signal is a cosine wave with peak value 10000 and frequency 2 * π * 6 / 256 radians per second, sampled at 1/256 second intervals, and analyzed in a 256 term FFT. One would expect to see a frequency peak of magnitude 10000 in location 6 of the raw transform data, right? Well, one might be in for a surprise. Half of the energy of the cosine wave appears at location 6, but the other half appears in the symmetric image at location 256-6. Each peak would have a value of 7071, representing half of the signal power in the original wave.

Examples

```
FFT (0, 8, 0, P1r, P2r, P2i)
```

Mode 0, forward transform of real data. Read blocks of 256 data values from pipe `P1r`, perform a forward transform, and place the complex results in pipes `P2r` and `P2i`. No window vector is used.

```
FFT (1, 8, 2, P1r, P1i, P2r, P2i)
```

Mode 1, forward transform of complex data. Read blocks of 256 complex data values from pipe `P1r` and `P1i`, perform a forward transform, and place the complex results in pipes `P2r` and `P2i`. A Hamming window is applied to the signal data before the transform.

```
FFT (2, 10, 0, P1r, P1i, P2r)
```

Mode 2, reverse transform of complex data preserving only real terms. Read blocks of 1024 data values from pipes `P1r` and `P1i`, where these data have special symmetry properties. Perform an inverse transform, and place the real results in pipe `P2r`, discarding the meaningless imaginary terms. No window vector is used.

```
FFT (3, 12, 0, P1r, P1i, P2r, P2i)
```

Mode 3, reverse transform of complex data. Read blocks of 4096 data values from pipes `P1r` and `P1i`. Perform an inverse transform, placing the complex results in pipe `P2r` and `P2i`. No window vector is used.

```
FFT (4, 9, 3, P1r, PPow)
```

Mode 4, forward transform of real data converted to power spectrum. Read blocks of 512 data values from pipe `P1r`, perform a forward transform, and place the 256 power spectral density terms in pipe `PPow`. Apply a Blackman window before applying the transform.

```
FFT (5, 8, 1, P1r, PMag)
```

Mode 5, forward transform of real data converted to magnitude spectrum. Read blocks of 256 data values from pipe `P1r`, perform a forward transform, and place the 128 magnitude terms in pipe `PMag`. Apply a von Hann window before applying the transform.

```
FFT (6, 10, 0, P1r, PMag, PPhase)
```

Mode 5, forward transform of real data converted to polar form. Read blocks of 1024 data values from pipe `P1r`, perform a forward transform, and place the 512 magnitude and phase terms in pipes `PMag` and `PPhase`. No window vector is used.

See Also

See Chapter 16 “Fast Fourier Transform” for more information about frequency spectra, sampling, and window operators.

FILL

Add data from a data list to a specified pipe.

FILL *<pipe_name> <data> [[,] <data>]**

F *<pipe_name> <data> [[,] <data>]**

Parameters

<pipe_name>

The pipe to be filled.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<data>

Data to be added to the specified pipe.

`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT`

Description

FILL adds the data in its data list to a specified pipe. **FILL** typically is used to fill a pipe with known data. It can be used for time-shifting a data stream, establishing initial conditions, or to build a known signal waveform.

For **FILL** commands that would exceed the maximum line length, use a sequence of **FILL** commands.

Examples

```
FILL P1 35 70 105 140
```

Place four values into word pipe `P1.`

```
FILL PF1 35  70.5  105.25  14.0175e-12
```

Place four values into floating point pipe `PF1.`

See Also

EMPTY

FINDMAX

Define a task that determines the maximum value in a data range.

FINDMAX *(<in_pipe>, <n>, <region>, <out_pipe1>*
[, <out_pipe2>])

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<n>
The number of values in each block read from *<in_pipe>*.
`WORD CONSTANT`

<region>
The region that determines which data values are scanned.
`REGION` specifier

<out_pipe1>
Output data pipe for the maximum value.
`WORD PIPE`

<out_pipe2>
An optional pipe to which the index of the maximum value is written.
`WORD PIPE`

Description

FINDMAX reads *<n>* values from the *<in_pipe>*. Values are indexed by 0 to n-1. All values whose indices are in *<region>* are scanned. The maximum value in this region is placed in *<out_pipe1>*. If *<out_pipe2>* also is specified, the index of the maximum value is placed in *<out_pipe2>*.

Note: While a `REGION` specification is normally applied to data values, in the **FINDMAX** command it is applied to the index of the data values.

This command can be used after an **FFT** to extract frequency components.

Examples

```
FINDMAX (P1, 128, INSIDE, 50, 100, P2)
```

Read blocks of 128 values from pipe `P1` and send the largest value between locations 50 and 100 to pipe `P2`.

```
FINDMAX (P1, 128, INSIDE, 50, 100, P2, P3)
```

Read blocks of 128 values from pipe `P1` and send the index of the largest value between locations 50 and 100 to pipe `P3` and the largest value to pipe `P2`.

See Also

BAVERAGE, **FFT**

FIRFILTER

Define a task to calculate filtered values using a finite impulse response filter.

FIRFILTER *(<in_pipe>, <coeffs>, <length>, <scale>, <decim>, <phase>, <out_pipe> [,<take>, <skip>])*

Parameters

<pipein>
A data pipe providing a stream of samples to be filtered.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<coeffs>
A vector specifying the filter characteristic.
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<length>
The number of terms in the coefficient vector.
`WORD CONSTANT`

<scale>
A scaling factor applied to each output value.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<decim>
A number that specifies one-out-of-n decimation.
`WORD CONSTANT`

<phase>
A number specifying a time-shift correction.
`WORD CONSTANT`

<out_pipe>
A data pipe where samples of the filtered data stream are placed.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<take>
Number of data values to retain.
`WORD CONSTANT`

<skip>
Number of data values to skip.
`WORD CONSTANT`

Description

FIRFILTER applies a finite impulse response digital filter with *<length>* number of taps to input data from *<in_pipe>*. The vector *<coeffs>* defines the filter characteristic. The elements of *<coeffs>* are multiplied with successive values from *<in_pipe>*, the products are added, and the final sum is divided by *<scale>* to produce each result. One result is retained for each sequence of *<decim>* input values. Optionally, the number of retained results can be further reduced by applying the *<take>* and *<skip>* parameters. Each remaining result is placed into *<out_pipe>*.

The maximum values for the *<length>* parameter depend on the input data type.

- For WORD data, it must be no larger than 1024.
- For LONG, FLOAT, or DOUBLE data, it must be no larger than 32767.

The *<length>* value must correspond exactly to the number of elements in *<coeffs>*. If the special value 0 is used for the *<length>* parameter, the filter length is automatically set equal to the length of the vector. Otherwise, any inconsistency is diagnosed.

The filter produces an output data stream of the same data type as the stream that it receives. The values in the *<coeffs>* vector represent signed binary fractions in a numeric notation appropriate for the input and output data type.

- For WORD data, the coefficient vector must be WORD type. The number 32768 represents the value 1.0, and the filter coefficients in the range –32767 to +32767 proportionally represent fractions less than 1.0 in absolute value. As a rule of thumb, to normalize the filter for unity gain at zero frequency, scale the coefficients so that their sum equals 32767 times the *<scale>* factor. To guarantee freedom from overflow errors, scale the sum of the absolute values of the coefficients to be less than 32767 times *<scale>*.
- For LONG data, the coefficient vector must be LONG type. The number 2147483648 (2 to the 31st power) represents the value 1.0, and the filter coefficients in the range -2147483647 to +2147483647 proportionally represent fractions less than 1.0 in absolute value. As a rule of thumb, to normalize the filter for unity gain at zero frequency, scale the coefficients so that their sum equals 2147483647 times the *<scale>* factor. To guarantee freedom from overflow errors, scale the sum of the absolute values of the coefficients to be less than 2147483647 times *<scale>*.
- For FLOAT or DOUBLE data, the coefficient vector must be matching type. Specify the coefficients in natural decimal notation. Normalize the filter for unity gain at zero frequency by scaling the coefficients so that their sum equals 1.0 times the *<scale>* factor.

Scaling is specified by the *<scale>* factor. When *<length>* is relatively small, a *<scale>* factor equal to 1 is appropriate for all data types. The special value 0 means "use no scaling" and is equivalent to specifying the scale factor 1. When the scale factor is 1, some optimizations are applied for the case of WORD data to achieve the highest filtering rate. For relatively large fixed-point filters, there tend to be many very small terms and rounding errors tend to become significant, so a *<scale>* factor greater than 1 allows the retention of more intermediate precision, provided that all coefficient terms remain representable in the precision used. There are some restrictions on the range of the *<scale>* parameter depending on the input data type.

- For WORD data, *<scale>* must be a power of two no larger than 512, and it should never exceed the first power of two less than *<length>*.
- For LONG data, *<scale>* must be a power of two no larger than 16384.
- For FLOAT or DOUBLE data, *<scale>* can be any appropriate FLOAT or DOUBLE constant.

The decimation factor *<decim>* is a non-negative number less than *<length>*. When *<decim>* is 0 or 1, no decimation is applied, and **FIRFILTER** returns one calculated value for each input sample. When *<decim>* is greater than 1, **FIRFILTER** computes one value and then omits the next *<decim>*-1 values. Normally, decimation is used with lowpass and bandpass filters, because fewer samples are required to represent a signal after high frequencies are removed by filtering.

The optional parameters *<take>* and *<skip>* apply further data reduction. If used, parameters *<take>* and *<skip>* must both be specified. When *<take>* and *<skip>* are specified, **FIRFILTER** retains *<take>* output values, placing them into *<out_pipe>*, and then skips *<skip>* output values. When decimation is specified, *<take>* and *<skip>* are applied to the data remaining after decimation is performed. In a typical application of the *<take>* and *<skip>* parameters, an input signal must be sampled at a very high rate to preserve high frequency information, but the results of the filtering analysis are needed less frequently, for example, to update a Fourier Transform display in a PC graphing program. The data reduction allows the PC to keep up with the data sent by the Data Acquisition Processor, even though the filtering is very fast and could send the PC host everything. Note that equivalent results can be obtained without the *<take>* and *<skip>* parameters, using a separate **SKIP** command, but the **FIRFILTER** command is more efficient because it avoids performing calculations that would eventually be discarded.

When the *<phase>* parameter is specified, the **FIRFILTER** command will replicate the first computed filter value an additional *<phase>* number of times prior to applying decimation. Most but not all applications of the **FIRFILTER** command use

symmetric filters. These filters have desirable phase properties, but they also have the effect of time-shifting the output. The delay is (*<length>*-1)/2 samples for an odd-length filter, or *<length>*/2 samples for an even-length filter. (Sometimes this is described as “linear phase” or “group delay” using terminology from linear filtering theory.) For example, when using a symmetric filter with 41 taps, the first output value calculated by the filter corresponds to the twenty-first input sample, sample number 20. If it is important to maintain sample count synchronization for this example, the *<phase>* parameter should be set to 20. The **FIRFILTER** command automatically computes the appropriate correction for a symmetric filter if the *<phase>* parameter is set to the special number -1. If a time shift adjustment is not important, the *<phase>* parameter should be set to 0. For filter designs that are not symmetric, some other appropriate *<phase>* value can be specified to compensate for the time delay to the nearest sample position. The phase corrections do not make results appear any sooner in real time, but they make identification of features in time sequences easier, because positions in the input and output data streams correspond.

An alternative to using a non-zero *<phase>* parameter is to use the **FILL** command to place the required number of extra samples into the *<out_pipe>* prior to starting the configuration that defines the **FIRFILTER** task. Arbitrary fill values, such as zeros, can be used.

The program FGEN can be used to generate symmetric filter data for common filter types. FGEN has an option for calculating scaled or unscaled filter vectors of all data types.

The filtering algorithms in the **FIRFILTER** command apply special optimizations when the filter has a symmetry property, when there is no scaling, and when decimation or data reduction operations are applied. The selected options will affect the maximum throughput rate.

Examples

```
PIPE   P1 WORD
FIRFILTER (IPIPE5, VEC1, 41, 0, 0, -1, P1)
```

Apply the symmetric filter defined by vector `VEC1` to data from input channel pipe 5, with 41 stages in the filter; apply no scaling factor or decimation; apply an initial time shift correction computed automatically from the filter length; and send all results to `WORD` pipe `P1`.

```
PIPE   PF1 FLOAT, PF2 FLOAT
FIRFILTER (PF1, FVEC2, 0, .7071, 4, 0, PF2, 1024, 4096)
```

Apply a filter of arbitrary length to data from floating point pipe `PF1`, using filter coefficients provided by the `FLOAT` vector `FVEC2`; apply a scaling factor of 0.7071 to each result, retain every fourth result; use no initial time-shift correction; send 1024 of the retained results to floating point pipe `PF2`, then ignore the next 4096 results, and repeat.

See Also

FIRLOWPASS, **RAVERAGE**

FIRLOWPASS

Define a command that applies a pre-defined lowpass FIR filter with decimation.

FIRLOWPASS *(<in_pipe>, <d>, <out_pipe>)*

Parameters

<in_pipe>

The pipe from which data values are read.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<d>

A number that selects the filter vector and decimation factor used.

`WORD CONSTANT`

<out_pipe>

The pipe to which filtered data values are written.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

The **FIRLOWPASS** command combines high-precision lowpass filtering with a decimation operation. **FIRLOWPASS** is a special case of the **FIRFILTER** command using pre-defined filter characteristics. The filtering is applied to the sampled signal stream from *<in_pipe>*, and the filtered data stream is placed into *<out_pipe>*. The value of decimation factor*<d>* must be in the range 2 through 12. An appropriate filter characteristic is provided automatically according to the signal data type and the decimation factor. Data types of the input and output data pipes must match.

FIRLOWPASS is useful when filtering must completely eliminate high frequency noise, while exactly preserving low frequency information. It is suitable for anti-aliasing applications, but the aggressive elimination of high frequencies is sometimes more than is strictly necessary for anti-aliasing alone.

After the **FIRLOWPASS** filtering operation is applied, the original sampling rate is higher than necessary to completely represent the filtered signal. Decimation eliminates part of this redundancy, retaining one sample from each group of *<d>* samples. The decimation causes no loss of information.

The filtering accuracy depends on the data type.

- `WORD` data stream: accuracy to 14 bits.

- LONG, FLOAT, or DOUBLE data stream: accuracy to 18 bits.

For example, 14 bit accuracy means that for a bipolar converter spanning the input range -8192 counts to +8192 counts, the output value is off by at most one converter count. When represented in the form of a 16 bit number, each 14-bit converter count yields an increment of 4. Stopband output levels of +4 peak for input levels of 32768 peak correspond to 78.3 dB rejection. A passband peak output level 32764, differing from the correct peak input level of 32768 by one converter count, corresponds to a passband gain error of 0.00104 dB.

The predefined filter characteristics are optimized to have the following properties:

- The magnitudes of frequencies up to 25% of the new Nyquist frequency (through 12.5% / *<d>* of the new sampling frequency) are preserved to full filtering accuracy.
- The magnitudes of frequencies beyond 75% of the new Nyquist frequency (beyond 37.5% / *<d>* of the new sampling frequency) are eliminated to full filtering accuracy.

When perfect filter gain is not critical, useful signal information can be obtained past the 25% absolute flat band. For example, the signal loss is 0.1 dB (about 1%) at approximately 30% of the Nyquist frequency, and the –3 dB cutoff frequency is at approximately 42% of the Nyquist frequency.

Because of the symmetry property of the FIR filtering characteristic, phase shift is exactly zero for all frequencies, but there is a time lag for the delivery of the filtered results. This can be interpreted in the frequency domain as a constant group delay.

The following table provides information about filter bands and delays for each decimation level and data type.

Deci-mation	Passband Limit (% Nyquist)	Stopband Limit (% Nyquist)	Data Type	Total taps	Delay samples (after decimation)
(after)	50	75		------	------
2	25	37.5	`Word`	37	9
			`Long Float Double`	53	13
3	16.67	25	`Word`	57	9.33
			`Long Float Double`	79	13
4	12.5	18.75	`Word`	75	9.25
			`Long Float Double`	105	13
5	10	15	`Word`	93	9.2
			`Long Float Double`	129	12.8
6	8.33	12.5	`Word`	113	9.33
			`Long Float Double`	155	12.83
7	7.14	10.71	`Word`	131	9.28
			`Long Float Double`	181	12.85
8	6.25	9.38	`Word`	149	9.25
			`Long Float Double`	207	12.87

Deci-mation	Passband Limit (% Nyquist)	Stopband Limit (% Nyquist)	Data Type	Total taps	Delay samples (after decimation)
9	5.56	8.33	`Word`	169	9.33
			`Long Float Double`	233	12.89
10	5	7.5	`Word`	189	9.4
			`Long Float Double`	259	12.9
11	4.55	6.82	`Word`	205	9.27
			`Long Float Double`	283	12.81
12	4.17	6.25	`Word`	225	9.33
			`Long Float Double`	309	12.83

Example

```
FIRLOWPASS (IP3, 10, P)
```

Apply a lowpass filter to the data in input channel pipe `IP3` and decimate by placing every tenth result into pipe `P`.

See Also

FIRFILTER, **RAVERAGE**

FORMAT

Define a task that sends formatted text data to the PC.

FORMAT *[COUNT=<num>] [OUTPUT=<cpipe>] [HEX]*
(<item> [,<item>])*

<item> = <string> | <operator> | <numeric> [: <precision>]

<operator> = # | ## | /

<numeric> = <pipe> | <variable> | <constant>

Parameters

<num>
A positive decimal number limiting the number of lines to print.
`WORD CONSTANT`

<cpipe>
Communication text pipe name.

<string>
Arbitrary text enclosed in double-quotes.
`STRING`

<precision>
Formatting expression (see description below).

<pipe>
Pipe providing data to display.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<variable>
Variable providing data to display.
`WORD VARIABLE | LONG VARIABLE | FLOAT VARIABLE | DOUBLE VARIABLE`

<constant>
Named or numeric constant value to display.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

Description

FORMAT builds formatted print lines and sends the text to the PC. A binary data transfer is more efficient and best for most finished applications; but **FORMAT** is very useful for quick tabulated listings during application testing and development

FORMAT allows some complicated parameter sequences, but in its simplest form, it requires only a list of pipe names—for example:

```
FORMAT (P1, P2)
```

When a **FORMAT** task is active, it looks for new data from each data source in sequence. The values of constants and variables are always available, but **FORMAT** must sometimes wait for data to appear in data pipes. **FORMAT** takes a value from each data source, then prints the set of values on a single line.

FORMAT is unusual in its handling of data pipes. For all other commands, each reference to a pipe produces a separate copy of the entire data stream. For the **FORMAT** command, multiple references to a data source result in items being taken in sequence from one stream. If, for example, the data sequence *1, 2, 3, 4, 5* etc. is loaded into pipe `P1`, the task

```
FORMAT( P1, P1, P1, P1)
```

will yield the display lines

```
1       2       3       4
5       6       7       8
9      10      11      12
etc.
```

and *not* the display lines

```
1       1       1       1
2       2       2       2
3       3       3       3
etc.
```

Avoid multiple references to an input channel pipe with a channel list. These are confusing, because each reference extracts data for a different channel.

The `COUNT`, `OUTPUT` and `HEX` options can provide additional control over **FORMAT** command operation. Usually these options are omitted.

- If a `COUNT` option is specified, the **FORMAT** task delivers *<num>* lines and then terminates. If `COUNT` is omitted, the **FORMAT** task prints until stopped by a **STOP** or **RESET** command. `COUNT` is useful for retaining just a few items when otherwise excessive amounts of data would be generated. Beware, after the `COUNT` condition is satisfied, the task suspends and does not read new data. This could result in a data backlog that forces other processing to stop.

- Use the OUTPUT option to send the formatted lines to an alternative communication pipe *<cpipe>* rather than the default $SYSOUT pipe.
- If the HEX keyword option is specified, fixed-point values are displayed using a hexadecimal notation. No *<precision>* expressions can be applied to hex integer formats.

There is no formal limit on the number of items in the **FORMAT** parameter list, but there is a practical limit. The maximum number of characters in a line is 236. If lines are too long, they can be split into multiple **FORMAT** tasks, or use the slash operator code to break the lines into shorter pieces. The DAPL system will keep print lines from multiple **FORMAT** tasks intact, but the order in which these lines are delivered is unpredictable.

Besides pipes, scalar values can also appear in a **FORMAT** command parameter list. The most current value of a variable is fetched as needed. Avoid displaying values of variables or constants without any pipes. Because there are no delays, a huge number of lines can be generated.

Strings are sometimes useful for labels or for interjecting separator characters. The text from the string is copied into the formatted lines in the position where the string appears in the parameter list. Enclose the text with double-quote characters.

The *<operator>* notations provide some additional line-formatting options. Place the *#*, *##*, or */* operator into the parameter list in the same manner as other parameter list items, including the comma separators.

- *#* causes **FORMAT** to generate a line number in a 16-bit decimal notation.
- *##* causes **FORMAT** to generate a line number in a 32-bit decimal notation.
- / causes **FORMAT** to split subsequent items onto a new text line. This new line is not counted as a separate line for purposes of the COUNT option.

Display of numeric parameter items can be modified using a *<precision>* notation. The notation is separated from the pipe, variable or constant parameter by a colon character. The following precision notations are supported for fixed point data:

- for WORD values, the precision notation is a decimal integer in the range of 0 through 5.
- for LONG values, the precision notation is a decimal integer in the range of 0 through 14.

The decimal number specifies a position, counting from the right, where a decimal point is to be inserted in the sequence of digits. If the number is too small, fill zeroes

are also inserted. This looks like scaling by a power of 10, but the original value is not changed.

- for floating point values, the precision notation is a prefix letter followed by a decimal integer in the range of 0 through 14. The prefix letter E means "display with a power-of-ten exponent notation." The prefix letter F means "display with a fixed fraction notation." (These codes are similar to the `%e` and `%f` formatting codes in the C programming language.) The number specifies the number of digits to appear after the decimal point.

Examples

For these examples, presume that `VT` is a floating point variable containing the value 66.1, `P1` is a pipe containing word data sequence 10, 20, 30, etc., and that `PF2` is a pipe containing floating point data sequence 12.345, 23.456, 34.567, etc. The lines generated by each example are illustrated below the command line.

```
FORMAT (#,P1,PF2)
    0       10          12.35
    1       20          23.46
    2       30          34.57
...
```

Display the line number and data values from pipes `P1` and `PF2` using default formats and options.

```
FORMAT COUNT=2 ("  ITEM",##,P1:3,PF2:F3)
  ITEM          0     .010       12.345
  ITEM          1     .020       23.456
```

Display line counts and values from pipes P1 and PF2, formatting both values to show three digits after a decimal point, and ending the listing after two lines.

```
FORMAT ("  MEASUREMENT ",P1:4,"  WITH OFFSET ",VT:F1)
  MEASUREMENT  .0010  WITH OFFSET      66.1
  MEASUREMENT  .0020  WITH OFFSET      66.1
...
```

Display labeled word values from pipes `P1` with inserted decimal point preceding four digits, and show corresponding values from variable `VT` using a fixed fraction notation.

```
FORMAT HEX (P1, /, “          “, PF2:E5)
   000A
          1.23450E1
   0014
          2.34560E1
   001E
          3.45670E1
...
```

Display fixed point values using a hexadecimal notation, and floating point values on a separate offset line using an exponent notation.

See Also

BPRINT, **PRINT**, **MERGE**

FREQUENCY

Define a task that determines the number of trigger assertions per block of samples.

FREQUENCY *(<trigger>, <length>, <out_pipe>)*

Parameters

<trigger>
The trigger being examined.
`TRIGGER`

<length>
A value that specifies the size of the sample block.
`WORD CONSTANT`

<out_pipe>
The pipe to which the number of assertions is written.
`WORD PIPE`

Description

FREQUENCY determines how many trigger assertions occur in each block of *<length>* samples. The number of assertions is sent to *<out_pipe>*, one number for each input block.

Example

```
FREQUENCY (T1, 100, P1)
```

Send the number of trigger assertions occurring in every block of 100 samples to pipe `P1`.

See Also

LIMIT, **LOGIC**, **PULSECOUNT**, **CTRATE**

GROUPS

Define the number of channel groups in an input sampling configuration.

GROUPS *<ngroups>*

Parameters

<ngroups>

The number of input channel groups to receive data.
WORD CONSTANT

Description

The **GROUPS** command configures the number of input channel groups that will receive input samples for a Data Acquisition Processor that samples multiple signal pins simultaneously. The total number of data channels is*<ngroups>* times the group size. The group size is determined by each individual Data Acquisition Processor model. See the Data Acquisition Processor hardware manual for information about channel group sizes.

The **CHANNELS** command is similar to the **GROUPS** command, except that the **CHANNELS** command implies single channel input sampling while **GROUPS** implies multiple channel simultaneous sampling.

An **IDEFINE** configuration should include *<ngroups>* number of **SET** command lines, each defining the characteristics of one channel group. Because the **IDEFINE** configuration needs to know the number of channel groups before other configuration information can be processed, the **GROUPS** command should appear as one of the first commands following the **IDEFINE** command.

Examples

```
IDEFINE INP3
   GROUPS  3
   SET IP(0..3)  SPG0
   SET IP(4..7)  SPG1
   SET IP(8..11) SPG2
   TIME 20
END
```

Configure the input sampling to capture data for 3 channel groups, on a Data Acquisition Processor model that samples four pins simultaneously. The total number of input data channels is 12.

See Also

SET, **CHANNELS**, **IDEFINE**

GROUPSIZE

Define the number of channels in a programmable input channel group.

GROUPSIZE *<size>*

Parameters

<size>

The number of input channels in a configurable input channel group.
WORD CONSTANT

Description

The **GROUPSIZE** command is available for Data Acquisition Processors that support software-configurable channel group sizes. The *<size>* parameter specifies the number of channels in each input channel group. Only certain restricted sizes are available, so see the Data Acquisition Processor hardware manual for information about supported channel group sizes.

If the **GROUPSIZE** command is omitted, a default group size is assumed. For information about default group size, see the Data Acquisition Processor hardware manual.

Because the **IDEFINE** configuration needs to know the channel group size before other configuration information can be processed, the **GROUPSIZE** command should appear as one of the first commands following the **IDEFINE** command.

Examples

```
IDEFINE INP2X4
   GROUPS  2
   GROUPSIZE 4
   SET IP(0..3)  SPG0
   SET IP(4..7)  SPG1
   TIME 20
END
```

Configure the input sampling to capture data for 2 channel groups with configurable group size 4, on a Data Acquisition Processor model that supports configurable

simultaneous sampling in groups of 4 pins. The total number of input data channels is 8.

See Also

GROUPS, **IDEFINE**

HELLO

Display identification information.

HELLO

Description

Prints Data Acquisition Processor hardware and software information. A single line is printed in the following format:

```
*** DAPL2000 Interpreter [2.0 T1/212] Serial# 43000 ***
```

`2.0` is the DAPL version number. `T` is the Data Acquisition Processor product code. `1` is the hardware version code. `212` is the model number. `43000` is the serial number of the Data Acquisition Processor.

Note: Other configuration information also may be displayed near the end of the information line.

HIGH

Define a task that scans blocks of data for maximum values.

HIGH *(<in_pipe>, <count>, <out_pipe1> [,<out_pipe2>])*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<count>
A value that specifies the size of blocks to be scanned.
`WORD CONSTANT`

<out_pipe1>
Output data pipe for maximum block values.
`WORD PIPE | LONG PIPE`

<out_pipe2>
If specified, the pipe to which the sample number of each maximum value is placed.
`WORD PIPE`

Description

HIGH scans blocks of size *<count>* for maximum values. The maximum of each block is placed in the pipe*<out_pipe1>*. If pipe*<out_pipe2>* is specified, the sample number of each maximum value is placed in *<out_pipe2>*. The sample number has a value between 0 and *<count>*-1.

When there are multiple values equal to the same maximum, the location of the first maximum is reported.

Examples

```
HIGH (IPIPE3, 100, P1)
```

Read blocks of 100 values from input channel pipe 3 and send the maximum data value in each block to pipe `P1.`

```
HIGH (P2, 1000, P3, P4)
```

Read blocks of 1000 values from pipe `P2`, send the maximum data value in each block to pipe `P3` and send the position of the maximum in the block to pipe `P4`.

See Also

FINDMAX, **LOW**, **PEAK**, **RANGE**, **VARIANCE**

HTRIGGER

Specify the operating mode of the hardware trigger.

HTRIGGER *<type>*

Parameters

<type>

A keyword, either `ONESHOT`, `GATED`, or `OFF`.

Description

`HTRIGGER` specifies the operating mode of the hardware trigger. Possible values of *<type>* are `ONESHOT`, `GATED`, and `OFF`. The default is `OFF`. The hardware trigger is described in the hardware documentation.

Example

```
HTRIGGER GATED
```

Specify that the hardware trigger allows sampling when the trigger level is high.

See Also

`CLCLOCKING`, **`CLOCK`**

IDEFINE

Define and configure input sampling.

IDEFINE *<name>*

IDEF *<name>*

Parameters

<name>

A unique name assigned to the input configuration.
Alphanumeric character sequence limited to 23 characters.

Description

IDEFINE begins a group of commands that define an input sampling configuration.

```
IDEFINE <name>
  [input configuration command] *
END
```

<name> is a unique name given to the input configuration. *<name>* must be an alphanumeric sequence with no spaces and is limited to 23 characters or less.

The **END** command completes the configuration started by the **IDEFINE** command, making the configuration available for execution.

The complete list of input configuration commands that can appear between the **IDEFINE** and the **END** command is given in Chapter 5. These commands establish the number of channels, configure input gains, specify sample time intervals, and so forth.

Older notations allow a number on the **IDEFINE** command line following the *<name>* field. This old notation is an alternative to using a **GROUPS** or **CHANNELS** command to configure the number of sampled pins or pin groups. However, various new hardware and software features will not be available when old notations are used.

Examples

```
// A configuration for a DAP 5400a/627
IDEFINE INP16
   GROUPS  2
   SET IP(0..7)  SPG0
   SET IP(8..15) SPG1
   TIME 5
END
```

Begin the definition of an input configuration named `INP16` with 2 input channel groups, for a total of 16 input channels.

```
// A configuration for a DAP 4000a/212
IDEFINE INP4
   CHANNELS  4
   SET IP0 SP0
   SET IP1 SP1
   SET IP2 SP0
   SET IP3 SP2
   TIME 25
END
```

Begin the definition of an input configuration named `INP4` with 4 individual input channels sampling three distinct signal sources.

See Also

END, **SET**, **VRANGE**, **CHANNELS**, **GROUPS**, **ODEFINE**, **PDEFINE**

INTEGRATE

Define a task that computes the integral of data by the trapezoidal method.

INTEGRATE *(<in_pipe>, <out_pipe> [, <reset>])*

Parameters

<in_pipe>
The pipe from which data are read.
`WORD PIPE | LONG PIPE`

<out_pipe>
The pipe to which the integral values of data are written.
`WORD PIPE | LONG PIPE`

<reset>
`TRIGGER | WORD CONSTANT`

Description

INTEGRATE computes the integral of data in *<in_pipe>* by the trapezoidal method. After each value is read, the integral value is sent to *<out_pipe>*.

The integral value is computed as half of the first value from *<in_pipe>* plus half of the most recent value from *<in_pipe>*, plus the sum of the other values from *<in_pipe>*. **INTEGRATE** sends one less value to *<out_pipe>* than it reads from *<in_pipe>* because one value is consumed in the integral computation.

The integral magnitude must never exceed a signed 31-bit number or approximately 2.1 billion to provide accurate results.

An optional parameter, *<reset>*, can be used to prevent overflow of the 32-bit integral value. If a trigger is used, the value of the integral is reset to zero whenever a trigger assertion occurs. If a constant is used, the value of the integral is reset to zero after every *<reset>* number of values. Each time the integral is reset, **INTEGRATE** uses the last input data value from the previous integration to initialize the new integration.

Example

```
INTEGRATE (P1, P2)
```

Read data from pipe `P1` and place the integral in pipe `P2.`

```
INTEGRATE (P1, P2, 100)
```

Reset the integral value to zero after every 100 values.

See Also

BINTEGRATE, **DELTA**

INTERP

Define a task that performs linear interpolation of a function.

INTERP *(<in_pipe>, <x_vector>, <y_vector>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<x_vector>
A list of abscissa (input) values.
`WORD VECTOR`

<y_vector>
A list of ordinate (output) values.
`WORD VECTOR`

<out_pipe>
Output data pipe.
`WORD PIPE`

Description

INTERP performs linear interpolation of an arbitrary function represented by a lookup table. This command is especially useful for sensor linearization.

<x_vector> and *<y_vector>* represent a set of ordered pairs of `WORD` numbers *(x,y)* that describe a function. The numbers of elements in *<x_vector>* and *<y_vector>* must be the same and the values in the *<x_vector>* must be in a strictly ascending sequence. For each value in *<in_pipe>*, an **INTERP** task searches for the nearest values in *<x_vector>*, interpolates between the corresponding values of *<y_vector>*, and places the result in *<out_pipe>*.

INTERP returns the first *<y_vector>* value for values from *<in_pipe>* that are less than the smallest value in *<x_vector>*. **INTERP** returns the last *<y_vector>* value for values from *<in_pipe>* that are greater than the largest value in *<x_vector>*.

Example

```
VECTOR X WORD = (-32768,0,32767)
VECTOR Y WORD = (-100,0,200)
INTERP (P1, X, Y, P2)
```

Read data from pipe `P1`, interpolate according to vectors `X` and `Y`, and send the results to pipe `P2`.

See Also

THERMO

LCOPY

Define a task that transfers data from an input pipe to one or more output pipes.

LCOPY *(<in_pipe>, <p1> [, <p2>]*)*

Parameters

<in_pipe>
The pipe from which data values are read.
`WORD PIPE`

<p1>
The first output pipe to which the transferred data are written.
`WORD PIPE`

<p2>
The second output pipe to which the transferred data are written.
`WORD PIPE`

Description

LCOPY transfers each value from *<in_pipe>* to one or more output pipes. As many as 64 output pipes are allowed.

Unlike **COPY**, **LCOPY** guarantees to transfer one value from *<in_pipe>* to each output pipe before reading the next value from *<in_pipe>*. As a result, **LCOPY** provides lower latency than **COPY** does at the expense of a decreased throughput.

See Also

COPY, **SKIP**

LET

Change the value of a variable or constant.

LET *<sym_name> = <value>*

Parameters

<sym_name>

Name of the variable or constant symbol whose value is being changed.

```
WORD CONSTANT   | WORD VARIABLE   |
LONG CONSTANT   | LONG VARIABLE   |
FLOAT CONSTANT  | FLOAT VARIABLE  |
DOUBLE CONSTANT | DOUBLE VARIABLE
```

<value>

The new value to assign to the variable or constant symbol.

```
WORD CONSTANT   | WORD VARIABLE   |
LONG CONSTANT   | LONG VARIABLE   |
FLOAT CONSTANT  | FLOAT VARIABLE  |
DOUBLE CONSTANT | DOUBLE VARIABLE
```

Description

LET changes the value of a variable or constant symbol. Floating point variables and constants cannot be used to assign a value to a `WORD` or `LONG` symbol, but otherwise, any constant or variable is acceptable if it provides a value in the representable range. The *<sym_name>* symbol name must be defined previously by a **VARIABLES** or **CONSTANTS** command.

When the **LET** command is used to change a constant symbol, no configurations can be active. A **LET** command can change a variable symbol at any time.

Reconfiguring a constant symbol using the **LET** command must be done with care. A constant value assigned by the **LET** command is evaluated when items using the symbol are initialized. For tasks, the evaluation occurs at task creation, as the configuration starts. For other system elements, the evaluation occurs when the configuration is downloaded to the DAPL system, as the commands are interpreted and executed. For example, if a downloaded **IDEFINE** command evaluates the number of data channels from a named constant, changing the named constant value later does not change the number of channels in the sampling configuration. An **EDIT** command can be used to change configuration parameters for sampling and update procedures.

Examples

```
LET SPEED=152.5
```

Set the value of floating point symbol `SPEED` to 152.5.

```
LET N = M
```

Change the value of symbol `N` to the current value of symbol `M`.

See Also

CONSTANTS, **EDIT**, **SDISPLAY**, **VARIABLES**

LIMIT

Define a task that scans data for values in a specified region.

LIMIT *(<in_pipe>, <region1>, <trigger> [,<region2>])*

Parameters

<in_pipe>
WORD PIPE

<region1>
The region for detecting trigger events.
REGION

<trigger>
The trigger asserted when values within a specified region are found.
TRIGGER

<region2>
An optional hysteresis region for suppressing subsequent trigger events.
REGION

Description

A **LIMIT** task scans input data for values that satisfy *<region1>*. When such a value is found, *<trigger>* is asserted.

An optional second region, *<region2>*, can be specified for trigger hysteresis. After a *<trigger>* event is asserted, subsequent data are ignored until *<region2>* is satisfied. Then, the scan for values that satisfy *<region1>* resumes.

Specifying *<region2>* prevents multiple triggering on slow or tightly-clustered events; this is required in most applications. *<region1>* usually is the same as *<region2>*, although this is not required.

The two limit values for a REGION can be variable. The location within a data stream where change to a REGION variable takes effect is indeterminate because variable changes are not synchronized with task processing. The variable change can appear to be shifted either forward or backward in time by an unpredictable number of sample positions.

Examples

```
LIMIT (IPIPE4, INSIDE,5500,5600, T1, INSIDE,5500,5600)
```

Read data from input channel pipe 4 and scan for values from 5500 to 5600; assert trigger `T1` whenever one of those values is found; after an assertion, do not trigger again until a value not from 5500 to 5600 is found.

```
LIMIT (P1, OUTSIDE,-20000,20000, T2, OUTSIDE,-20000,20000)
```

Scan data in pipe `P1` for values less than -20000 or greater than 20000 and assert trigger `T2` when one of those values is found; after an assertion, `T2` will not trigger again until a value from -20000 to 20000 is found.

See Also

CHANGE, **DLIMIT**, **LOGIC**, **PEAK**

LOGIC

Define a task that asserts a trigger when data bits match a specified pattern.

LOGIC *(<in_pipe>, <xor>, <and>, <select>, <trigger>)*

Parameters

<in_pipe>
The pipe from which bits are taken.
`WORD PIPE`

<xor>
A number identifying bit positions that are active-low.
`WORD CONSTANT | WORD VARIABLE`

<and>
A number selecting bit positions to test.
`WORD CONSTANT | WORD VARIABLE`

<select>
This parameter is for future expansion and must be zero.
`WORD CONSTANT | WORD VARIABLE`

<trigger>
The trigger that is asserted when bits match the specified conditions.
`TRIGGER`

Description

LOGIC asserts a trigger when bits from *<in_pipe>* match conditions specified by the *<xor>*, *<and>*, and *<select>* parameters. The values in *<in_pipe>* are typically obtained by sampling the digital input port. There is a built-in hysteresis behavior, and **LOGIC** asserts *<trigger>* only once each time the conditions are satisfied. The *<select>* parameter is for future expansion and must be zero.

To check triggering conditions, a **LOGIC** task computes a Boolean expression for each input value by:

- inverting each bit whose corresponding *<xor>* bit is 1,
- masking to 0 each bit whose corresponding *<and>* bit is 0,
- setting the Boolean value to 1 if the result is equal to the *<and>* parameter,
- setting the Boolean value to 0 otherwise.

If the Boolean value is 1, *<trigger>* is asserted. After that, the **LOGIC** task must receive an input value for which the computed Boolean is 0. The trigger is asserted again the next time the Boolean value is 1, and so forth. If more than one bit is tested, **LOGIC** requires all the selected bits to satisfy the triggering condition.

The following configuration outlines how to compute values for the three **LOGIC** parameters:

1. Determine which bits to use for triggering. Set the corresponding bits of the *<and>* parameter to 1. Note that bit 0 on the digital port is the least significant bit and bit 15 on the digital port is the most significant bit.

2. To trigger when a particular bit value is 1, set the corresponding bit of the *<xor>* parameter to 0. To trigger when a particular bit value is 0, set the corresponding bit of the *<xor>* parameter to 1.

3. Set the *<select>* parameter to 0.

Note: The behavior of this command is very similar to edge-triggered operation, but not exactly the same. Edge-triggering uses one sample of past history, but this command does not. For true edge-triggered operation, use the command with a **TRIGGERS** operating mode having a nonzero STARTUP property.

Example

```
LOGIC (IPIPE3, $0004, $0004, 0, T1)
```

Assert trigger T1 when bit 2 of the input channel pipe data value is 0.

```
TRIGGER T2 MODE=NORMAL STARTUP=1
LOGIC (IPIPE3, 0, $5555, 0, T2)
```

Assert trigger T2 when all data bits in odd-numbered positions are 1, and this condition was not true for the previous sample.

See Also

CHANGE, **DLIMIT**, **LIMIT**, **PEAK**, **TRIGGERS**

LOW

Define a task that scans blocks of data for minimum values.

LOW *(<in_pipe>, <count>, <out_pipe1> [, <out_pipe2>])*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<count>
A value that specifies the size of blocks to be scanned.
`WORD CONSTANT`

<out_pipe1>
Output data pipe for minimum block values.
`WORD PIPE | LONG PIPE`

<out_pipe2>
If specified, the pipe to which the sample number of each minimum value is placed.
`WORD PIPE`

Description

LOW scans blocks of size *<count>* for minimum values. The minimum of each block is placed in the pipe *<out_pipe1>*. If pipe *<out_pipe2>* is specified, the sample number of each minimum value is placed in *<out_pipe2>*. The sample number has a value between 0 and *<count>*-1.

When there are multiple values equal to the same minimum, the location of the first minimum is reported.

Examples

```
LOW (IPIPE3, 100, P1)
```

Read blocks of 100 values from input channel pipe 3 and send the minimum of each block to pipe `P1`.

```
LOW (P2, 1000, P3, P4)
```

Read blocks of 1000 values from pipe `P2`, send the minimum of each block to pipe `P3`, and send the position of the minimum in each block to pipe `P4`.

See Also

FINDMAX, **HIGH**, **PEAK**, **RANGE**

MASTER

Configure an input or output configuration's sampling clock as a master clock.

MASTER

Description

The **MASTER** command configures an input or output configuration's sampling clock as a master clock for other Data Acquisition Processors. The **MASTER** command is used in multiple Data Acquisition Processor systems; the master Data Acquisition Processor supplies a clock signal to all slave Data Acquisition Processors for synchronized input sampling or output updates.

See Also

CLOCK, **SLAVE**

MERGE

Define a task that merges data sequentially from multiple input pipes.

MERGE *(<in_pipe_0>, ... , <in_pipe_n-1>, <out_pipe>)*

Parameters

<in_pipe_0> ... <in_pipe_n-1>
Input data pipes.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>
Output pipe for merged data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

MERGE reads data from one or more input pipes and places the data consecutively into an output pipe.

Data arrival rates in all pipes *<in_pipe_0>* through *<in_pipe_n-1>* must be equal. If different volumes of data arrive in different pipes, data will backlog in the pipes having higher volumes, causing processing to stall when a pipe has no capacity to accept more data.

MERGE is useful for merging binary data from several pipes to a single communication pipe for transmission to the host computer. **MERGE** is the inverse of **SEPARATE**.

When there is a mix of input and output data types, the **MERGE** command applies a mix of strategies for converting input data to fit into the output pipe. Conversion operations can be slow, so it is best to avoid mixed data types. When data types must be mixed, refer to the following chart to determine how the data representations are altered. Depending on the type of input and output pipes, the data are handled in one of five ways:

1. Copy – If the pipes are the same type, **MERGE** copies the data directly from *<in_pipe>* to *<out_pipe>* with no changes and no inconsistencies.

2. Alias – This is a kind of copy that preserves the bit representation when values have equal size, but leaves the bit representation inconsistent with the natural data type of the destination pipe. The values are not directly

meaningful if interpreted as the data type of the destination pipe. To recover the data, the **SEPARATE** command can be used; or the data can be transferred to the PC host where a software application reinterprets the bits within buffer storage as the original data type.

3. Convert – A value-preserving type conversion operation can be applied when the *<out_pipe>* data type can represent all possible values allowed by *<in_pipe>* with no loss of resolution, range or accuracy. The new representation placed into *<out_pipe>* is consistent with the type of that pipe. Conversions from WORD to LONG type are done by sign extension, so this conversion is easily reversed by ignoring the 16 extension bits.

4. Split – Consistent with previous versions of DAPL, data elements that are too long to fit into shorter data elements of the destination pipe are split. The shorter fragments are then sent to the destination pipe. This can be done only when the destination pipe is a fixed point type, because bit patterns in the fragments will not always correspond to valid floating point values. The least significant parts of the bit representations are sent first. To recover split data, the **SEPARATE** command can be used; or the data can be transferred to the PC host where the parts must be arranged in storage so that the software can properly interpret bit representations as the original type.

5. Not Allowed – If none of the above options can transfer the information without loss of accuracy, range or resolution, that combination of input and output types is not allowed. An error will be diagnosed and the command will terminate.

The following table shows which operations are applied for each combination of *<in_pipe>* and *<out_pipe>* data types.

<in_pipe>	*<out_pipe>*			
	WORD	LONG	FLOAT	DOUBLE
WORD	copy	convert	convert	convert
LONG	split	copy	not allowed	convert
FLOAT	split	alias	copy	convert
DOUBLE	split	split	not allowed	copy

Examples

```
MERGE (P1,P2,P3,P4)
```

Read data from pipes P1, P2, and P3 in sequence, and place the data consecutively into pipe P4.

```
MERGE (P5,P6,P7,$BINOUT)
```

Transfer data from pipes P5, P6, and P7 to the host PC through the binary output communication pipe.

See Also

BMERGE, MERGEF

MERGEF

Define a task that merges data from several pipes in arbitrary sequence.

MERGEF *(<in_pipe_0>, ... , <in_pipe_n-1>, <out_pipe>)*

Parameters

<in_pipe_0>
First input data pipe.
`WORD PIPE | LONG PIPE`

<in_pipe_n-1>
Last input data pipe.
`WORD PIPE | LONG PIPE`

<out_pipe>
Output pipe for merged data.
`WORD PIPE | LONG PIPE`

Description

MERGEF merges data from several pipes into a single pipe. An identifying flag is included with each value. **MERGEF** reads a value from any input pipe that has data. Each time a value is read from a pipe, two values are placed in *<out_pipe>*. The first value is a flag identifying the pipe from which the value was read. The identifying flag is a number from 0 to n-1. The flag is followed by the value.

When transferring a word value to a long pipe, a long flag is placed in the output pipe followed by a long value that is sign extended. When transferring a long value to a word pipe, a word flag is placed in the output pipe followed by two words with the least significant word first.

MERGEF is different from **MERGE** in that data values are not read sequentially from the input pipes. Therefore, the data rates from different input pipes need not match, and the order that data is placed into *<out_pipe>* is not predictable.

MERGEF is particularly useful for transmitting binary data to the PC when data values are not written to the pipes at the same rate. Based on the identifying flag preceding each data value, the PC can process the data correctly.

When using **MERGEF** with DAPview for Windows, the Binary Prefix of DAPview for Windows can allow selective logging or display of data originating from a specific **MERGEF** input pipe.

MERGEF is the inverse of **SEPARATEF**.

See Also

BMERGE, **BMERGEF**, **MERGE**, **SEPARATE**, **SEPARATEF**

NMERGE

Define a task that merges data in groups.

NMERGE *(<n1>, <in_pipe1>, <n2>, <in_pipe2>, ... , <out_pipe>)*

Parameters

<n1>
Number of data values to transfer from the first input pipe.
`WORD CONSTANT`

<in_pipe1>
First input pipe.
`WORD PIPE | LONG PIPE`

<n2>
Number of data values to transfer from the second input pipe.
`WORD CONSTANT`

<in_pipe2>
Second input pipe.
`WORD PIPE | LONG PIPE`

<out_pipe>
Output pipe for merged data.
`WORD PIPE | LONG PIPE`

Description

NMERGE reads data from one or more input pipes and places the data into an output pipe. **NMERGE** transfers *<n1>* data values from *<in_pipe1>* to *<out_pipe>*, then transfers *<n2>* data values from *<in_pipe2>* to *<out_pipe>*, etc. When transferring a word value to a long output pipe, the value is sign extended. When transferring a long value to a word output pipe, two words are placed in the output pipe, with the least significant word first, followed by the most significant word.

NMERGE is useful for merging different quantities of binary data from several pipes to a single communication pipe for transmission to the host computer. For instance, a single trigger time stamp value in one pipe can be merged with many values from a trigger **WAIT** in another pipe.

Example

```
NMERGE (1, P1, 1000, P2, $BINOUT)
```

Transfer one data value from pipe `P1` followed by 1000 data values from pipe `P2` to `$BINOUT`.

See Also

BMERGE, **BMERGEF**, **MERGE**, **MERGEF**, **SEPARATE**, **SEPARATEF**

NTH

Define a task that retains one out of every `N` trigger events.

NTH *(<trigger1>, <n>, <trigger2>)*

Parameters

<trigger1>
The trigger containing the original event sequence.
`TRIGGER`

<n>
Number of trigger assertions.
`WORD CONSTANT`

<trigger2>
The trigger containing the modified event sequence.
`TRIGGER`

Description

NTH reads from *<trigger1>* and passes the last of every *<n>* trigger assertions to *<trigger2>*, ignoring the first *<n-1>* triggers. *<n>* must be a positive, nonzero integer.

Example

```
NTH (T1, 100, T2)
```

Pass every hundredth trigger from `T1` to `T2.`

See Also

WAIT

ODEFINE

Begin an output updating configuration.

ODEFINE *<name> <n>*

ODEF *<name> <n>*

Parameters

<name>

Assigned output configuration name.
Alphanumeric string limited to 23 characters.

<n>

Number of output channels to be defined.
WORD CONSTANT

Description

ODEFINE begins a group of commands that define an output updating configuration.

```
ODEFINE <name> <n>
  [output configuration command] *
END
```

The complete list of output configuration commands is given in Chapter 5. These command lines configure output channel pipes, specify update intervals, etc.

An output configuration configures the Data Acquisition Processor for isochronous (clocked) output updates. Use **DACOUT** or **DIGITALOUT** for unclocked output updates with lower latency.

The **END** command completes the configuration started by the **ODEFINE** command, making the configuration available for execution.

<name> is a unique name given to the output configuration. *<name>* must be an alphanumeric string with no spaces and is limited to 23 characters.

<n> is the number of output channel pipes. The expansion capacity of the Data Acquisition Processor board determines the possible range of outputs.

Example

```
ODEFINE OUTPR 2
  SET OP0 A0
  SET OP1 A1
  TIME 1000
END
```

Begin the definition of an output configuration named `OUTPR` with 2 analog output channel pipes.

See Also

END, **IDEFINE**, **PDEFINE**, **DACOUT**, **DIGITALOUT**

OFFSET

Define a task that adds 16-bit signed numeric offsets to data values.

OFFSET *(<in_pipe>, <offset_value>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<offset_value>
A value that specifies the 16-bit signed numeric offset to be added to each data value.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for modified data.
`WORD PIPE`

Description

OFFSET reads data from *<in_pipe>*, adds a 16-bit signed numeric offset to each value, and sends the results to *<out_pipe>*. This command is useful for removing DC offset voltages from analog input channel pipe data.

Example

```
OFFSET (IPIPES(0,2,4), -20, P1)
```

Read from input channel pipes 0, 2, and 4, subtract 20 from each value, and send the results to pipe `P1.`

OPTIONS

Set various system options.

OPTIONS *<name>=<value> [, <name>=<value>]**

OPTION *<name>=<value> [, <name>=<value>]**

OPT *<name>=<value> [, <name>=<value>]**

O *<name>=<value> [, <name>=<value>]**

Parameters

<name>
Option name.

<value>
New system option.

Description

OPTIONS sets various system options. The system option names are:

AINEXPAND	Set analog input expansion mode.
BPOUTPUT	Set bipolar output.
BUFFERING	Set buffering mode.
DECIMAL	Set input/output format.
ERRORQ	Hold error messages.
FLOATERROR	Set floating point error mode.
OVERFLOWQ	Hold overflow messages.
PROMPT	Control the system prompt character.
QUANTUM	Set global time slice quantum.
RESTORE	Restore the options from the options stack.
SAVE	Push the current options onto a stack.
SCHEDULING	Set task scheduling mode.
SYSINECHO	Set echo mode for input.
TERMINAL	Select input response for interactive programs.
UNDERFLOWQ	Hold underflow messages.

Most *<names>* are set/reset options; they can be assigned ON/OFF or YES/NO values.

AINEXPAND = ON|OFF

The connector pinout on some Analog Input Expansion Boards is different than the Data Acquisition Processor connector pinout. Only older 64-channel input expansion boards use AINEXPAND; check the "Features of DAPL dependent on DAP model" document for product numbers. When AINEXPAND is on, DAPL remaps input pins on an expansion board to match the pinout of the Data Acquisition Processor analog connector. When AINEXPAND is off, DAPL does not remap input pins. AINEXPAND defaults to off. See the Analog Input Expansion Board documentation for more details of input pin mapping using AINEXPAND.

BPOUTPUT = ON|OFF

If BPOUTPUT is on, numbers sent to the digital-to-analog converters are interpreted as bipolar voltages. If BPOUTPUT is off, numbers sent to the digital-to-analog converters are interpreted as unipolar voltages. This option defaults to on; if the output range of the digital-to-analog converters is changed to unipolar, this option must be changed to off. See Chapter 8 for more information about how to use integers with unipolar outputs. See the connector chapters for more information about output range configuration.

BPOUTPUT is an abbreviation for BiPolar OUTPUT.

BUFFERING = OFF|MEDIUM|LARGE

The BUFFERING option helps tasks to optimize their data buffering configuration. Three modes are available, OFF, MEDIUM and LARGE. MEDIUM is the default mode. In this mode, tasks use moderate size memory buffers suitable for most operations. The mode LARGE improves processing efficiency by providing tasks with larger memory buffers. The mode OFF disables buffering, and tasks use small buffers or single values for lowest latency.

DECIMAL = ON|OFF

If DECIMAL is on, numbers sent from the Data Acquisition Processor are in decimal format, and numbers received from the PC also are interpreted as decimal numbers. Hexadecimal numbers can be specified with a '$' prefix in either mode. If DECIMAL is off, all numbers are in hexadecimal format. The default is on.

Status displays and messages are not affected by the DECIMAL flag. Memory addresses and register displays always are hexadecimal while other values always are decimal. Configuration and diagnostic displays generated by **DISPLAY** and **SDISPLAY** commands always show decimal values. The vector length display always is decimal.

ERRORQ = ON|OFF

When ERRORQ=ON, DAPL suppresses all warning and error messages. This option allows a program in the host PC to process data at high speed without allowing for error messages. The default is on.

The commands **DISPLAY ENUM** or **DISPLAY EMSG** can be used at any time to determine whether any errors have been suppressed because of the ERRORQ option. The Data Acquisition Processor retains the first error message that is suppressed after ERRORQ is set to on; setting ERRORQ to off after an error has occurred causes the Data Acquisition Processor to print this error message. The error message is printed only if the error has not already been displayed using DISPLAY ENUM or DISPLAY EMSG.

OVERFLOWQ, UNDERFLOWQ, and ERRORQ are useful when the output of the Data Acquisition Processor is being manipulated by a host computer program. By turning these flags on, the program can prevent error messages from interrupting the Data Acquisition Processor output. Errors then can be checked under controlled conditions using the **DISPLAY** command.

FLOATERROR = ON|OFF

The FLOATERROR parameter takes a Boolean value ON or OFF. The default is OFF. Setting FLOATERROR=ON allows the hardware floating point unit or emulator software to force an interrupt, generating a diagnostic message and terminating the task when various floating point errors occur. This feature is most useful during custom command development and testing. Most finished applications should select the OFF setting. The OFF setting attempts to provide a fix and continue processing after a floating point error occurs.

OVERFLOWQ = ON|OFF

OVERFLOWQ controls whether sampling overflow messages are reported immediately or queued. If OVERFLOWQ is true, overflow messages are queued and can be viewed with the command **DISPLAY OVERFLOWQ**. See Chapter 11 for more information. The default is on.

OVERFLOWQ, UNDERFLOWQ, and ERRORQ are useful when the output of the Data Acquisition Processor is being manipulated by a host computer program. By turning these flags on, the program can prevent error messages from interrupting the Data Acquisition Processor output. Errors then can be checked under controlled conditions using the **DISPLAY** command.

PROMPT = ON|OFF

If PROMPT is on, the DAPL command interpreter operating with option SYSINECHO=ON prints a prompt character at the beginning of any line on which

DAPL is requesting input. Since the prompt character changes when DAPL enters different modes, this character provides an easy way to determine the type of input expected. If prompt is `OFF`, no prompt character is printed. Setting `PROMPT=OFF` is convenient when DAPL output is being processed by a computer program other than DAPview for Windows. The default is `OFF`.

`QUANTUM = <n>`

The `QUANTUM` option sets DAPL's global time slice quantum to a given value. A valid time quantum value ranges from 100 to 5000 (µs), board dependent. The default time quantum is 2000 µs. A smaller time quantum forces the CPU to switch among tasks more often. More frequent task scheduling reduces the task switching latency at the expense of decreased system efficiency. On the other hand, a larger time quantum improves efficiency by reducing task switching overhead but at the cost of higher latency. See Chapter 13 for more information.

`RESTORE`

Restores the most recent options stored on the options stack. See the description of `SAVE` below. `RESTORE` does not take a *<value>*.

`SAVE`

Pushes the current options onto a stack. The options stack is five levels deep. An `OPTIONS SAVE` command beyond five stack entries will cause the oldest entry on the stack to be lost. `SAVE` does not take a *<value>*.

`SCHEDULING = ADAPTIVE|FIXED`

The `SCHEDULING` option sets the task scheduling mode. DAPL supports two scheduling modes, `ADAPTIVE` and `FIXED`. The default scheduling mode is `FIXED`. In the `ADAPTIVE` mode, DAPL schedules some tasks less often than others, based on the actual data flow. In the `FIXED` mode, all tasks are scheduled equally often in a round robin fashion. See Chapter 13 for more information.

`SYSINECHO = ON|OFF`

The `SYSINECHO` option controls the echoing of characters read from the default text input pipe, `$SYSIN`. The option `SYSINECHO=ON` is most useful for interactive programs such as DAPview for Windows. The default is `OFF`.

`TERMINAL = ON|OFF`

The `TERMINAL` option determines the Data Acquisition Processor response when it receives text input while it is sending text output. When `TERMINAL` is on and the Data Acquisition Processor receives any character, the output of any active **`FORMAT`** tasks is suspended until an entire line, ending with a carriage return, is received. The `TERMINAL` option should be on if the Data Acquisition Processor is

communicating with an interactive program such as DAPview for Windows. The default is off.

`UNDERFLOWQ = ON|OFF`

`UNDERFLOWQ` controls whether output configuration underflow messages are reported immediately or queued. If `UNDERFLOWQ` is true, underflow messages are queued and can be viewed with the command **`DISPLAY UNDERFLOWQ`**. See Chapter 11 for more information. The default is `ON`.

`OVERFLOWQ`, `UNDERFLOWQ`, and `ERRORQ` are useful when the output of the Data Acquisition Processor is being manipulated by a host computer program. By turning these flags `ON`, the program can prevent error messages from interrupting the Data Acquisition Processor output. Errors then can be checked under controlled conditions using the **`DISPLAY`** command.

Note: When DAPview for Windows starts up, it changes some options to provide interactive features such as command echoing and automatic error display.

Examples

```
OPTIONS DECIMAL=OFF
```

Set all input and output in hexadecimal format.

```
OPTIONS PROMPT=ON,SYSINECHO=ON,TERMINAL=ON,\
ERRORQ=OFF,OVERFLOWQ=OFF,UNDERFLOWQ=OFF
```

Set the Data Acquisition Processor in an interactive mode.

```
OPTIONS BPOUTPUT=OFF
```

Tell the DAPL interpreter that the DACs are configured to generate unipolar voltages.

```
OPTIONS QUANTUM=500
```

Set the time slice quantum to 500 µs.

```
OPTIONS SAVE
```

Save current options.

See Also

DEXPAND, **DISPLAY**, **OUTPORT**, **SDISPLAY**

OUTPORT

Inform DAPL of the types and addresses of output expansion boards in a system.

OUTPORT *<x>[..<y>] TYPE=<z> [BPOUTPUT = <bpswitch>]*

OUTPORTS *<x>[..<y>] TYPE=<z> [BPOUTPUT = <bpswitch>]*

Parameters

<x>
A value that specifies the output port number.
`WORD CONSTANT`

<y>
An optional output port number range.
`WORD CONSTANT`

<z>
A value that specifies the output port type.
`WORD CONSTANT`

<bpswitch>
A keyword specifying an analog output mode, `ON` for bipolar output, `OFF` for unipolar output.
`WORD CONSTANT`

Description

`OUTPORT` informs DAPL of the types of output expansion boards in a system and their output port addresses. *<x>* and *<y>* are integers between 0 and 63. *<z>* is 0 or 1.

<x> is an output port number. It optionally is followed by ..*<y>* to define an output port number range. The range must be a multiple of four and start from a boundary that is a multiple of 4. For example, 0..3 and 8..15 are valid address ranges, while 0..4 and 7..15 are not.

<z> is the output port type. Type 0 is for digital output expansion boards. Type 1 is for analog output expansion boards. For analog output expansion boards the optional parameter `BPOUTPUT=`*<bpswitch>* selects bipolar or unipolar range. Setting *<bpswitch>* value `OFF` selects unipolar operation. Setting *<bpswitch>* value `ON` selects bipolar operation. The default is `ON`.

The following table summarizes the output behavior of the analog signal for each mode of operation. The value of Vmax is typically +5V or +10V and is dependent on the hardware configuration. See your hardware documentation for more information about voltage ranges.

`BPOUTPUT` mode -----> Output voltage range

Number Range	`ON`	`OFF`
-32767 to 0	-Vmax to 0	0
0 to +32767	0 to Vmax	0 to Vmax

Example

```
OUTPORT 0..3 TYPE=1
```

Set output port address 0-3 to analog output expansion.

See Also

DISPLAY, **OPTIONS**, **DEXPAND**, **DIGITALOUT**, **DACOUT**

OUTPUTWAIT

Delay output updating until a specified number of samples are present in output channel pipes.

OUTPUTWAIT *<n>*

Parameters

<n>

Number of samples that must be in an output channel pipe before updating.
`WORD CONSTANT | LONG CONSTANT`

Description

OUTPUTWAIT causes DAPL to wait until *<n>* samples are in an output configuration's output channel pipes before output updating begins. *<n>* must not be zero and must be a multiple of the number of output channel pipes. **OUTPUTWAIT** applies to noncyclical output configurations only, and defaults to 100 milliseconds of output data. For most applications, the default value of **OUTPUTWAIT** should not be overridden.

Note: Setting *<n>* less than its default value may result in output underflow.

Example

```
OUTPUTWAIT 1000
```

Wait for 1000 values before starting to update the outputs.

See Also

COUNT, **CYCLE**, **UPDATE**

PAUSE

Cause DAPL command processing to pause for a specified number of milliseconds.

PAUSE *<milliseconds>*

PA *<milliseconds>*

Parameters

<milliseconds>

Time in milliseconds that DAPL pauses.

`WORD CONSTANT | LONG CONSTANT`

Description

PAUSE causes the DAPL command interpreter to pause for a specified number of milliseconds.

This command can be used when scheduling a sequence of operations, to allow time for each test in the sequence to finish.

Note: The timing accuracy of **PAUSE** depends on the timing accuracy of the Data Acquisition Processor real-time clock. The real-time clock of the Data Acquisition Processor is derived from the CPU crystal clock and provides good long-term accuracy.

Example

```
PAUSE 1000
```

Pause DAPL for 1 second.

PCASSERT

Define a task that asserts a trigger based on asynchronous input.

PCASSERT *(<control>, <trigger> [,<ref_rate>])*

Parameters

<control>
A source of data indicating a triggering event.
`WORD VARIABLE | LONG VARIABLE | WORD PIPE`

<trigger>
The trigger asserted when the variable changes to a nonzero value.
`TRIGGER`

<ref_rate>
An optional value that specifies the amount of data reduction.
`WORD CONSTANT | WORD PIPE | LONG PIPE`

Description

PCASSERT asserts a trigger based on asynchronous input from *<control>*. *<control>* may be a variable or a pipe. A trigger is asserted each time the variable changes to a nonzero value, or each time that a number appears in the input pipe. **PCASSERT** automatically resets the variable to zero, or removes the value from the pipe to clear the trigger request. The *<control>* is ordinarily used by the PC and not by internal DAPL operations. Internal DAPL operations should use synchronous triggering commands for guaranteed synchronization.

PCASSERT differs from other triggering commands in the way it updates the trigger sample count. Commands such as **LIMIT** analyze a stream of data. Trigger sample numbers are based on the number of values that **LIMIT** scans. **PCASSERT**, on the other hand, does not trigger on events from a data stream.

PCASSERT bases its trigger sample numbers on the sample count of the active input configuration. For an input configuration with a single sampled channel, the assertion timestamp corresponds to the current sample number of the active input configuration. When there are multiple input channels in the sampling configuration, or when a command such as **AVERAGE** reduces the data rate, this count is too large. In this case a constant *<ref_rate>* can be specified. The input configuration sample count is divided by *<ref_rate>* before the trigger is updated.

When the number of samples in the data stream does not have a simple relationship to the sampled data, the *<ref_rate>* parameter can alternatively specify a data pipe acting as a reference stream. The contents of this pipe are counted but not processed, and this count is used for updating the trigger.

PCASSERT is intended for individual, isolated events. **PCASSERT** will not trigger multiple times on the same sample. Pipe contents or the variable value can be monitored to check on completion of the request.

Examples

```
PCASSERT (V, T)
```

Assert trigger at current input sample number whenever variable `V` becomes nonzero.

```
PCASSERT (P1, T)
```

Assert trigger at current input sample number whenever a value is read from pipe `P1`.

```
PCASSERT (V, T, 100)
```

Assert trigger for a **WAIT** command that is reading data reduced 100 times by an **AVERAGE** command.

See Also

WAIT, **LIMIT**

PCOUNT

Define a task that counts the number of values placed into a pipe.

PCOUNT *(<in_pipe>, <variable>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<variable>
Variable where the counts are incremented.
`WORD VARIABLE | LONG VARIABLE`

Description

PCOUNT reads values from *<in_pipe>* and increments the contents of *<variable>* each time a value is read. The values are discarded.

<variable> is not reset to zero when a **PCOUNT** task is started. Consequently, **PCOUNT** variables can be used to keep running totals of the number of values placed in pipes. To reset a variable, use a **LET** command.

A useful application of this command is temporary removal of data from a pipe during application development and testing.

Example

```
PCOUNT (P1,V1)
```

Count the number of values appearing in pipe `P1` and update the value of variable `V1`.

See Also

PVALUE

PDEFINE

Create a processing procedure.

PDEFINE *<name>*

PDEF *<name>*

Parameters

<name>

Unique processing procedure name.
Alphanumeric string limited to 23 characters.

Description

PDEFINE begins a group of commands that define a processing procedure.

```
PDEFINE <name>
  [task definition command] *
END
```

Each task definition can specify one of the pre-defined commands listed in Chapter 5, a downloaded custom command, or a DAPL expression calculation.

The **END** command completes the procedure started by the **PDEFINE** command, making the procedure available for execution.

<name> must be a unique name assigned to the processing procedure. *<name>* must be an alphanumeric string with no spaces and is limited to 23 characters.

Example

```
PDEFINE PR
...
END
```

Define a processing procedure named `PR`.

See Also

END, **IDEFINE**, **ODEFINE**

PEAK

Define a task that detects maxima and minima of data.

PEAK *(<in_pipe>, <peak_select>, <trigger> [, <region>])*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<peak_select>
A numeric parameter that determines the types of peaks to detect.
`WORD CONSTANT`

<trigger>
The trigger asserted when peaks are detected.
`TRIGGER`

<region>
An optional parameter that specifies the region for detecting peaks.
`REGION`

Description

PEAK looks for maxima and minima of data read from *<in_pipe>*. *<peak_select>* determines the types of peaks to detect according to the following table:

`0`	Detect minima
`1`	Detect maxima
`2`	Detect both minima and maxima

When a peak of the appropriate type is detected, as indicated by a sign change in the derivative of the input data, **PEAK** asserts *<trigger>*.

An optional *<region>* also can be specified. If this *<region>* is present, only peaks with values satisfying *<region>* are detected; all others are ignored.

Examples

```
PEAK (P1, 2, T1)
```

Assert trigger `T1` when any maxima or minima are detected in the data from pipe `P1`.

```
PEAK (P2, 0, T2, INSIDE,8000,9000)
```

Assert trigger `T2` when a minimum is detected with a value from 8000 to 9000.

See Also

DLIMIT, **HIGH**, **LIMIT**, **LOW**, **FINDMAX**

PID1

Define a task for PID feedback control.

```
PID1 (<in_pipe>, <set_point>, <pgain>, <igain>, <dgain>
    [ <bypass>, ] <out_pipe>, [<low_clamp>, <high_clamp>]
    [, <ramp>])
```

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<set_point>
`WORD CONSTANT | WORD VARIABLE`

<pgain>
A value that determines the degree of proportional response.
`WORD CONSTANT | WORD VARIABLE`

<igain>
A value that determines the degree of integral response.
`WORD CONSTANT | WORD VARIABLE`

<dgain>
A value that determines the degree of derivative response.
`WORD CONSTANT | WORD VARIABLE`

<bypass>
An override switch for bypassing automatic control.
`WORD VARIABLE`

<out_pipe>
Output data pipe for control output values.
`WORD PIPE`

<low_clamp>
An optional parameter that limits the minimum control output values.
`WORD CONSTANT | WORD VARIABLE`

<high_clamp>
An optional parameter that limits the maximum control output values.
`WORD CONSTANT | WORD VARIABLE`

<ramp>
A value that specifies the maximum amount of change in setpoint allowed per PID update.
WORD CONSTANT

Description

PID1 implements Proportional Integral Derivative closed-loop process control. The **PID1** command reads feedback samples of the controlled system's output from *<in_pipe>* and compares them to the desired system output level specified by *<set_point>*. The difference is the control error. The **PID1** command computes control output values to make the controlled system output match the specified setpoint level, reducing the control error to 0. PID control output values are placed into *<out_pipe>*.

The PID strategy uses three correction rules to drive the system output toward the setpoint:

1. Proportional correction: Use a greater control effort when the system's output deviates further from the setpoint.

2. Integral correction: Use gradually increasing control effort when the system's output remains away from the setpoint for an extended time.

3. Derivative correction. Decrease the control effort if the system's output changes too fast.

This strategy is more formally defined by the following equation:

```
control output = -(<pgain>*e + <igain>*intgrl(e) +
<dgain>*deriv(e))
```

where

```
e  = feedback from <in_pipe> - <setpoint>
intgrl(e) = integral of e
deriv(e)  = derivative of e
```

The integral and derivative are estimated using discrete approximations.

A common alternate form of the PID formula expresses the output in terms of a proportionality constant (P), integral time (TI), and derivative time (TD):

```
                  1
output = P ( e +---- * intgrl(e) + TD * deriv(e) )
                  TI
```

The parameters of the 2 forms of the PID equation are related by:

```
                  P
P = <pgain>     ----    = <igain>      P * TD = <dgain>
                  TI
```

The *<pgain>*, *<igain>*, *<dgain>*, *<low_clamp>* and *<high_clamp>* parameters of the **PID1** command can be variables. Variable parameters can be changed at any time, using the **PVALUE** command or the **LET** command. The **PID1** command checks for changed variables each update cycle.

The *<pgain>*, *<igain>*, and *<dgain>* parameters may be constants or variables. These gain parameters determine the degree of proportional, integral, and derivative response, respectively. The gain coefficients have the following scaling and range restrictions:

- *<pgain>* ranges from -10000 to 10000. Each unit of *<pgain>* represents a step of 0.01, yielding an effective range of -100.00 to 100.00 .
- *<igain>* ranges from -10000 to 10000. Each unit of *<igain>* represents a step of 0.0001, yielding an effective range of -1.0 to 1.0 .
- *<dgain>* ranges from -10000 to 10000. Each unit of *<dgain>* represents a step of 0.01, yielding an effective range of -100.00 to 100.00 .

When the optional *<bypass>* variable is specified and has a nonzero value, PID action is not applied. Instead, the control output directly tracks changes to the setpoint value, remaining constant otherwise. This feature is useful, for example, in manually bringing a system from a ‘cold start’ to an operating level near the desired setpoint. Setting the *<bypass>* variable to 0 enables PID action from that point. (Most systems will also need an adjustment to the setpoint value.)

When the optional *<low_clamp>* and *<high_clamp>* parameters are specified, these parameters respectively limit the minimum and maximum control output values of the **PID1** command. While the **PID1** command output is being limited, the error integral is not updated. This improves transient response for large changes in the setpoint and at initial startup.

The optional *<ramp>* parameter is useful for systems that require high proportional gain settings and are sensitive to rapid setpoint changes. The *<ramp>* parameter specifies the maximum amount of change in setpoint that is allowed per PID update. For example, when the *<ramp>* value is 10 and the *<setpoint>* is changed from 0 to 1000, the setpoint is increased during the next 100 updates to reach the new setpoint level. The *<ramp>* parameter does not have any effect when the *<bypass>* option is active.

Variable gain parameters, variable limit parameters, the *<ramp>* feature and the *<bypass>* feature add small amounts of overhead to PID loop operation. These

features should be used only when needed, and should be avoided for high-speed operation.

Note: The **PID1** command is a replacement for the PID command provided in previous versions of DAPL and DAPL 2000.

Examples

```
PID1 (P1,12000,VP,VI,VD,P2)
```

Receive feedback information about system output level in pipe P1. For each value received, use gains specified in variables VP, VI, and VD to compute control values for driving the system level to the setpoint level 12000. Send control output values to pipe P2.

```
VARIABLE VSET=8000
VARIABLE VLOCK=1
PID1 (P1,VSET,VP,VI,VD,VLOCK,P2,40)
...
START
PAUSE 10000
LET VLOCK=0
LET VSET=12000
```

Send the controlled system the value 8000 for the first 10 seconds. After this amount of time, adjust the setpoint level to 12000 and begins normal PID updates. During the setpoint transition from 8000 to 120000, adjust the setpoint by 40 before each update, so that the setpoint reaches the 12000 level after 100 update intervals.

```
PID1 (P1,12000,VP,VI,VD,P2, 0,32767)
```

Operate the same as the first example, but clamp the output levels so that negative values of control output are not allowed.

See Also

DACOUT, **PVALUE**, **PWM**

PIPES

Create new pipes.

PIPES *<pipe_def> [, <pipe_def>]**

PIPE *<pipe_def> [, <pipe_def>]**

P *<pipe_def> [, <pipe_def>]**

<pipe_def> = <pipe_name> [MAXSIZE=<max_size>]
[WAIT | NOWAIT] [BYTE | WORD | LONG | FLOAT | DOUBLE]

Parameters

<pipe_name>
Assigned pipe name.

<max_size>
A value that sets the maximum number of values stored in a pipe.
`WORD CONSTANT | LONG CONSTANT`

Description

The **PIPES** command creates new pipes. The `BYTE`, `WORD`, or `LONG` keyword specifies the data type of the pipe. If the data type is omitted, the pipe holds 16-bit (word) values; if `LONG` is specified, the pipe holds 32-bit (long) values; if `BYTE` is specified, the pipe holds 8-bit (byte) values.

Pipes initially are empty.

<max_size> sets the maximum number of values stored in a pipe. The actual maximum size may be slightly larger than the number specified, because of internal system rounding. If a maximum size is not specified, it defaults to 32768. The maximum size of a DAPL pipe is limited by the available onboard buffer memory. In most cases, it is best to omit this parameter and let DAPL select the maximum sizes of pipes.

When a pipe reaches its maximum size during the execution of a processing procedure, it cannot accept additional data. If the pipe is defined with the `WAIT` option, a task attempting to add data to the pipe is temporarily suspended until some data are removed from the pipe. If the `NOWAIT` option is specified, a task trying to add data to the full pipe does not wait for space in the pipe. Instead of adding the

data, the data is lost. WAIT is the default option; NOWAIT is useful only in specialized applications.

Examples

```
PIPE P1
```

Define the pipe P1, with a maximum size of 32768.

```
PIPES P2 MAXSIZE=1024, P3 LONG
```

Define a word pipe with maximum size 1024 and a long pipe with maximum size 4096.

```
PIPE A234 MAXSIZE=10000 NOWAIT BYTE
```

Define a byte pipe with maximum size 10000 and the NOWAIT option, which flushes data to be added whenever the pipe is full.

See Also

EMPTY, **FILL**

POLAR

Define a task that converts input data pairs from Cartesian to polar coordinates.

POLAR *(<p1>, <p2>, <p3>, <p4> [, <limit>])*

Parameters

<p1>
The pipe that contains the real parts of complex numbers.
`WORD PIPE`

<p2>
The pipe that contains the imaginary parts of complex numbers.
`WORD PIPE`

<p3>
The pipe that receives the amplitude of the complex numbers.
`WORD PIPE`

<p4>
The pipe that receives the phase of the complex numbers.
`WORD PIPE`

<limit>
An optional value that specifies the lower limit for the amplitude input.
`WORD CONSTANT`

Description

Converts input data into polar coordinates. *<p1>* and *<p2>* are pipes that contain real and imaginary parts of complex numbers. *<p3>* and *<p4>* are pipes that contain the amplitude and phase of the complex numbers read from *<p1>* and *<p2>*. The phase is a binary fraction of one half cycle. To convert to radians, divide by 32768 and multiply by π. To convert to degrees, divide by 32768 and multiply by 180. The following DAPL expression performs this conversion:

```
P2 = P1 * 180 / 32768
```

In some applications, small input numbers should be ignored. If the optional parameter *<limit>* is present, the phase output is forced to zero when the amplitude of the input is less than *<limit>*.

Examples

```
POLAR (P1, P2, P3, P4)
```

Read real and imaginary parts of complex numbers from `P1` and `P2`, and return amplitude and phase in `P3` and `P4`.

See Also

DECIBEL, **FFT**, **CABS**

PRINT

Define a task that prints data as ASCII text.

PRINT [(*<in_pipe> [, <out_pipe>]*)]

Parameters

<in_pipe>
Optional input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>
Optional output text pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

PRINT formats data as ASCII text and transfers this text to the PC for logging or display. The formatting is not configurable. When *<in_pipe>* is omitted, data is taken from all data channels sampled by the currently active input sampling configuration. When *<in_pipe>* is specified, a subset of the channel range can be selected, and an optional *<out_pipe>* parameter can name a communication text pipe other than the default `$SYSOUT` communication pipe to receive the text.

PRINT attempts to build each text line with one sample from each channel. If *<in_pipe>* has too many channels to display in this manner, or if channels cannot be identified from the source pipe, each sample value is displayed on a separate text line.

Examples

```
PRINT
```

Print all input channel pipes.

```
PRINT (IPIPES(0,1,2), LISTING)
```

Print data from input channel pipes 0, 1, and 2, sending the text to `LISTING`, a communications pipe set up for this purpose.

See Also

BPRINT, **FORMAT**

PULSECOUNT

Define a task that counts low to high bit transitions.

PULSECOUNT *(<in_pipe>, <bit_number>, <cnt>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<bit_number>
A value that represents the bit number from which low to high transitions are detected.
`WORD CONSTANT`

<cnt>
A variable that is incremented by one every time a low to high transition is detected.
`WORD VARIABLE | LONG VARIABLE`

Description

PULSECOUNT reads data from *<in_pipe>* and detects low to high transitions of bit *<bit_number>*. Bit 0 is the least significant bit. Every time a low to high transition is detected, the value of variable *<cnt>* is incremented by one.

Example

```
PULSECOUNT (P, 4, V)
```

Read data from pipe `P` and increment variable `V` whenever bit 4 of the data changes from zero to one.

See Also

CTCOUNT, **FREQUENCY**

PVALUE

Define a task that updates a variable with the most recent value from a pipe.

PVALUE *(<in_pipe>, <variable>)*

Parameters

<in_pipe>

Input data pipe.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<variable>

The variable that is updated.

`WORD VARIABLE | LONG VARIABLE | FLOAT VARIABLE | DOUBLE VARIABLE`

Description

PVALUE reads data from *<in_pipe>* and updates *<variable>* with the most recent value. The data types of the pipe and variable must match.

Example

```
PVALUE (P1, V1)
```

Read data from `P1` and place the data in variable `V1.`

See Also

PCOUNT

PWM

Define a task that converts a continuous signal to a pulsewidth modulated signal on a single bit.

PWM *(<in_pipe>, <low>, <high>, <length>, <source>, <bit>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<low>
A lower threshold corresponding to output duty cycle of 0%.
`WORD CONSTANT`

<high>
An upper threshold corresponding to output duty cycle of 100%.
`WORD CONSTANT`

<length>
A value that specifies the data block size.
`WORD CONSTANT`

<source>
A data value or stream supplying values for output data bits not affected by the modulation.

`WORD CONSTANT | WORD VARIABLE | WORD PIPE`

<bit>
A value that selects the bit to modulate.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output data pipe for modulated output signal.
`WORD PIPE`

Description

PWM is an abbreviation for Pulse Width Modulation. This command allows encoding of a continuous signal as digital pulses of varying width. The output pulse stream is then multiplexed with other digital pulse streams.

PWM converts data from *<in_pipe>* to ON/OFF bit transitions in *<out_pipe>*. *<bit>* selects the bit to modulate. A block of *<length>* values is read from *<in_pipe>* and averaged. **PWM** then generates *<length>* output values. If the average is less than *<low>*, the bit is turned OFF for all output values. If the average is greater than *<high>*, the bit is turned ON for all the output values.

Otherwise, the values between *<low>* and *<high>* are mapped linearly to the number range *0..<length>*. Calling this number M, the output block contains M values with the output bit set to 1, followed by *<length - M>* values with the output bit set to 0.

The **PWM** command output is typically sent to the digital output port. If no other digital output bits are needed, the *<source>* parameter can specify a fixed bit pattern for all of the other bits. Or, if other digital output values are needed, the **PWM** command will copy all of the bits from the original *<source>* pipe and modify only the one bit position specified by *<bit>*.

Note: The difference between *<high>* and *<low>* must not exceed 32767.

Example

```
PIPES P1, P2, B1
PWM (P1, 0, 10000, 500, $FFFF, 0, B1)
PWM (P2, 0, 10000, 500, B1, 1, OP1)
```

Take values from pipes P1 and P2 and map the data ranges from 0 to 10000 into pulse width modulated digital signals. Put the first modulated signal into bit position 0 and the second modulated signal into bit position 1. Set all unused digital bits to value 1.

See Also

ODEFINE, **PID1**

RANDOM

Define a task that generates 16-bit or 32-bit random values.

RANDOM *(<type>, <seed>, <out_pipe>)*

Parameters

<type>
This parameter is for future expansion and must be zero.
`WORD CONSTANT`

<seed>
A value that initializes the sequence of random numbers.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>
Output data pipe for pseudorandom numbers.
`WORD PIPE | LONG PIPE`

Description

RANDOM generates pseudorandom numbers appropriate for the specified *<out_pipe>* (16-bit or 32-bit). Random numbers can be sent to a digital-to-analog converter to generate white noise.

<type> must be zero.

<seed> determines the sequence of random numbers. Any nonzero value generates a predetermined, reproducible sequence of numbers. A value of zero creates a seed that is derived from the Data Acquisition Processor real-time clock.

Note: Some previous versions of **RANDOM** use a different method of producing the pseudorandom values. A seed value producing a known, repeatable sequence of numbers in the previous version will now produce a different repeatable sequence.

Examples

```
PIPE P1
RANDOM (0, 0, P1)
```

Send pseudorandom numbers between 0 and 32767 to word pipe `P1`.

```
PIPE P1 LONG
RANDOM (0, 1774985, P1)
```

Send pseudorandom numbers between 0 and 2147483563 to long pipe `P1`.

```
PIPES  PR1 WORD, PR2 WORD, PRAND WORD
RANDOM(0,0,PR1)
RANDOM(0,0,PR2)
PRAND = (PR1-PR2)/2
```

Generate 16-bit zero mean, triangularly distributed white noise on interval -32767 to +32767.

See Also

WAVEFORM

RANGE

Define a task that transfers data in a specified region from an input pipe to an output pipe.

RANGE *(<in_pipe>, <region>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<region>
A region selecting which data values are transferred.
`REGION`

<out_pipe>
Output data pipe.
`WORD PIPE`

Description

RANGE transfers data values that satisfy *<region>* from *<in_pipe>* to *<out_pipe>*. Values that do not satisfy *<region>* are ignored.

Example

```
RANGE (P1, INSIDE,-10000,10000, P2)
```

Read data from pipe `P1` and transfer all values from -10000 to 10000 to pipe `P2.`

See Also

HIGH, **LIMIT**, **LOW**, **PEAK**

RAVERAGE

Define a task that computes the running average of a data stream.

RAVERAGE *(<in_pipe>, <count>, <out_pipe>,*
[<phase_correction>])

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<count>
A positive integer that specifies the number of values used in each averaging calculation.
`WORD CONSTANT`

<out_pipe>
Output data pipe.
`WORD PIPE`

<phase_correction>
An optional value that specifies a time-shift correction for filtering applications.
`WORD CONSTANT`

Description

For each value **RAVERAGE** reads from *<in_pipe>*, it computes the average of the last *<count>* values, and puts this running average into *<out_pipe>*. *<count>* is a positive integer. Unlike the **AVERAGE** command that computes one result per block of data, **RAVERAGE** computes a new result for each new value.

RAVERAGE begins generating output only after *<count>* samples are read. This produces a "time lag" between the input and output streams. This can be important in triggering applications because a trigger assertion generated from averaged data contains a different sample count from the sample count of a trigger assertion from unaveraged data. If the optional parameter *<phase_correction>* is present, the first averaged result is repeated *<count>* times so that sample counts from **RAVERAGE** correspond to unaveraged sample counts. The value of *<phase_correction>*, if present, should be zero.

Example

```
RAVERAGE (P1, 10, P2)
```

Reads data from pipe `P1`, compute the averages of moving windows of 10 values, and put the average values in pipe `P2`.

See Also

AVERAGE, **BAVERAGE**, **FIRFILTER**

REPLICATE

Define a task that replicates data values.

REPLICATE *(<in_pipe>, <cnt>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<cnt>
A value that specifies the number of copies of each data value to transfer.
`WORD CONSTANT`

<out_pipe>
Output data pipe.
`WORD PIPE | LONG PIPE`

Description

REPLICATE reads data values from *<in_pipe>* and places *<cnt>* copies of each data value into *<out_pipe>*.

Example

```
REPLICATE (P1, 3, P2)
```

Transfer three copies of each data value from `P1` into pipe `P2`.

See Also

COPY, **MERGE**

RESET

Stop all configurations and clear the current configuration.

RESET

Description

RESET stops all configurations, clears all memory, and erases all DAPL symbols except communication pipes and downloaded command modules.

RESET does not affect **OPTIONS** settings.

See Also

EMPTY, **ERASE**

RMS

Define a task that calculates the root-mean-square average of data blocks.

RMS *(<in_pipe>, <n>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<n>
Number of data values per block.
`WORD CONSTANT`

<out_pipe>
Output data pipe.
`WORD PIPE`

Description

RMS reads blocks of *<n>* data values from *<in_pipe>*, calculates the square root of the average of the squares of the *<n>* data values, and writes the result to pipe *<out_pipe>*. **RMS** is an abbreviation for Root Mean Square.

Example

```
RMS(P1, 256, P2)
```

Calculate **RMS** values of blocks of 256 values from `P1` and place the results in `P2.`

See Also

AVERAGE, **VARIANCE**

SAMPLEHOLD

Pause DAPL command processing until all sampled data is processed.

SAMPLEHOLD

Description

SAMPLEHOLD pauses DAPL until the currently active input configuration finishes sampling and all input channel pipes are empty. The **SAMPLEHOLD** command guarantees that all data are taken for processing, and the final PAUSE allows enough time to finish the last processing and complete data transfers to the PC.

SAMPLEHOLD can be used only when the active input configuration has a **COUNT** specification. It should not be used with **UPDATE** BURST mode.

Example

```
START <input_configuration>
SAMPLEHOLD
PAUSE 500
STOP <input_configuration>
```

After starting an input configuration, wait for sampling initiated by hardware triggering to finish. Wait an additional 1/2 second after sampling has finished to allow processing of data to complete. Then stop everything.

See Also

COUNT, **START**, **STOP**, **UPDATE**

SAWTOOTH

Define a task that generates sawtooth wave data.

```
SAWTOOTH (<amplitude>, <period>, <out_pipe>
          [,<mod_type>, <mod1> [, <mod2>]])
```

Parameters

<amplitude>
A value that is one half of the peak to peak range of the output.
`WORD CONSTANT | WORD VARIABLE`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for sawtooth wave data.
`WORD PIPE`

<mod_type>
A value that selects amplitude and/or frequency modulation of the output wave.
`WORD CONSTANT`

<mod1>
Pipe for first modulation signal.
`WORD PIPE`

<mod2>
Pipe for second modulation signal.
`WORD PIPE`

Description

SAWTOOTH generates sawtooth wave data and places the data in *<out_pipe>*. *<period>* is the number of sample values in each wave. The *<amplitude>* is one half the peak to peak distance of the output wave. The maximum value of *<amplitude>* is 32767.

Note: **SAWTOOTH** is identical to **WAVEFORM**, with *<mod_type>* set equal to 1.

Three optional modulation parameters may be specified. *<mod_type>* selects amplitude and/or frequency modulation of the output wave. The value of *<mod_type>* must be one of the following:

1. amplitude modulation controlled by the data in *<mod1>*
2. frequency modulation controlled by the data in *<mod1>*
3. amplitude and frequency modulation controlled by the data in *<mod1>* and *<mod2>*, respectively

<mod1> and *<mod2>* are pipes. One value is read from the pipe(s) for each value output by **SAWTOOTH**. Modulation values are interpreted as signed binary fractions; they are multiplied by the base amplitude or frequency to obtain the amplitude or frequency.

An alternative method for changing the amplitude or frequency of **SAWTOOTH** during execution uses a DAPL variable as the *<amplitude>* or *<period>* parameter of **SAWTOOTH**. This variable can be changed during execution using a **LET** command. This is efficient, but cannot adjust the amplitude or frequency continuously, and changes are detected and applied asychronously.

Example

```
SAWTOOTH (1000, 100, P2)
```

Generate a sawtooth wave with values ranging from -1000 to 1000, with a period of 100 samples.

See Also

COSINEWAVE, **SINEWAVE**, **TRIANGLE**, **SQUAREWAVE**, **WAVEFORM**

SCALE

Define a task that applies gain and offset corrections to data.

SCALE *(<in_pipe>, <X>, <Y>, <Z>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<X>
Scalar offset to apply to data.
`WORD CONSTANT | WORD VARIABLE`

<Y>
Scalar multiplier to apply to data.
`WORD CONSTANT | WORD VARIABLE`

<Z>
Scalar divisor to apply to data.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output data pipe.
`WORD PIPE | LONG PIPE`

Description

SCALE reads a value from *<in_pipe>*, transforms the value, and puts the result into *<out_pipe>*. The following formula summarizes the arithmetic operations performed by **SCALE**, where the data value is represented by the letter `D`:

```
(D + X) * Y
-----------
     Z
```

`X`, `Y` and `Z` can be negative; `Z` must not be zero.

Note: A DAPL expression can also perform this scaling operation, and is more general but slightly slower.

Examples

```
SCALE (P3, -100, 3, 2, P4)
```

Subtract 100 from data values and then multiply by 3/2.

```
SCALE (P5, 0, -5000, -32768, P6)
```

Convert -5 to +5 volt bipolar A/D counts into millivolt readings.

See Also

OFFSET

SCAN

Define a task that copies sets of input samples as a unit.

SCAN *(<in_pipe>, <p1> [, <p2>]*)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<p1>
First output data pipe.
`WORD PIPE`

<p2>
Additional output data pipes.
`WORD PIPE`

Description

SCAN reads data from *<in_pipe>* and puts the data into one or more output pipes. The input pipe is an input channel list pipe. **SCAN** does not transfer any data until a sample is available for each channel pipe in the input channel pipe list. Then **SCAN** transfers the block of input channel pipe data into its output pipes.

SCAN is most useful in low latency applications that require one sample from each channel. See Chapter 13 for more information.

Example

```
SCAN (IPIPES(0,1,2,3,4), P1)
```

Transfer data from input channel pipes 0, 1, 2, 3, and 4 to the pipe `P1` in groups of five samples, one per channel.

See Also

COPY, **LCOPY**, **OPTIONS**

SDISPLAY

Print information about individual DAPL symbols.

SDISPLAY *<symbol>[, <symbol>]**

SDISP *<symbol>[, <symbol>]**

SD *<symbol>[, <symbol>]**

Parameters

<symbol>
Symbol name.

Description

SDISPLAY (symbol display) formats information about individual DAPL symbols and sends the information to the `$SYSOUT` pipe for display. For example, if a processing procedure symbol is given, the contents of the processing procedure are displayed. If a variable name is specified, the current value of the variable is displayed.

Example

```
SDISPLAY V1,P1,T1
```

Display information about symbols `V1`, `P1`, and `T1.`

See Also

DISPLAY, **VARIABLES**

SEPARATE

Define a task that distributes data consecutively into one or more output pipes.

SEPARATE *(<in_pipe>, <out_pipe_0>, ... , <out_pipe_n-1>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe_0>
First output pipe for separated data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe_n-1>
Last output pipe for separated data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

SEPARATE reads a stream of mixed data from *<in_pipe>* and places the data consecutively into one or more output pipes. One value is sent to *<out_pipe_0>*, the next value is sent to *<out_pipe_1>*, and so forth. After all output pipes have received a value, the cycle repeats.

Depending on the data type of the *<in_pipe>* and the data type of the output pipe receiving the data, values from the input stream are processed in one of several ways:

- Copy — copied with no change to value or type,
- Alias — copied in a manner that preserves bit patterns but interprets the patterns as a different data type,
- Splice — combined with additional values from the input stream to reconstruct a number of higher precision,
- Convert — converted to the data type of the output pipe in a manner that preserves value,
- Not allowed — conversions that could result in loss of precision or range errors are *not supported*.

The following table shows which operations are applied for each combination of *<in_pipe>* and *<out_pipe>* data types.

<in_pipe>	*<out_pipe>*			
	WORD	LONG	FLOAT	DOUBLE
WORD	copy	splice	splice	splice
LONG	convert	copy	alias	splice
FLOAT	convert	----	copy	convert
DOUBLE	convert	convert	convert	copy

Each conversion operation reverses the conversion that the **MERGE** command would apply to construct the merged stream. See the conversion table provided for the **MERGE** command.

Note: Be careful when applying the **SEPARATE** command to data streams not constructed by the **MERGE** command. It is possible to construct numerically valid streams of data that **MERGE** could not have produced. Results of applying the **SEPARATE** command to such a data stream are undefined.

Examples

```
SEPARATE (PW1,PL2,PF3,PD4)
```

Read 16-bit WORD data from pipe P1, taking as many values as necessary to construct higher precision numbers placed consecutively into LONG pipe PL2, FLOAT pipe PF3, and DOUBLE pipe PD4.

```
SEPARATE ($BININ,P5,P6,P7)
```

Transfer data from the PC host to the Data Acquisition Processor through the binary input com pipe, placing successive values into WORD pipes P5, P6, and P7.

See Also

MERGE, **MERGEF**, **SEPARATEF**

SEPARATEF

Define a task that distributes flagged data.

SEPARATEF *(<in_pipe>, <out_pipe_0>, ... , <out_pipe_n-1>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<out_pipe_0>
First output pipe for separated data.
`WORD PIPE | LONG PIPE`

<out_pipe_n-1>
Last output pipe for separated data.
`WORD PIPE | LONG PIPE`

Description

SEPARATEF reads flagged data from *<in_pipe>*, removes the flags, and writes each data value to one of *<out_pipe_0>*, … , *<out_pipe_n-1>*. Each flagged data value consists of two values, a flag from 0 to n-1, which identifies the destination pipe, and a data value to be placed in the pipe. Flagged data values with flags out of range are detected and cause the **SEPARATEF** task to terminate with an error.

When *<in_pipe>* is a word pipe and an output is a long pipe, two consecutive words, low word then high word, are read from the input pipe and concatenated to form a long output value.

When *<in_pipe>* is a long pipe and an output is a word pipe, the low word of the long value is transferred, and the high word is ignored.

SEPARATEF is useful for reading binary data from a host computer and splitting the binary data stream into several pipes for processing. **SEPARATEF** is the inverse of **MERGEF**.

Examples

```
SEPARATEF (P1,P2,P3,P4)
```

Read flagged data from pipe `P1` and place data selectively into pipes `P2`, `P3`, and `P4`.

```
SEPARATEF ($BININ,P5,P6,P7)
```

Transfer data from the binary input com pipe to pipes `P5`, `P6`, and `P7`.

See Also

MERGE, **MERGEF**, **SEPARATE**

SET (individual channel sampling)

Associate an individual input channel pipe with an input pin.

SET *<channel> <input_pin> [<gain>]*

Parameters

<channel>
Input channel pipe identifier.

<input_pin>
Input pin identifier.

<gain>
An integer number that specifies the gain.
`WORD CONSTANT`

Description

SET associates an individual input channel pipe (`IPIPE`) with an input pin. Models of Data Acquisition Processor that sample multiple channels simultaneously use a different version of this command – see the **SET** command version described in the next section. Output updating configurations use another version of this command, also in this chapter.

A *<channel>* identifier consists of an `IPIPE` keyword followed by a decimal number. It assigns a name to a data channel. The `IPIPE` keyword may be abbreviated to `IP.` Some examples:

```
IP7
IPIPE482
```

The range of the channel identifier numbers is restricted by the Data Acquisition Processor model. When the sampling configuration runs, it will capture samples in order of channel identifier numbers rather than by order of appearance within the `IDEFINE` section.

An *<input_pin>* identifier begins with a identifier character. The pin type identifier characters `S`, `D`, and `G` represent single-ended, differential, and ground reference analog inputs, respectively. The identifier letter is followed immediately by a number to identify the hardware pin. One physical pin can be sampled into multiple input channels and thus appear on more than one **SET** command.

A separate *<gain>* number can follow. If *<gain>* is omitted, it defaults to 1. The allowed gains are 1, 10, 100, or 500. See the Data Acquisition Processor hardware manual for information about sampling rate limitations at each gain. The following are examples of analog channel specifiers and gains:

```
S2 100
D0
G  10
```

A pin type identifier character `B` indicates a binary (digital) input source. A number follows immediately to indicate the digital port. The digital port number is optional, so identifier `B` means the same thing as `B0`. The following are examples of digital *<input_pin>* identifiers.

```
B4
B
```

The analog and digital pin numbers are restricted according to the physical signals available on the Data Acquisition Processor and attached accessory boards. Digital or analog expansion boards increase the number of available physical digital or analog signals, extending the range of meaningful pin numbers. See the hardware documentation for each Data Acquisition Processor and accessory board type for more information about the available pin numbers.

When digital and analog input pins are used in the same input configuration, a digital input is acquired one sampling period later than a corresponding analog input. See the Data Acquisition Processor hardware documentation for more information about the timing of input channel sampling.

When an external Counter/Timer Board is connected to the digital input/output port of a Data Acquisition Processor, two additional *<input_pin>* names are valid: `CTLx` and `CTy`, where '`x`' is the number 0 or 1 and '`y`' is a number from 0 to 9. The sample value produced by a `CTLx` sampling operation is not meaningful, but the operation freezes the values of all counters in counter circuit '`x`' (Counter Timer Load). A `CTy` sampling operation (Counter Timer read) reads the value of input counter '`y`'. See the Counter/Timer Board documentation for more information.

A DAPL task can read sampled data using an input channel pipe notation in a processing task definition. A processing task can read from a single input channel pipe, using the notation `IPIPE`*x* or `IP`*x*, where x is a number indicating an input channel similar to the **SET** command. A processing task can read from several input channel pipes using an input channel pipe list notation beginning with the name `IPIPES`, and followed immediately by a list of input channel pipe numbers enclosed in parentheses. A range of consecutive channel pipe numbers can be selected by specifying the first channel number, two consecutive periods, and the last channel

number. The identifier IPIPES can be abbreviated to IPIPE or IP. Input channel pipe numbers must appear in ascending order and must not be repeated. The channel pipe numbers correspond to the channel pipe identifiers assigned on **SET** commands. The following is an example of a task parameter list using a mix of single-channel and channel range specifications to copy ten channels.

```
COPY (IP(0..3,10..13,22,23),$binout)
```

A channel list also can use a named word vector defined by a **VECTOR** command. The following example shows equivalent notations to access input channels 0 through 4.

```
VECTOR CLIST WORD = (0, 1, 2, 3, 4)

IPIPES(0,1,2,3,4)
IPIPES CLIST
```

Examples

```
SET IPIPE0 S4
```

Input channel pipe 0 contains samples from single-ended input 4, at unity gain.

```
SET IPIPE1 D5 10
```

Input channel pipe 1 contains samples from differential input 5, with a gain of 10.

```
SET IP2 G 100
```

Input channel pipe 2 contains samples of a ground reference measured with gain of 100.

```
SET IP3 B0
```

Input channel pipe 3 contains samples from the binary input port.

See Also

IDEFINE, **ODEFINE**, variant of **SET** for multiple channel simultaneous sampling, variant of **SET** for single channel output updating

SET (multiple channel simultaneous sampling)

Associate grouped input channels with an input sampling pin group.

SET *<channel_group>* *<pin_group>*

Parameters

<channel_group>
Input channel pipe identifier.

<pin_group>
Input pin group identifier.

Description

SET associates an input channel group (IPIPE) with an input pin group. Models of Data Acquisition Processors that sample individual channels use a different kind of **SET** command – see the **SET** command version described in the previous section. Output updating configurations use another version of the **SET** command, described in the next section.

The logical data channels to be assigned are specified by the *<channel_group>*. The group of signal pins is specified by the *<pin_group>*. If the Data Acquisition Processor hardware provides a programmable ground reference signal source, this source is the default signal source. Any channel group not assigned to a signal pin group by a **SET** command will receive samples from the ground reference source. If the Data Acquisition Processor hardware does not provide a programmable ground reference signal source, a **SET** command is required for each channel group specified by the **GROUPS** command.

A *<channel_group>* identifier consists of an IPIPE keyword followed by a restricted form of channel list. When the sampling configuration runs, it will capture samples in order of channel group identifier numbers rather than by order of appearance within the IDEFINE section. The channel list is a pair of decimal numbers separated by two periods and enclosed in parentheses. The IPIPES keyword may be abbreviated to IPIPE or IP. Some examples:

```
IPIPES(4..7)   // 4 pin group, channels 4 through 7
IPIPE(0..3)    // 4 pin group, channels 0 through 3
IP(16..23)     // 8 pin group, channels 16 through 23
```

Only certain numbers are allowed to begin and end the special lists. The range must start with an integer that is a multiple of the channel group size supported by the Data Acquisition Processor hardware. The range must cover the exact group size. The range limits for the channel pipe numbers depend on the Data Acquisition Processor model. When the hardware supports a group size of 4, the following ranges are acceptable:

```
IP(0..3)
IP(4..7)
IP(8..11)
...
```

When the hardware supports a group size of 8, the following ranges are acceptable.

```
IP(0..7)
IP(8..15)
IP(16..23)
...
```

When the sampling configuration runs, it will capture samples in order of channel identifier numbers rather than by order of appearance within the **IDEFINE** section.

A group of simultaneously sampled pins is specified by the notation `SPGx`, where `x` is an integer. The pin groupings are predefined. The range of the pin numbers depends on the Data Acquisition Processor model and attached expansion cards. For a Data Acquisition Processor model with analog input pin groups of size 4, the mapping of physical input pins to pin groups is as follows:

```
SPG0   S0, S4, S8, S12
SPG1   S1, S5, S9, S13
SPG2   S2, S6, S10, S14
SPG3   S3, S7, S11, S15
SPG4   S16, S20, S24, S28
SPG5   S17, S21, S25, S29
...
```

For a Data Acquisition Processor model with pin groups of size 8, expandable to 1024 inputs, the mapping of physical input pins to pin groups is as follows:

```
SPG0   S0, S2, S4, S6, S8, S10, S12, S14
SPG1   S1, S3, S5, S7, S9, S11, S13, S15
SPG2   S16, S18, S20, S22, S24, S26, S28, S30
SPG3   S17, S19, S21, S23, S25, S27, S29, S31
SPG4   S32, S34, S36, S38, S40, S42, S44, S46
...
```

There are variations in the pin mapping scheme for certain analog expansion boards. These are discussed in the hardware manuals for each individual hardware board.

A DAPL task can read sampled data using an input channel pipe or input channel pipe list notation in a processing task definition, without regard to the grouping imposed as the data is sampled. To read from a single channel, use the notation `IPIPE<number>` or `IP<number>`. To read multiplexed data from several input channels, use an input channel pipe list notation that begins with `IPIPES`, followed immediately by a list of input channel pipe numbers enclosed in parentheses. The identifier `IPIPES` can be abbreviated to `IPIPE` or `IP.` Input channel pipe numbers must appear in ascending order and must not be repeated. The channel pipe numbers must be within the channel pipe ranges assigned on **SET** commands. A range of channel pipe numbers can be selected by specifying the first channel number, two consecutive periods, and the last channel number. The following is an example of a task parameter list using a mix of single-channel and channel range specifications to copy twelve channels.

```
COPY (IP(0..3,10..15,22,23),$binout)
```

Examples

```
IDEFINE A
  GROUPS 3
  SET IP(0..3)  SPG3
  SET IP(4..7)  SPG0
  SET IP(8..11) SPG1
  TIME 100
END
PDEFINE B
  AVERAGE(IP0,100,$BINOUT)
END
```

For a Data Acquisition Processor that has input channel groups of size 4, associate the channel pipes `IP0-IP3` to pins `S3`, `S7`, `S11`, and `S15`; channel pipes `IP4-IP7` to pins `S0`, `S4`, `S8`, and `S12`; and channel pipes `IP8-IP11` to pins `S1`, `S5`, `S9`, and `S13`. Process only the data from channel 0 from channel group `IP(0..3)`.

```
  SET IP(0..7)  SPG0
  SET IP(8..15) SPG1
```

For a Data Acquisition Processor that has input channel groups of size 8, specify two input channel groups, one connected to pins S0, S2, S4, S6, S8, S10, S12 and S14, and the other to pins S1, S3, S5, S7, S9, S11, S13 and S15. These signals are recorded in logical channels 0 through 15.

See Also

GROUPS, **IDEFINE**, **ODEFINE**, variant of **SET** for individual channel sampling, variant of **SET** for single channel output updating, **VRANGE**

SET (single channel output updating)

Associate an output channel pipe with a clocked output pin.

SET *<channel>* *<output_pin>*

Parameters

<channel>
Output channel pipe specifier.

<output_pin>
Output pin specifier.

Description

SET associates an individual output channel pipe (OPIPE) with a clocked output pin. See the preceding variants of the **SET** command for configuring input sampling.

A *<channel>* identifier consists of an OPIPE keyword followed by a decimal number. It assigns a name to a data channel. The OPIPE keyword may be abbreviated to OP. Some examples:

```
OP7
OPIPE62
```

The range of the channel identifier numbers is restricted by the Data Acquisition Processor model. The *<output_pin>* specifier can be A0, A1, or B0 without using output expansion boards. Output pins A0 and A1 are the two analog output ports; B0 is the digital output port. The pin number following B is optional; the notation B without a number is equivalent to B0. More output pins are available with output expansion.

A task can provide output data by writing to an output channel pipe identified in the task parameter list by the notation OPIPE, followed immediately by the input channel pipe number. OPIPE can be abbreviated to OP.

To enforce the restriction that data are written to the output channel pipe in a strict multiplexed order, the output channel list notation is very helpful. An output channel list parameter in a processing task definition consists of the identifier OPIPES followed immediately by a list of the output channels enclosed in parentheses. The output channel pipe numbers in the list correspond to the channels assigned on the **SET** commands. OPIPES can be abbreviated to OPIPE or OP. A range of channel pipe numbers is selected by specifying the first channel number, two consecutive

periods, and the last channel number. An example of a task that writes four channels for clocked output updates:

```
MERGE  (P0,P1,P2,P3, OPIPE(0..3))
```

Examples

```
ODEFINE  D  2
  SET OPIPE0 A1
  SET OPIPE1 B
  TIME 50
END
```

Define output updating with one analog output to analog output pin A1 and one digital output to binary port 0.

See Also

ODEFINE

SINEWAVE

Define a task that generates sine wave data.

SINEWAVE *(<amplitude>, <period>, <out_pipe> [,<mod_type>, <mod1> [,<mod2>]])*

Parameters

<amplitude>
A value that is one half of the peak to peak range of the output.
`WORD CONSTANT | WORD VARIABLE`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for sine wave data.
`WORD PIPE`

<mod_type>
A value that selects amplitude and/or frequency modulation of the output wave.
`WORD CONSTANT`

<mod1>
Pipe for first modulation signal.
`WORD PIPE`

<mod2>
Pipe for second modulation signal.
`WORD PIPE`

Description

SINEWAVE generates sine wave data and places the data in *<out_pipe>*. *<period>* is the number of sample values in each wave. The *<amplitude>* is one half the peak to peak distance of the output wave. The maximum value of *<amplitude>* is 32767.

Note: **SINEWAVE** is identical to **WAVEFORM**, with *<type>* set equal to 2.

Three optional modulation parameters may be specified. *<mod_type>* selects amplitude and/or frequency modulation of the output wave. The value of *<mod_type>* must be one of the following:

1. amplitude modulation controlled by the data in *<mod1>*
2. frequency modulation controlled by the data in *<mod1>*
3. amplitude and frequency modulation controlled by the data in *<mod1>* and *<mod2>*, respectively

<mod1> and *<mod2>* are pipes. One value is read from the pipe(s) for each value output by **SINEWAVE**. Modulation values are signed binary fractions and are multiplied by the base amplitude or frequency to obtain the amplitude or frequency.

An alternative method for changing the amplitude or frequency of **SINEWAVE** during execution uses a DAPL variable as the *<amplitude>* or *<period>* parameter of **SINEWAVE**. The value of this variable can be changed during execution using **LET**. This is efficient, but cannot adjust the amplitude or frequency continuously, and changes are detected and applied asynchronously.

Example

```
SINEWAVE (1000, 100, P2)
```

Generate a sine wave with values ranging from -1000 to 1000, with a period of 100 samples.

See Also

COSINEWAVE, **SAWTOOTH**, **TRIANGLE**, **SQUAREWAVE**, **WAVEFORM**

SKIP

Define a task that alternately copies and skips data.

SKIP *(<in_pipe>, <initial_skip>, <take_cnt>, <skip_cnt>*
<out_pipe>)

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<initial_skip>
A value that specifies the initial number of values to skip.
`WORD CONSTANT | LONG CONSTANT`

<take_cnt>
A value that specifies the number of values to move.
`WORD CONSTANT | LONG CONSTANT`

<skip_cnt>
A value that specifies the number of values to skip.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>
Output data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

SKIP provides flexible options for moving selected data from *<in_pipe>* to *<out_pipe>.* After doing a one time skip of *<initial_skip>* values, **SKIP** repeats a cycle of moving *<take_cnt>* values to *<out_pipe>* then ignoring *<skip_cnt>* values. The parameters *<initial_skip>, <take_cnt>* and *<skip_cnt>* must be all non-negative. If the value of *<take_cnt>* is zero, the effect is to discard all data. If the value of *<take_cnt>* is positive but the value of *<skip_cnt>* is zero, the effect is to copy all data following the initial *<initial_skip>* values.

Applications of the **SKIP** command include data selection, reducing volumes of data to process, and decimation of signal streams.

Examples

```
SKIP (IP0, 0, 1000, 2000, P1)
```

Transfer 1000 values to `P1`, ignore a block of 2000 values, and repeat.

```
SKIP (IP0, 100, 500, 100, P1)
```

Ignore 100 values from `IP0`, transfer 500 values to `P1`, then repeat.

```
SKIP (P1, 50, 1, 0, P2)
```

Ignore first 50 values from `P1`, then continuously transfer remaining data.

SLAVE

Configure an input or output configuration's clock source to be another Data Acquisition Processor.

SLAVE

Description

The **SLAVE** command configures an input or output configuration's clock source to be another Data Acquisition Processor. The **SLAVE** command is used in synchronized multiple Data Acquisition Processor systems; a slave Data Acquisition Processor synchronizes input sampling or output updates to a clock signal from a master Data Acquisition Processor.

If **SLAVE** is used in an output configuration, the **OUTPUTWAIT** count must be satisfied before the master output configuration is started.

Note: **UPDATE** BURST mode is not available with **SLAVE**.

See Also

CLOCK, **HTRIGGER**, **MASTER**

SQRT

Define a task that computes square roots of data.

SQRT *(<in_pipe>, <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<out_pipe>
Output pipe for square root data.
`WORD PIPE | LONG PIPE`

Description

SQRT computes square roots of data from *<in_pipe>* and places the results in *<out_pipe>*. If an input data value is negative, **SQRT** sends the number zero to the output pipe. The returned value is the greatest integer lower bound on the exact square root value.

Example

```
SQRT (P1, P2)
```

Read data from pipe `P1`, compute square roots, and place the results in pipe `P2.`

See Also

CABS, **RMS**

SQUAREWAVE

Define a task that generates square wave data.

SQUAREWAVE *(<amplitude>, <period>, <out_pipe> [,<mod_type> <mod1> [, <mod2>]])*

Parameters

<amplitude>
A value that is one half of the peak to peak range of the output.
`WORD CONSTANT | WORD VARIABLE`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for square wave data.
`WORD PIPE`

<mod_type>
A value that selects amplitude and/or frequency modulation of the output wave.
`WORD CONSTANT`

<mod1>
Pipe for first modulation signal.
`WORD PIPE`

<mod2>
Pipe for second modulation signal.
`WORD PIPE`

Description

SQUAREWAVE generates square wave data and places the data in *<out_pipe>*. *<period>* is the number of sample values in each wave. The *<amplitude>* is one-half the peak to peak distance of the output wave, with a maximum value of 32767.

Note: **SQUAREWAVE** is identical to **WAVEFORM**, with *<type>* set equal to 3.

Three optional modulation parameters may be specified. *<mod_type>* selects amplitude and/or frequency modulation of the output wave. The value of *<mod_type>* must be one of the following:

1. amplitude modulation controlled by the data in *<mod1>*
2. frequency modulation controlled by the data in *<mod1>*
3. amplitude and frequency modulation controlled by the data in *<mod1>* and *<mod2>*, respectively

<mod1> and *<mod2>* are pipes. One value is read from the pipe(s) for each value output by **SQUAREWAVE**. Modulation values are interpreted as signed binary fractions; they are multiplied by the base amplitude or frequency to obtain the amplitude or frequency.

An alternative method for changing the amplitude or frequency of **SQUAREWAVE** during execution uses a DAPL variable as the *<amplitude>* or *<period>* parameter of **SQUAREWAVE**. The value of this variable can be changed during execution using a **LET** command. This is efficient, but it cannot adjust the amplitude or frequency continuously, and changes are detected and applied asynchronously.

Example

```
SQUAREWAVE (1000, 100, P2)
```

Generate a square wave with values ranging from -1000 to 1000, with a period of 100 samples.

See Also

COSINEWAVE, **SAWTOOTH**, **SINEWAVE**, **TRIANGLE**, **WAVEFORM**

START

Activate input configurations, processing procedures, and output configurations.

START *[<name> [,<name>]*]*

STA *[<name> [,<name>]*]*

Parameters

<name>

Name of an input, output, or processing configuration.

Description

START activates input configurations, processing procedures, and output configurations.

START can be used with no parameters to start all defined configurations. Only one input configuration and one output configuration should be defined when using **START** without parameters. When more than one input configuration or output configuration is defined, the **START** command with no parameters will generate a warning message and start the first configuration defined.

Activation of an input configuration initializes the Data Acquisition Processor hardware and software to allow sampling, as defined by the input configuration's sampling configuration. Once the hardware begins sampling the input pins, sampling continues until the sample count reaches the input configuration's **COUNT** specification or until a **STOP** command is issued. Only one input configuration can be active at a given time.

Activation of a processing procedure starts each of the tasks in the processing procedure. Any number of processing procedures can be active at one time. Starting a processing procedure does not affect the Data Acquisition Processor input sampling or output update status.

Activation of an output configuration initializes the Data Acquisition Processor hardware and software to allow output updating, as defined by the output configuration. Once the hardware begins updating the output pins, updating continues until the output sample count reaches the output configuration's **COUNT** specification or until a **STOP** command is issued. Only one output configuration can be active at a given time.

When starting an input or output configuration that specifies an external clock or trigger, sampling might not begin immediately. See the hardware documentation for details about external clocks and triggers.

When stopping and restarting an input or output configuration, it is best to use the **STOP** and **START** commands with no parameters to perform a complete stop and restart. It is possible, however, to stop an input or output configuration while other processing continues. In this case, it also is necessary to stop the tasks that read from or write to the input or output configuration channel pipes. To restart the input or output configuration, start the input or output configuration and then start the input or output tasks.

Examples

```
START
START A
START A,B
```

See Also

RESET, **STOP**

STATISTICS

Report processor utilization information about the system and running tasks.

STATISTICS *ON | [DISPLAY] | OFF*

STAT *ON | [DISPLAY] | OFF*

Description

The **STATISTICS** command displays information about CPU utilization of the system and processing tasks. To initiate collection of the measurements, issue the **STATISTICS** command with the ON command line option. After **STATISTICS** collection is ON, pause for a few seconds, then issue a **STATISTICS** DISPLAY command, or simply a **STATISTICS** command with no command line options, to see the collected information. **STATISTICS** can be displayed multiple times, but there is some interaction between the measured CPU utilization and the utilization of the **STATISTICS** command itself, so the first measurement will be the most accurate. When the statistics are no longer needed, issue the **STATISTICS** command with the OFF command line option so there is no possibility that the **STATISTICS** processing will interfere with other processing.

The abbreviated command form **STAT** is useful when entering DAPL commands interactively.

The report displayed by **STATISTICS** might look something like the following:

```
Task                                CPU Time Used
HOST_TSK                               3555
MEM_TSK                                  0
*DAPL*                                 199390
INF_TSK                                  0
CFG_TSK                                  0
OVR_CHK                                272
UND_CHK                                272
MEM_TSK                                407
COPY                                   281831
COPY                                   266681
FORMAT                                 267887
system idle/overhead                   986654
```

```
Total time elapsed:                    2006000
Average task cycle latency               890
Longest task cycle latency              3445
   (all numbers are in uS)
```

All measured times are in microsecond units. The table shows the total time in microseconds used by each task running at the time that statistics are taken. The first tasks on the list are DAPL system tasks. After that, the names of the processing commands appear.

A large value for system idle/overhead typically indicates that the Data Acquisition Processor is performing well. All processing is completed and there is reserve CPU capacity. When the Data Acquisition Processor has extra capacity, it spends much of its time switching tasks while it waits for data. As the demands on the Data Acquisition Processor increase, it spends more time processing data and less time switching tasks. Its efficiency increases, as the amount of time spent on system overhead decreases. On the other hand, a heavily-loaded CPU cannot respond to real-time events as quickly.

The average task latency is the typical number of microseconds of real time between activations of a particular task. If tasks typically are not busy, but are very busy when data arrives, the longest task cycle latency might be very much larger than the average latency. This is usually okay, but if the longest latency gets too large, this could indicate problems for systems needing fast real-time response.

STATUS

Display information about the current status of the system.

STATUS

STAT

Description

STATUS displays information about the current status of the system — memory usage, active system tasks, interrupt status variables, etc. In most cases, this information is of interest to system implementers only. The **STATUS** command should not be used while an input or output configuration is active.

STOP

Stop input sampling, processing, and output updating configurations.

STOP *[<name> [,<name>]*]*

Parameters

<name>

Name of input, output, or processing configuration.

Description

The **STOP** command with no parameters stops input sampling and output updating, stops all tasks, empties all pipes, and clears all triggers. The Data Acquisition Processor stops producing output and flushes all data.

The **STOP** command with a list of names stops each input configuration, output configuration, or processing procedure named.

Stopping an input configuration results only in termination of input pin sampling. Processing of buffered data in input channel pipes and in pipes continues.

Stopping a processing procedure stops all the tasks in the processing procedure.

Stopping an output configuration stops the updating of the analog or digital outputs. Processing of buffered data in pipes continues.

Note: Only the variant of **STOP** without parameters flushes buffered data. **STOP** without parameters is the recommended way to stop an application.

Examples

```
STOP A
STOP A,B
STOP
```

See Also

RESET, **START**

STRING

Define a string.

STRING *<name>* = *“<text>“*

STR *<name>* = *“<text>“*

Parameters

<name>
String name.

<text>
String text.

Description

STRING defines a string. Strings can be used by the **FORMAT** command to include alphanumeric information on printed lines. Enclose the text in double quote characters. To include a quote character in the content of a string, use two quote characters in sequence.

A **STRING** also is useful for passing configuration information to custom command tasks.

Example

```
STRING HEADING = ¨MAXIMUM AT PEAK:¨
```

TAND

Define a task that calculates a logical ‘and’ of trigger assertions.

TAND *(<in_trigger_1>, ... , <in_trigger_n>,*
<out_trigger> [, <delta>])

Parameters

<in_trigger_1>
First input trigger.
TRIGGER

<in_trigger_n>
Last input trigger.
TRIGGER

<out_trigger>
Output trigger.
TRIGGER

<delta>
Tolerance specification for almost simultaneous events.
WORD CONSTANT

Description

TAND is used to detect simultaneous or near-simultaneous events. **TAND** calculates a logical ‘and’ of trigger assertions. Each time that triggers *<in_trigger_1>*, … , *<in_trigger_n>* are all asserted within *<delta>* sample times, *<out_trigger>* is asserted. When this occurs, *<out_trigger>* is asserted at the earliest of the times in *<in_trigger_1>*, … , *<in_trigger_n>*, and one trigger assertion is removed from each of *<in_trigger_1>*, … , *<in_trigger_n>*. *<delta>* is an optional parameter; its default value is zero. When *<delta>* is zero, the timestamps from *<in_trigger_1>*, … , *<in_trigger_n>* must match exactly to generate an output event.

Any other trigger events that do not satisfy the conditions for sending an assertion to *<out_trigger>* are removed from *<in_trigger_1>*, … ,*<in_trigger_n>* and ignored.

Example

```
TAND (T1, T2, T3, T_OUT, 25)
```

Assert `T_OUT` each time `T1`, `T2`, and `T3` all are asserted within an interval of 25 sample times.

See Also

TOR

TCOLLATE

Define a task that combines trigger assertions and produces a combined event stream.

TCOLLATE *(<trig_0>, ..., [<trig_n-1>,] <trig_out>)*

Parameters

<trig_0>
First source of events.
TRIGGER

<trig_n-1>
Subsequent sources of events.
TRIGGER

<trig_out>
Output trigger for the combined event stream.
TRIGGER

Description

The **TCOLLATE** command combines trigger assertions from n triggers *<trig_0>* through *<trig_n-1>* in sequence, where n is in the range 2 to 16. It produces a combined event stream in trigger *<trig_out>*. Timestamps from the n input triggers, *<trig_0>* through *<trig_n-1>*, must be based on the same data rates.

The input triggers *<trig_0>* through *<trig_n-1>* are processed in sequence. When an event appears in *<trig_0>*, it is copied to *<trig_out>*. Next, trigger *<trig_1>* is processed, discarding any events prior to the event timestamp taken from trigger *<trig_0>*. When a suitable event appears, it is copied to *<trig_out>*. Processing continues in this manner for each input trigger in the list. When the list is exhausted, processing begins again at the start of the list. This processing sequence ensures that the events posted in trigger *<trig_out>* are in a strictly increasing time sequence.

A common application for the **TCOLLATE** command is enforcing a strict alternating sequence of trigger events from two independent triggering tasks. These events might be interpreted, for example, as ON events alternating with OFF events. This kind of alternating sequence is required by the **TOGGWT** command. Extremely general triggering conditions can be defined for the **TOGGWT** command using a **TCOLLATE** command in combination with any two trigger generating commands.

Examples

```
LIMIT(P1,INSIDE,24000,32767,T1,INSIDE,1,32767)
LIMIT(P1,INSIDE,-32768,0,T2,INSIDE, -32768,0)
TCOLLATE(T1, T2, T3)
```

Alternate trigger events from trigger `T1`, generated by a **`LIMIT`** task that detects values 24000 or greater in pipe `P1`, with trigger events from trigger `T2`, generated by another **`LIMIT`** task that detects negative values in pipe `P1`. Place the alternating trigger sequence in trigger `T3`. For this particular example, which requires only simple region tests, the **`TOGGLE`** command is an alternative.

```
CUSTOM(P1,T1)
TGEN(1000,T2)
TCOLLATE(T1,T2,T3)
```

Guarantee that at most one assertion event generated by custom command `CUSTOM` is retained each 1000 samples. Alternate the events generated by `CUSTOM` in trigger `T1` with artificial events generated each 1000 samples by the **`TGEN`** command in trigger `T2`, to produce the alternating sequence in trigger `T3`.

See Also

`TAND`, **`TOR`**, **`TOGGLE`**, **`TOGGWT`**

TFUNCTION1

Define a task that calculates transfer functions from Fourier transform data.

TFUNCTION1 *(<p1>, <p2>, <p3>, <p4>, <scale>,*
<p5>, <p6> [, <limit1>, <limit2>])

Parameters

<p1>
A pipe that contains the real part of the Fourier transform of the input to a system under test.
`WORD PIPE`

<p2>
A pipe that contains the imaginary part of the Fourier transform of the input to a system under test.
`WORD PIPE`

<p3>
A pipe that contains the real part of the Fourier transform of the output of the system under test.
`WORD PIPE`

<p4>
A pipe that contains the imaginary part of the Fourier transform of the output of the system under test.
`WORD PIPE`

<scale>
A scale factor for decibel output.
`WORD CONSTANT`

<p5>
Output pipe for amplitude data.
`WORD PIPE`

<p6>
Output pipe for phase data.
`WORD PIPE`

<limit1>
An optional word constant for suppressing computations with very small inputs.
`WORD CONSTANT`

<limit2>
An optional word constant for suppressing computations with very small outputs.
WORD CONSTANT

Description

TFUNCTION1 calculates transfer functions from Fourier transform data. The transfer function of a system is defined as the ratio of the output of the system to the input of the system, calculated in the frequency domain. The transfer function can be calculated either from Fourier transforms or from crosspower spectrum and autopower spectrum. **TFUNCTION1** uses the transforms directly. **TFUNCTION2** uses power spectrum methods. If averaging is to be performed before the transfer function is calculated, use **TFUNCTION2**.

<p1> and *<p2>* are pipes that contain the real and imaginary components of the Fourier transform of the input to a system under test. *<p3>* and *<p4>* are pipes that contain the real and imaginary components of the Fourier transform of the output of the system under test. **TFUNCTION1** calculates the ratios of corresponding terms in the Fourier transforms, then converts to amplitude and phase, and writes the amplitude in decibels to pipe *<p5>* and the phase to pipe *<p6>*. *<scale>* is a scale factor for the decibel output. See the descriptions of the **DECIBEL** and **POLAR** commands for the scaling of the amplitude and phase outputs.

The transfer function output is meaningful only where the inputs and outputs are not too small. *<limit1>* and *<limit2>* are optional word constants that fix the transfer function result when the inputs are too small. If the amplitude of the complex value in *<p1>* and *<p2>* is less than *<limit1>*, both *<p5>* and *<p6>* are set to zero. If the amplitude of the complex value in *<p3>* and *<p4>* is less than *<limit2>*, *<p6>* is set to zero.

See Also

DECIBEL, **FFT**, **POLAR**, **TFUNCTION2**

TFUNCTION2

Define a task that calculates transfer functions from cross power spectrum data and autopower spectrum data.

TFUNCTION2 *(<p1>, <p2>, <p3>, <scale>, <p5>, <p6>*
[, <limit1>, <limit2>])

Parameters

<p1>
A pipe that contains the real parts of the autopower spectrum of the input to a system under test.
`LONG PIPE`

<p2>
A pipe that contains the real parts of the crosspower spectrum of the output of the system under test.
`LONG PIPE`

<p3>
A pipe that contains the imaginary parts of the crosspower spectrum of the output of the system under test.
`LONG PIPE`

<scale>
A scale factor for decibel output.
`WORD CONSTANT`

<p5>
Output pipe for amplitude data.
`WORD PIPE`

<p6>
Output pipe for phase data.
`WORD PIPE`

<limit1>
An optional word constant for suppressing computations with very small inputs.
`WORD CONSTANT`

<limit2>
An optional word constant for suppressing computations with very small outputs.
`WORD CONSTANT`

Description

TFUNCTION2 calculates transfer functions from crosspower spectrum data and autopower spectrum data. The transfer function of a system is defined as the ratio of the output of the system to the input of the system, calculated in the frequency domain. The transfer function can be calculated either from Fourier transforms or from crosspower spectrum and autopower spectrum. **TFUNCTION1** uses the transforms directly. **TFUNCTION2** uses power spectrum methods. If averaging is to be performed before the transfer function is calculated, use this command.

<p1> is a pipe that contains the real components of the autopower spectrum of the input to a system under test. *<p2>* and *<p3>* are pipes that contain the real and imaginary components of the crosspower spectrum of the output of the system under test. **TFUNCTION2** calculates the ratios of corresponding terms in the Fourier transforms, then converts to amplitude and phase, and writes the amplitude in decibels to pipe *<p5>* and the phase to pipe *<p6>*. *<scale>* is a scale factor for the decibel output. See the descriptions of the **DECIBEL** and **POLAR** commands for the scaling of the phase and amplitude outputs.

The transfer function output is meaningful only where the inputs and outputs are not too small. *<limit1>* and *<limit2>* are optional constants that fix the transfer function result when the inputs are too small. If the amplitude of the value in *<p1>* is less than *<limit1>*, both *<p5>* and *<p6>* are set to zero. If the amplitude of the complex value in *<p2>* and *<p3>* is less than *<limit2>*, *<p6>* is set to zero.

See Also

CROSSPOWER, **DECIBEL**, **POLAR**, **TFUNCTION1**

TGEN

Define a task that generates periodic trigger assertions.

TGEN *(<n>, <trigger>)*

Parameters

<n>

The sample count change for each trigger assertion.
`WORD CONSTANT`

<trigger>

Output stream of artificial trigger events.
`TRIGGER`

Description

TGEN generates periodic trigger assertions. The sample count for each trigger assertion increases by *<n>* sample counts.

TGEN is useful for sending uniformly separated trigger assertions to a **WAIT** task or to a **TOR** task.

Note: **TGEN** generates trigger assertions independent of input sampling activity.

Example

```
TGEN (100, T)
```

Place trigger assertions into trigger `T` corresponding to sample counts 99, 199, 299, 399, … .

See Also

TAND, **TOR**

THERMO

Define a task that converts thermocouple voltages to temperatures.

THERMO *(<in_pipe>, <type>, <scale1>, <scale2>, [<offset>,] <out_pipe> [, <cjc>])*

Parameters

<in_pipe>
Input thermocouple junction voltage data.
`WORD PIPE`

<type>
An integer indicating the thermocouple type.
`WORD CONSTANT`

<scale1>
First term of scaling factor.
`WORD CONSTANT`

<scale2>
Second term of scaling factor.
`WORD CONSTANT`

<offset>
Constant offset adjustment.
`WORD VARIABLE`

<out_pipe>
Output data pipe.
`WORD PIPE`

<cjc>
Optional variable that contains the temperature of the cold junction, in units of tenths of a degree.
`WORD VARIABLE`

Description

THERMO performs linearization of thermocouple voltage data from *<in_pipe>* and places the corresponding temperature data in *<out_pipe>*. *<type>* is an integer from 0 to 9, indicating type E, J, K, R, S, T, N, U, L, or B thermocouple. See the table below for more details on thermocouple types. Input values are multiplied by the factor *<scale1>*/*<scale2>* before linearization. The result of this multiplication is interpreted as a voltage in tens of microvolts. The scale factor should be adjusted according to the gain of the programmable amplifier.

If the *<cjc>* parameter is specified, **THERMO** also performs cold junction compensation. *<cjc>* is a word variable that contains the temperature of the cold junction, in units of tenths of a degree Celsius. An additional table lookup is used to determine a voltage correction to apply to the input data.

The optional *<offset>* parameter is a word variable that is subtracted from each data value read by **THERMO** This parameter allows the easy removal of a DC ground offset from the input data.

The output values are expressed in units of tenths of a degree Celsius. The temperature can be converted to Fahrenheit using the following DAPL expression:

```
PF = PC*9/5 + 320
```

The Applications Manual provides several examples of thermocouple processing using the **THERMO** command.

The following table specifies the temperature ranges of the linearization data for each thermocouple type supported by the **THERMO** command. The Error column describes the contribution of the thermocouple linearization to the total measurement error.

Thermocouple Type	Code	Range (Deg C)	Error* (± Deg C)
ANSI type E (IEC 584)	0	-270 to 1000	0.4
ANSI type J (IEC 584)	1	-200 to 1200	0.3
ANSI type K (IEC 584)	2	-270 to 1370	0.4
ANSI type R (IEC 584)	3	-50 to 1760	0.4
ANSI type S (IEC 584)	4	-50 to 1750	0.3
ANSI type T (IEC 584)	5	-270 to 400	0.3
ANSI proposed type N	6	-260 to 1300	0.3
Type U (DIN 43710)	7	-200 to 600	0.2
Type L (DIN 43710)	8	-200 to 900	0.3
Type B (IEC 584)	9	0 to 1750	0.6

* Maximum contribution to overall error by **THERMO**.

The conversions provided by the **THERMO** command span the full standard temperature range for the thermocouple devices. There is a substantial variation among individual devices, and the standard correction curves are typically in error of 1 to 2 degrees Celsius for any individual thermocouple device. Much higher conversion accuracy is possible by calibrating an individual device and using the **INTERP** command to perform the linearization.

Example

```
THERMO (P1, 2, 10000, 32767, P2)
```

Read input data from pipe `P1`, multiply the data by 10000/32767, perform type `K` thermocouple linearization, and place the results in pipe `P2`.

See Also

INTERP

TIME

Set the time intervals at which successive inputs are sampled or successive outputs are updated.

TIME *<interval>*

Parameters

<interval>
The time in microseconds.

Description

TIME sets the time intervals at which successive inputs are sampled or outputs are updated. The time is specified in microsecond units. For Data Acquisition Processor models with multiplexed input sampling or output updating, and with *M* channels in the input or output channel pipe, each channel is sampled or updated every *<interval>*M* microseconds. For Data Acquisition Processor models with simultaneous input sampling, and with *M* channel groups defined in the input or output channel pipe, each channel group is sampled every *<interval>*M* microseconds.

The minimum *<interval>* for the **TIME** command depends on the Data Acquisition Processor model. The following table summarizes the minimum sampling and update times for most Data Acquisition Processors. All times are in microseconds.

	Minimum Analog Input Time	Minimum Digital Input Time	Minimum Analog Output Time	Minimum Digital Output Time	Increment	Maximum Time
840/103	1.25	1.25	2.50	1.25	0.05	52428
3000a/212	1.30	0.60	1.20	0.60	0.10	13107
3200a/415	1.30	0.60	1.20	0.60	0.10	104856
3216a/415	5.00	0.60	2.00	0.60	0.10	104856
4000a/112	1.25	1.25	2.50	1.25	0.05	52428
4000a/212	1.25	1.25	2.50	1.25	0.05	52428
4200a/526	1.30	0.60	1.20	0.60	0.10	104856
4400a/446	1.25	NA	NA	NA	0.05	52428
5200a/526	1.25	0.06	1.20	0.60	0.05	52428
5200a/626	1.25	0.06	1.20	0.60	0.05	52428
5216a/626	3.00	0.06	2.50	0.60	0.05	52428
5400a/627	0.80 (*8*)*	NA	NA	NA	0.02	83884
5400a/627	0.50 (*4*)*	NA	NA	NA	0.02	83884

* Minimum time interval depends on the number of *channels* in a channel group

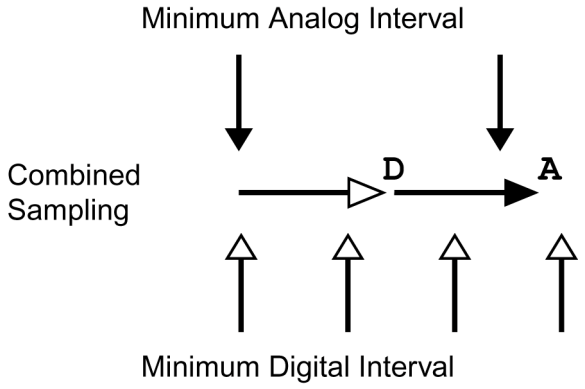
The information in this table should be used with caution. Even though the conversion hardware and the processing can sustain sampling at these rates, full accuracy of the conversion is not always guaranteed. Some of the conditions that affect the maximum sampling and updating rates:

- *Quality of signal cables.* Cable type, length and termination are critical to the success of high speed sampling and updating. Cabling should be tested to verify proper operation at high speeds.
- *Slew rate limiting.* If an input or output amplifier switches between widely differing input or output voltage levels, the conversion might not settle to full accuracy during a sampling interval. See the Data Acquisition Processor hardware manual for information about slew rate limitations in various configurations.
- *High gain.* The bandwidth of a feedback amplifier reduces at high gains. When a **SET** command assigns a gain higher than 1.0, the Data Acquisition Processor programmable gain amplifier requires extra time to settle to full accuracy. See the Data Acquisition Processor hardware documentation for more information.

Data Acquisition Processor models place different restrictions on fractional microsecond timings. DAPL allows three decimal places of precision in the time specification. Time specifications that are multiples of 0.1 microseconds are compatible with all current models. The “increment” column of the table shows the time increments that are allowed for each Data Acquisition Processor model. `<interval>` must be a multiple of the time increment. If `<interval>` cannot be realized exactly, a warning is displayed and the actual sampling time is rounded down to the nearest valid multiple.

For maximum compatibility among Data Acquisition Processor models, the *<interval>* should not exceed 10000 μs, but most models allow 50000 μs or more. Even though **TIME** restricts the maximum sampling time, the effective sampling interval can be made longer by using **SKIP** or **AVERAGE** commands. The effective output update time can be made longer by using the **REPLICATE** command.

Some models of Data Acquisition Processors allow digital sampling at shorter intervals than analog sampling. When there is a mix of analog and digital input signals, these Data Acquisition Processor models allow *<interval>* to be set to a value shorter than the minimum allowed for analog signals, provided that the combined time intervals for each analog channel and its preceding digital channels is at least as long as the minimum analog interval.



The combined digital and analog intervals cannot be less than the minimum analog interval.

A configuration of this kind that mixes digital and analog signals, and has a sampling time interval shorter than the minimum allowed for analog signals alone, is called "fast sampling." To ensure that there are enough digital samples in the sequence when starting each pass through the channel list, a fast configuration must always begin with an appropriate number of digital samples.

Note: Fast sampling is not effective when using a Counter/Timer Board. The Counter/Timer Board needs a **TIME** interval no less than 3.0 μs.

The following is an example of fast input sampling for a Data Acquisition Processor with a minimum analog input **TIME** of 1.3 μs, a minimum digital sampling time of 0.6 μs, and a time interval resolution of 0.1 μs. The configuration has one digital channel and one analog channel. If the **TIME** statement specifies *<interval>* to be 0.6 μs, sampling in the following manner is *incorrect*:

- the digital channel is sampled at 0.6 μs (okay)
- the analog channel is sampled at 1.2 μs (error, less than minimum)
- this cycle repeats

One solution to this problem is to sample the digital channel an extra time.

- the digital channel is sampled at 0.6 μs (discarded)
- the digital channel is sampled at 1.2 μs (okay)
- the analog channel is sampled at 1.8 μs (okay)
- this cycle repeats

However, the only thing gained by discarding a sample is a time delay. It is also possible to achieve a time delay by adjusting the time interval. Selecting the next allowed sampling interval of 0.7 μs:

- the digital channel is sampled at 0.7 μs (okay)
- the analog channel is sampled at 1.4 μs (okay)
- this cycle repeats

Because this configuration has a net faster sampling rate on the analog signal, it is probably the preferred configuration. The DAPL commands for this example would look like the following:

```
IDEF A
  CHANNELS  2
  SET IP0 B
  SET IP1 S0
  TIME 0.7
  END
```

Fast input sampling configurations are possible because input hardware devices allow sampling and conversions to be started while digital sampling is done in parallel. Output updating does not have a similar feature, so output conversions must start and end within the *<interval>* specified by the **TIME** command. The output configuration accepts the minimum digital output time for both analog and digital updates and does not check whether analog update intervals are long enough. If they are not, output voltages can be latched before they completely settle to the correct value. In some applications, when all multiplexed output signals are close to the same level and changes from one update to the next are small, less settling time is required for output voltage transitions to settle. These applications might be able to track the desired output signal with sufficient accuracy at higher rates. Test carefully to determine actual accuracies achieved.

Examples

```
TIME 5000
```

Set sampling speed to 5 milliseconds per sample.

```
TIME 2.5
```

Set update speed to 2.5 microseconds per update.

TOGGLE

Define a task that tests for sequences of ON events alternating with OFF events.

TOGGLE *(<on_pipe>*, *<on_region> [, <off_pipe>]*,
<off_region>, *<trig>)*

Parameters

<on_pipe>
Input data pipe for ON events.
WORD PIPE

<on_region>
Testing region for ON events.
REGION

<off_pipe>
Input data pipe for OFF events.
WORD PIPE

<off_region>
Testing region for OFF events.
REGION

<trig>
Output trigger for alternating trigger assertions.
TRIGGER

Description

TOGGLE tests for sequences of ON events alternating with OFF events, placing alternating trigger assertions into *<trig>* . For detecting ON events, data from the *<on_pipe>* are tested using the *<on_region>*, in a manner similar to the **LIMIT** command. The first event is always an ON event. Once an ON event is detected, data from the *<off_pipe>* are tested using the *<off_region>*, again similar to the **LIMIT** command. If the same data stream is to be tested both for the ON and the OFF conditions, the *<off_pipe>* parameter is omitted. When both *<on_pipe>* and *<off_pipe>* are specified, data from only one of the streams is tested at any time; while testing for ON conditions, data from the *<off_pipe>* are skipped, and while testing for OFF conditions, data from the *<on_pipe>* are skipped. These conditions enforce a strict synchronization and alternation between ON and OFF events.

The strict alternation cannot be enforced when trigger modes are used that suppress trigger events or artificially introduce events. For this reason, the NATIVE operating mode is recommended for the trigger. **TOGGLE** will not run if the trigger has a nonzero HOLDOFF property or operates in the AUTO mode.

The **TOGGLE** command usually is used in conjunction with the **TOGGWT** command, which extracts variable-length blocks of data from a data stream in response to ON/OFF events.

Example

```
TOGGLE(P1, INSIDE, 1000, 10000, OUTSIDE, 0, 32767, TT)
```

Test data from pipe P1 for an "on-event" value inside the region 1000 to 10000, and when one is detected, assert an event in trigger TT. Then, test the same source pipe P1 for an "off-event" negative value, and when one is detected, assert another event in trigger TT. Repeat the cycle.

```
TOGGLE(P1, INSIDE, 1000, 10000, P2, OUTSIDE, 0, 32767, TT)
```

Test samples from pipe P1 for "on-event" values inside the region 1000 to 10000, and test samples from a separate pipe P2 for "off-event" negative values.

See Also

LIMIT, **TOGGWT**

TOGGWT

Collect data between alternating ON and OFF trigger events.

TOGGWT (*<source>, <toggle>, <dest> [, <size>]*
[, <format> [, <tags>]])

<format> = *STREAM | <block specifier>*

<block specifier> = *BLOCKS [SPANNED | SINGLE] [STAMPED]*

Parameters

<source>
Input data pipe.
WORD PIPE

<toggle>
The trigger that that signals "ON" and "OFF" events.
TRIGGER

<dest>
Output data pipe.
WORD PIPE

<size>
A value that specifies the maximum data block size.
WORD CONSTANT

<tags>
Specifies an optional separate pipe for identification information.
LONG PIPE

Description

TOGGWT acts somewhat like the **WAIT** command. It interprets a stream of events from trigger *<toggle>* as alternating ON and OFF events. The first event is always an ON event. When an ON event is received, the **TOGGWT** command begins to accept data from the *<source>* pipe, copying data to the *<dest>* pipe. When an OFF event is received at trigger *<toggle>*, the **TOGGWT** command stops accepting data from the *<source>* pipe. Data that do not occur between ON and OFF events are discarded.

Data formatting options are specified by optional parameters. The *<size>* parameter specifies a maximum block size. If no *<size>* parameter is specified, the

DAPL system supplies a default. The *<format>* parameter selects formatting options according to keywords in a format specifier string. The *<tags>* parameter specifies an optional separate pipe for identification information.

The syntax of the format option string is:

```
" <format string> "

format string  =   STREAM | <block specifier>

block specifier  =
BLOCKS  [ SPANNED | SINGLE ]  [ STAMPED ]
```

Some examples of valid format specifiers:

```
"FORMAT = BLOCKS"
"FORMAT = STREAM"
"FORMAT = BLOCKS SINGLE STAMPED"
"FORMAT = BLOCKS SPANNED"
```

When a format string is specified, one of the following two options must be selected:

- `STREAM`. The default. The selected data is placed into the *<dest>* pipe without any identifier marks. The optional *<tags>* parameter is not allowed with this option. This format is probably the preferred one for isolated or single events. The data are sent in an arbitrary number of blocks of maximum size *<size>* until all data from the `ON` event to the `OFF` event are transferred.
- `BLOCKS`. Data are sent in blocks and length tags are placed into the identification information. If the *<tags>* parameter is specified, the tag information is placed into that separate pipe. Otherwise, the tag information is merged with the data stream and precedes each data block.

When the `BLOCKS` format is specified, there are two further choices.

- `SINGLE`. Specifies that the data are to be sent in a single block of up to length *<size>*. The actual size is indicated in the tag information. No data are transmitted until the block is full or an `OFF` event terminates the block at a smaller size. If there are more than *<size>* samples between the `ON` and the `OFF` event, any samples that will not fit in the block are ignored. It is recommended that *<size>* be specified, rather than using the default.
- `SPANNED`. The default when `BLOCKS` is specified. The data are sent in blocks no larger than *<size>*. Each block is tagged, in addition to a length tag, with a `CONTINUED/COMPLETED` tag. If marked `CONTINUED` , the block covers only part of the remaining data, and another block containing a non-zero number of

additional values is expected. If marked COMPLETED , the block contains all of the remaining samples up to the OFF event. For example, if data are sent as 2 blocks, the first block will be marked CONTINUED and the next will be marked COMPLETED. The numeric representation of the CONTINUED/COMPLETED tags is given below.

Both of the BLOCKS options can have an additional STAMPED option. When this is selected, a 32-bit timestamp corresponding to the sample timestamp of the ON event is placed after the other tag information.

Note: The timestamps can also be obtained by using the **TSTAMP** command and then the **SKIP** command to select only ON events.

The format of the tag data is as follows:

```
length          16 bits
continuation    16 bits     (COMPLETED = 0, CONTINUED = 1)
timestamp       32 bits     (sent with STAMPED option only)
```

The continuation field is only meaningful for SPANNED blocks. With SINGLE blocks, the continuation field is always zero. Note that when tag information is merged with long data or placed into a separate *<tags>* pipe, the 16-bit length and continuation fields are merged into a single 32-bit value, with the length in the low-order 16 bits and the continuation in the high-order 16 bits. Also note that when tag information is merged with 16-bit data, the 32-bit timestamp will appear as a sequence of two words, with the low-order 16 bits first, followed by the high-order 16 bits. The timestamp field is not sent unless the STAMPED option is selected.

Examples

```
TOGGWT(P1, TT, P2)
```

Take samples from pipe P1, according to events in trigger TT, starting at an ON event, and stopping at an OFF event. Place the data in the P2 pipe. Use the default formatting option, STREAM, which generates no identification information.

```
TOGGWT(P1, TT, P2, 2000, “FORMAT=BLOCKS SINGLE”,TAGS)
PCOUNT(TAGS,XXX)
```

Copy single blocks of up to 2000 items from pipe P1 to pipe P2, beginning at an ON event from trigger TT, and ending when the block is full or when an OFF event arrives from TT. Discard the tag information, by placing it into a separate TAGS pipe and using another command to empty the pipe.

```
TOGGWT(P1, TT, $BINOUT, 512, “FORMAT=BLOCKS SPANNED”)
```

Copy data from pipe P1 to communication pipe $BINOUT beginning at an ON event from trigger TT and continuing until an OFF event. The spanned format is used, sending the data in blocks no larger than 512 items, and merging tag information ahead of each data block in the $BINOUT data stream.

See Also

TSTAMP, **SKIP**, **WAIT**, **TOGGLE**

TOR

Define a task that calculates a logical ‘or’ of trigger assertions.

TOR *(<in_trigger_1>, ... , <in_trigger_n>, <out_trigger>)*

Parameters

<in_trigger_1>
An event source.
TRIGGER

<in_trigger_n>
Additional event sources.
TRIGGER

<out_trigger>
Combined new event stream.
TRIGGER

Description

TOR calculates a logical ‘or’ of trigger assertions. *<out_trigger>* is asserted each time at least one of *<in_trigger_1>*, … , *<in_trigger_n>* is asserted.

TOR typically is used when a trigger event can occur for more than one reason or can be detected on a number of separate data channels.

Example

```
TOR (T1, T2, T_OUT)
```

Assert T_OUT each time T1 or T2 is asserted.

See Also

TAND

TRIANGLE

Define a task that generates triangle wave data.

TRIANGLE *(<amplitude>, <period>, <out_pipe>*
[,<mod_type>, <mod1> [, <mod2>]])

Parameters

<amplitude>
A value that is one half the peak to peak distance of the output.
`WORD CONSTANT | WORD VARIABLE`

<period>
The number of sample values in each wave.
`WORD CONSTANT | WORD VARIABLE`

<type>
A value that specifies the type of wave function.
`WORD CONSTANT`

<out_pipe>
Output pipe for triangle wave data.
`WORD PIPE`

<mod_type>
A value that selects amplitude and/or frequency modulation of the output wave.
`WORD CONSTANT`

<mod1>
Pipe for first modulation signal.
`WORD PIPE`

<mod2>
Pipe for second modulation signal.
`WORD PIPE`

Description

TRIANGLE generates triangle wave data and places the data in *<out_pipe>*. *<period>* is the number of sample values in each wave. The *<amplitude>* is one half the peak to peak distance of the output wave. The maximum value of *<amplitude>* is 32767.

Note: **TRIANGLE** is identical to **WAVEFORM**, with *<type>* set equal to 0.

Three optional modulation parameters may be specified. *<mod_type>* selects amplitude and/or frequency modulation of the output wave. The value of *<mod_type>* must be one of the following:

1. amplitude modulation controlled by the data in *<mod1>*
2. frequency modulation controlled by the data in *<mod1>*
3. amplitude and frequency modulation controlled by the data in *<mod1>* and *<mod2>*, respectively

<mod1> and *<mod2>* are pipes. One value is read from the pipe(s) for each value output by **TRIANGLE**. Modulation values are interpreted as signed binary fractions; they are multiplied by the base amplitude or frequency to obtain the amplitude or frequency.

An alternative method for changing the amplitude or frequency of **TRIANGLE** during execution uses a DAPL variable as the *<amplitude>* or *<period>* parameter of **TRIANGLE**. The value of this variable can be changed during execution using a **LET** command. This is efficient, but cannot adjust the amplitude or frequency continuously, and changes are detected and applied asynchronously.

Example

```
TRIANGLE (1000, 100, P2)
```

Generate a triangle wave with values ranging from -1000 to 1000 and a period of 100 samples.

See Also

COSINEWAVE, **SAWTOOTH**, **SINEWAVE**, **SQAREWAVE**, **WAVEFORM**

TRIGARM

Define a task that allows a task or PC application to asynchronously arm or disarm a software trigger.

TRIGARM *(<pipe>, <trig>)*

Parameters

<pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<trig>
Trigger of interest.
`TRIGGER`

Description

The **TRIGARM** command allows a task or PC application to asynchronously arm or disarm software trigger *<trig>* by setting the trigger property `GATE=ARMED` or `GATE=DISARMED`, respectively. All trigger operating modes except the default (`NATIVE`) mode are affected. A zero value received from *<pipe>* disarms the trigger; a non-zero value arms the trigger. While disarmed, the trigger will not respond to new assertions and will not generate artificial events. See the **TRIGGERS** command for information about operating modes and trigger properties.

A typical application of the **TRIGARM** command is one-shot data collection. The trigger is defined with the `MANUAL` operating mode and an initial `GATE=DISARMED` property. When the DAPL processing procedure is started, trigger events are not recognized because the trigger is disarmed. Later, when the application places a nonzero value into *<pipe>*, the **TRIGARM** command changes the `GATE` property of *<trig>* to `ARMED` . Operating in `MANUAL` mode, trigger *<trig>*, responds to the next asserted event, and then resets the `GATE=DISARMED` property.

The system command **EDIT** is an alternate means of arming and disarming a trigger.

Example

```
TRIGARM(P1,T1)
```

Set the GATE property of trigger T1 to DISARMED when a value of 0 is received from pipe P1. Set the GATE property of trigger T1 to ARMED when a non-zero value is received from pipe P1.

See Also

TRIGGERS, **HTRIGGER**, **EDIT**, **WAIT**

TRIGGERS

Define one or more software triggers.

TRIGGERS *<t_def> [, <t_def>]**

TRIGGER *<t_def> [, <t_def>]**

TRIG *<t_def> [, <t_def>]**

T *<t_def> [, <t_def>]**

<t_def> = *<name> [MODE=<mode> [<property>]*]*

<property> =
HOLDOFF=<hold> | STARTUP=<start> | CYCLE=<cycle> |
GATE=<arm>

Parameters

<name>
Trigger symbol name.

<mode>
A keyword selecting the trigger operating mode. Must be one of: `NATIVE`, `NORMAL`, `MANUAL`, `AUTO`, `DEFERRED`.

<arm>
A keyword for enabling or disabling trigger activity. Must be either `ARMED` or `DISARMED`.

<cycle>
Automatic triggering cycle for `AUTO` mode.
`LONG CONSTANT`

<hold>
Holdoff interval for all modes except `NATIVE`.
`LONG CONSTANT`

<start>
Initial startup interval for all modes except `NATIVE`.
`LONG CONSTANT`

Description

TRIGGERS defines one or more software triggers. A *<name>* parameter constructs a software trigger and assigns it a symbol name. The *<name>* symbol can then be used as a parameter for processing tasks. Note that all other parameters are optional, and most applications will not need them. A trigger definition with none of the optional parameters operates in the NATIVE mode, as defined below.

The trigger begins continuous operation, with dynamic allocation and release of memory, when all reader and writer tasks for the trigger are started.

Optional operating modes modify the way that trigger events are asserted. The operating modes act as filters, accepting some requests to assert the trigger, suppressing other requests.

- NATIVE. The NATIVE mode applies no filtering actions, and the trigger does not use any of the trigger properties. This mode is optimized for maximum speed when other triggering features are not needed.
- NORMAL. The NORMAL mode uses the HOLDOFF, STARTUP, and GATE properties. This mode simulates the normal mode operation of an oscilloscope, in which a display sweep must be completed before responding to another trigger event.
- DEFERRED. This mode is the same as NORMAL mode, except that events occurring inside the HOLDOFF interval are delayed until just after the HOLDOFF interval.
- AUTO. This mode is similar to NORMAL mode, except that artificial events are inserted as specified by the *<cycle>* property when no events occur otherwise. This simulates the AUTO triggering mode of an oscilloscope. The CYCLE property value *<cycle>* cannot be smaller than the HOLDOFF property value *<hold>*.
- MANUAL. This mode is for one-shot events. The HOLDOFF, STARTUP, and GATE properties are recognized. The trigger responds to only one event, and then it sets its GATE property to DISARMED. It can be rearmed using an **EDIT** or **TRIGARM** command.

The operating modes use the following trigger properties. These properties are not available for the NATIVE mode.

- GATE. This property can optionally be assigned the value ARMED or DISARMED. The trigger does not accept new assertions when disarmed. The setting can be changed later using an **EDIT** or **TRIGARM** command. Default is ARMED.
- HOLDOFF. This property specifies a number of samples, *<hold>*. The trigger will not allow a new assertion event for *<hold>* samples after a natural or artificial event is asserted. Default is zero.
- STARTUP. This property specifies an interval similar to HOLDOFF, except that events are ignored if they occur during the first *<start>* number of samples. Default is zero.

- CYCLE. This property is used by the AUTO mode to determine the number of samples between artificially-generated events.

Note: A trigger stores 32-bit sample counts corresponding to assertions. If an application generates more than 2^32 data values (4294967296 data values), trigger assertion counts wrap around to zero.

Examples

```
TRIGGERS T1
```

Define one trigger T1 using the default NATIVE mode.

```
TRIGGERS TA MODE=AUTO CYCLE=2000 HOLDOFF=100
```

Trigger TA is defined to operate in the AUTO mode, and artificial events are inserted every 2000 samples when there are no other events. There is a holdoff interval of 100 samples after each actual or artificial event during which no additional assertions are accepted.

```
TRIGGERS TM MODE=MANUAL STARTUP=5000 GATE=ARMED
```

Define trigger TM to operate in the MANUAL mode; no event is recognized until after the startup interval of 5000 samples. After that interval, the trigger will respond to the next event, also changing its GATE property from ARMED to DISARMED .

```
TRIGGERS TN MODE=NORMAL STARTUP=5000 HOLDOFF=100 \
  GATE=DISARMED
```

Trigger TN is designed to operate in the NORMAL mode, but starting initially with the trigger DISARMED. The trigger will ignore all events until the GATE property is changed to ARMED and the startup interval of 5000 samples is completed. After both of these conditions are satisfied, the trigger continues operation in NORMAL mode.

See Also

LIMIT, **WAIT**, **TRIGARM**, **HTRIGGER**

TRIGRECV

Define a task that recovers transferred triggering information.

TRIGRECV *(<in_pipe>, <out_trigger>)*

Parameters

<in_pipe>
Input data pipe for encoded triggering information.
`LONG PIPE`

<out_trigger>
Output trigger.
`TRIGGER`

Description

TRIGRECV recovers encoded software triggering information received from another task through a user-defined or communication data pipe *<in_pipe>*. The triggering information is placed into trigger *<out_trigger>*.

The data pipe must contain a long data type. An examples of a compatible user-defined data pipe is the following:

```
PIPE   PXTRIG    LONG
```

See Chapter 14 for information on how to set up the communication pipes.

A typical application for **TRIGRECV** is high speed data acquisition on a slaved Data Acquisition Processor board, where high-speed trigger detection processing is performed by a separate Data Acquisition Processor, with triggering information transmitted using the **TRIGSEND** command.

Example

```
TRIGRECV(XF3, T3)
```

Encoded triggering information is received through `LONG` communication pipe `XF3` and reconstructed in trigger `T3`.

See Also

PIPES, **TRIGSEND**

TRIGSCALE

Define a task that modifies a stream of trigger events.

TRIGSCALE *(<trig_in>, <offset>, <mul>, , <trig_out>)*

Parameters

<trig_in>
Input trigger.
TRIGGER

<offset>
A value that specifies the offset adjustment.
WORD CONSTANT

<mul>
A value that specifies the channel multiplication scaling.
WORD CONSTANT

* *
A value that specifies the data reduction scaling.
WORD CONSTANT

<trig_out>
Output trigger.
TRIGGER

Description

The **TRIGSCALE** command modifies a stream of trigger events received from trigger *<trig_in>* and places the results in trigger *<trig_out>*. The modifications to the trigger values adjust for data rates, channel groupings, and time (phase) offsets.

The **TRIGSCALE** command applies three operations to the values taken from the *<trig_in>* trigger: an offset adjustment specified by the *<offset>* parameter, a data reduction scaling specified by the * * parameter, and a channel multiplication scaling specified by the *<mul>* parameter.

```
timestamp = ((old_timestamp + <offset>) / <div>) * <mul>
```

The *<offset>* operation corresponds to a time-shift, in terms of a number of samples in the input sequence. A positive value indicates a delay, and a negative

value indicates an advance. A negative value has somewhat the same effect as pre-triggering samples when using the **WAIT** command. The first sample, when the Data Acquisition Processor starts, is always sample number zero, so any event advanced ahead of sample zero is removed. If there is no time shift, set *<offset>* to zero.

The * * operation accounts for data rate reduction due to processing. For example, suppose that an event is detected in a stream of raw data samples, and this data is also averaged in blocks of 100 samples, producing one averaged value for every 100 raw input values. To locate the average value that corresponds to the block containing the detected event, specify a value of 100 for the * * parameter. The value of * * must always be positive, so if there is no data reduction, specify the value one.

The *<mul>* operation accounts for data blocks or multiple channel groups. For example, suppose that the **BAVERAGE** command is used to smooth the voltage readings from a group of 6 thermocouple devices. One of the 6 channels is then tested for a triggering condition (such as temperature limits) using a **SKIP** command and a **LIMIT** command. To locate the 6-channel block of data corresponding to an event, specify 6 for the value of the *<mul>* parameter. Note that the **WAIT** command provides this channel multiplier factor implicitly when used with an input channel pipe list, but in this example, the data comes from a user-defined pipe after averaging, and not from an input channel list. The value of *<mul>* must always be positive, so if there is no data reduction, specify the value one.

Notice that the * * operation is applied first, with remainder ignored, and then the *<mul>* operation is applied. The most important effect of this ordering is that the resulting trigger timestamps always occur at the first sample in a group.

In practice, most applications will need only one of the three adjustments: a shift, reduction, or group scaling.

Example

```
TRIGSCALE(T1, 0, 100, 100, T2)
```

Divide each event timestamp appearing in trigger `T1` by 100, then multiply it by 100, placing the result in trigger `T2`, with no time shift adjustment. The effect is to move any event occurring in a block of 100 samples to the beginning of the block.

```
TRIGSCALE(T1, -100, 8, 1, T2)
```

Convert each trigger assertion appearing in trigger `T1` to a trigger assertion in trigger `T2`, such that the new timestamp corresponds to a location where 100 pre-trigger samples are available for each of 8 multiplexed channels in a data stream. Trigger events prior to timestamp 100 are ignored.

See Also

LIMIT, **WAIT**

TRIGSEND

Define a task that transfers trigger information to another Data Acquisition Processor.

TRIGSEND *(<in_trigger>, [<notify> ,] <pipe> [, <pipe>])*

Parameters

<in_trigger>
Input trigger.
`TRIGGER`

<notify>
A value that specifies the number of samples to process before sending status information.
`WORD CONSTANT | LONG CONSTANT`

<pipe>
Output pipe(s) for encoded information.
`LONG PIPE | LONG COMMUNICATIONS PIPE`

Description

TRIGSEND encodes trigger information in a data pipe, allowing this information to be transferred to another Data Acquisition Processor for coordinated software trigger processing. **TRIGSEND** extracts event and status information from *<in_trigger>* and copies this information to the specified list of *<pipe>* destinations.

The optional *<notify>* parameter requests sending status information after processing each *<notify>* number of samples from the data pipe associated with *<in_trigger>*. This parameter should normally be omitted, letting the DAPL system supply a default. This parameter is useful in certain situations where sampling rates are slow. In such situations, a reasonable number to specify would be the number of samples processed for triggering (with or without events) in a few milliseconds. If this parameter is too small, **TRIGSEND** can cause a high level of data bus traffic, which interferes with other PC communication.

The destination pipes must accept a long data type. An example of a compatible user-defined pipe is the following:

```
PIPE   PXTRIG    LONG
```

See Chapter 14 for information on how to set up the communication pipes.

A typical application for **TRIGSEND** is triggered data acquisition on multiple, slaved Data Acquisition Processor boards. A Data Acquisition Processor configured as a master can issue trigger events to a number of slave processors, allowing software-controlled data capture at very high rates on many channels. Each slave board receives and reconstructs the triggering information using a **TRIGRECV** task.

Example

```
TRIGSEND(T1, XF2, XF3)
```

Encode the information from trigger `T1` and transfer it through `LONG` communication pipes `XF2` and `XF3`.

See Also

PIPES, **TRIGRECV**

TSTAMP

Define a task that is used to time stamp trigger events.

TSTAMP *(<trigger>, <out_pipe>)*

Parameters

<trigger>
Stream of trigger assertions.
`TRIGGER`

<out_pipe>
Output data pipe for the sample numbers of the trigger events.
`WORD PIPE | LONG PIPE`

Description

TSTAMP is used to time stamp trigger events. **TSTAMP** waits for *<trigger>* assertions. Each time *<trigger>* is asserted, **TSTAMP** puts the sample number of the trigger event into *<out_pipe>*.

Multiplying the sample number generated by **TSTAMP** by the time interval between successive samples converts the sample number to elapsed time.

Example

```
TSTAMP (T1, PL1)
```

Each time trigger `T1` is asserted, place the sample number of the event causing the assertion into pipe `PL1`.

See Also

TRIGGERS

UPDATE

Specify burst mode options for an input or output configuration.

UPDATE *<option>*

Parameters

<option>
A keyword, either BURST or CONTINUOUS.

Description

The **UPDATE** command selects BURST or CONTINUOUS mode operation in an input configuration or output configuration. The default is CONTINUOUS.

CONTINUOUS mode is the normal operating mode of an input configuration or output configuration. Once sampling or updating begins, it continues until the configuration is stopped.

For input configuration BURST mode, a **COUNT** command and an **HTRIGGER** ONESHOT command must be specified. Input sampling begins when an external trigger is asserted. Sampling stops when **COUNT** values have been sampled. Input sampling begins again when the external trigger is asserted. **COUNT** must be an integral multiple of the number of channels in the input configuration. Note that when external clocking is used with input configuration BURST mode, the last sampled value is held in the pipeline until the next external clock.

In output configuration BURST mode, output updating stops when output channel pipe data are exhausted, but no output underflow warning occurs. Output updating resumes once the available data again exceed the threshold specified by the **OUTPUTWAIT** command. Output configuration BURST mode is not available if an output configuration **CYCLE** is specified.

The command **SLAVE** is not supported by **UPDATE** BURST mode.

Example

```
UPDATE BURST
```

Specify that output updating operate in burst mode.

See Also

CYCLE, **COUNT**, **HTRIGGER**

VARIABLES

Define named variables.

VARIABLES *<name> <type> [= <value>]*
*[, <name> <type> [= <value>]]**

VARIABLE *<name> <type> [= <value>]*
*[, <name> <type> [= <value>]]**

VAR *<name> <type> [= <value>]*
*[, <name> <type> [= <value>]]**

V *<name> <type> [= <value>]*
*[, <name> <type> [= <value>]]**

Parameters

<name>
Text of assigned variable name.
Variable Name

<type>
Keyword for data type of new variable symbol.
`WORD | LONG | FLOAT | DOUBLE`

<value>
An optional initial value for the variable.

`WORD CONSTANT`	`WORD VARIABLE`	
`LONG CONSTANT`	`LONG VARIABLE`	
`FLOAT CONSTANT`	`FLOAT VARIABLE`	
`DOUBLE CONSTANT`	`DOUBLE VARIABLE`	

Description

The **VARIABLES** command defines named, adjustable number values. Variables can be used by tasks to share information that changes asynchronously. The *<type>* keyword specifies the data type of the new variable.

An initial value can be specified when defining a variable. Floating point variables and constants cannot be used to assign a value to a `WORD` or `LONG` variable, but otherwise, any constant or variable is acceptable if it provides a value in the representable range. If the equal sign operator and initializer term *<value>* are

omitted, the variable is created with an initial value of zero. The value of the variable is not changed by a **STOP** command, so use the **LET** command if necessary to restore the initial value after running a DAPL configuration.

Note: For compatibility with earlier versions of the DAPL system, two older command forms are also accepted. The **VARIABLES** command will accept a variable declaration that does not specify a data type. For this special case, the type defaults to `WORD.` The **VARIABLES** command will also accept a declaration that specifies a `WORD` or `LONG` data type keyword after the initializer term. These old command forms are not compatible with floating point data types, and hexadecimal expressions could be interpreted in a manner inconsistent with other DAPL commands. Use of the old command notations should be avoided.

Examples

```
VARIABLES V1 WORD,F2 FLOAT
```

Define two variables, each with an initial value of zero.

```
VAR GAMMA LONG = GAMMA17
```

Define a 32-bit variable `GAMMA` with an initial value taken from symbol GAMMA17.

See Also

CONSTANTS, **LET**, **SDISPLAY**

VARIANCE

Define a task that computes variance statistics for blocks of data values.

VARIANCE (*<in_pipe>*, *<count>*, *<out_pipe>*)

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE`

<count>
The number of data values per block.
`WORD CONSTANT`

<out_pipe>
Output pipe for variance data.
`WORD PIPE | LONG PIPE`

Description

VARIANCE computes the variance of blocks of *<count>* data values. The variance is sent to *<out_pipe>*.

Note: The variance is defined as the sum of the squares of the deviations from the block average, divided by *<count>*. For some statistical applications, division by *<count>*-1 is appropriate. The variance can be adjusted by a DAPL expression, as long as *<count>* is guaranteed to be greater than one.

Example

```
VARIANCE (P1, 10000, P2)
```

Compute the variance of blocks of 10000 observations from pipe `P1` and place the variance in pipe `P2.`

See Also

AVERAGE, **RMS**

VECTOR

Define a vector.

VECTOR *<name> <type> = (vn [, <vn>]*)*

VECT *<name> <type> = (vn [, <vn>]*)*

VEC *<name> <type> = (vn [, <vn>]*)*

Parameters

<name>

Text of assigned vector name.

NAME

<type>

A keyword specifying a data type for the vector data.

WORD | LONG | FLOAT | DOUBLE

<vn>

A numeric value to initialize a term of the vector.

WORD CONSTANT | LONG CONSTANT |
FLOAT CONSTANT | DOUBLE CONSTANT

Description

VECTOR defines a vector of numbers in shared DAPL system storage. The keyword *<type>* specifies the numeric type of the data elements in the vector. The expressions *<vn>* in the initializer list specify constant numerical values representable by the specified data type. Each vector term must be initialized, and the number of initializer expressions in the list determines the vector length, which cannot exceed 16384 terms.

The initializer list can be coded on multiple command lines. Placing a list separator comma at the end of a line tells the DAPL system to expect more initializer terms on the next line. It is acceptable but not necessary to place a backslash "\" continuation character at the end of lines continued in this manner.

Note: For compatibility with older versions of the DAPL 2000 system, the **VECTOR** command might also accept alternate notations that omit the *<type>* parameter or

defer it to the end. The obsolete notations are incompatible with floating point data types. New applications should avoid these old notations.

Examples

```
VECTOR A LONG = (1,2,-3,-4)
```

Define a four element vector of 32-bit LONG integer values.

```
VECTOR B WORD = (5,6,7,8,
           9,10,11,
           12,13)
```

Define a vector of nine 16-bit WORD values using multiple command lines.

```
VECTOR C DOUBLE = (1, 4.555, -1E6)
```

Define a three element vector of 64-bit `DOUBLE` data type.

See Also

COPYVEC, **FIRFILTER**

VRANGE

Set the default input voltage range for a sampling configuration.

VRANGE *[<low> <high>* | BIPOLAR=*<high>]*

Parameters

<low>

Input voltage low limit.

WORD CONSTANT | FLOAT CONSTANT

<high>

Input voltage high limit.

WORD CONSTANT | FLOAT CONSTANT

Description

VRANGE is available for Data Acquisition Processor models that have a programmable input voltage range. Only certain specific voltage ranges are allowed. See the manual for each Data Acquisition Processor model for information about supported ranges.

The *<low>* parameter specifies the low limit of the input voltage range. The*<high>* parameter specifies the high limit of the input voltage range. When the alternate BIPOLAR notation is used, only the *<high>* limit is specified, and the implied *<low>* limit is the negative of the high limit. Voltage ranges are typically an exact integer number of volts, with no decimal fraction required.

Because the **VRANGE** command establishes a default condition that will be used for configuring all data channels, the **VRANGE** command should appear as one of the first commands following the **IDEFINE** command.

Examples

```
IDEFINE INP3
   GROUPS  3
```

```
    VRANGE   -10.0 +10.0
    SET IP(0..3)  SPG0
    SET IP(8..11) SPG1
    SET IP(4..7)  SPG2
    TIME 5
END
```

Set the default input voltage range for sampling configuration `INP3` to be –10 to +10 volts. This voltage range is applied to all channel groups in the configuration.

See Also

SET, **IDEFINE**

WAIT

Define a task that selects data according to trigger events.

WAIT *(<in_pipe>, <trigger>, <pre>, [<post>], <out_pipe>)*

Parameters

<in_pipe>
Input data pipe.
`WORD PIPE | LONG PIPE`

<trigger>
The trigger that is examined.
`TRIGGER`

<pre>
The number of values to transfer before the trigger event.
`WORD CONSTANT | LONG CONSTANT`

<post>
The number of values to transfer after the trigger event.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>
Output data pipe.
`WORD PIPE | LONG PIPE`

Description

A **WAIT** task skips data from *<in_pipe>* until *<trigger>* is asserted. When a trigger is asserted, **WAIT** then transfers *<pre>* values from immediately before the trigger event and *<post>* values immediately after the trigger event from *<in_pipe>* to *<out_pipe>*. An effect can never be measured before the event that produces it, so the sample where an effect is first detected is considered to occur after the triggering event. Numbers *<pre>* and *<post>* are nonnegative integers. If specified, *<post>* must not be zero. The number *<pre>* is often called the pre-trigger count and the number *<post>* is often called the post-trigger count. The number of samples retained after each event is *<pre>* + *<post>*. If *<post>* is omitted, the **WAIT** task transfers data continuously after an event occurs.

The data rate into *<in_pipe>* must be the same as the rate of the data that cause the trigger assertion; otherwise, the asserted trigger count does not correspond to the correct *<in_pipe>* data.

WAIT treats an input channel pipe list as a special case. When processing trigger information, **WAIT** multiplies the trigger counts by the number of input channel pipes in the input channel pipe list. **WAIT** assumes that the task asserting the trigger is testing data at the speed of only one input channel pipe. This allows **WAIT** to accurately handle several input channel pipes while the triggering command is scanning a single input channel pipe.

Examples

```
WAIT (IP(0..3), T1, 0, 100, P1)
```

Wait for a trigger assertion on trigger `T1` and after the trigger event, transfer a block of 100 values, 25 values for each channel, from the input channel pipe to pipe `P1`.

```
WAIT (P2, T2, 50, 25, P3)
```

Wait for a trigger assertion on `T2` and transfer from pipe `P2` to pipe `P3` 50 values before and 25 values after the trigger event.

```
WAIT (PX, T1, 0, PY)
```

Wait for a trigger assertion on trigger `T1` and transfer data continuously from pipe `PX` to pipe `PY` starting with the sample of the trigger event.

See Also

CHANGE, **DLIMIT**, **LIMIT**, **LOGIC**, **PEAK**, **TRIGSCALE**

WAVEFORM

Define a task that generates a specified type of wave data.

WAVEFORM *(<type>, <amplitude>, <period>, <out_pipe>*
[,<mod_type>, <mod1> [, <mod2>]])

Parameters

<type>
A value that specifies the type of wave function.
`WORD CONSTANT`

<amplitude>
A value that is one half of the peak to peak range of the output.
`WORD CONSTANT | WORD VARIABLE`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | WORD VARIABLE`

<out_pipe>
Output pipe for wave data.
`WORD PIPE`

<mod_type>
A value that selects amplitude and/or frequency modulation of the output wave.
`WORD CONSTANT`

<mod1>
Pipe for first modulation signal.
`WORD PIPE`

<mod2>
Pipe for second modulation signal.
`WORD PIPE`

Description

WAVEFORM generates wave data and places the data in *<out_pipe>*. The*<type>* parameter specifies the type of wave function.

Wave types are defined as follows:

type 0:	triangle
type 1:	sawtooth
type 2:	sine
type 3:	square

Note: The commands **TRIANGLE**, **SAWTOOTH**, **SINEWAVE**, and **SQUAREWAVE** are identical to **WAVEFORM**, with *<type>* set equal to 0, 1, 2, or 3.

<period> is the number of sample values in each wave. The *<amplitude>* is one half the peak to peak distance of the output wave. The maximum value of *<amplitude>* is 32767.

Three optional modulation parameters may be specified. *<mod_type>* selects amplitude and/or frequency modulation of the output wave. The value of *<mod_type>* must be one of the following:

1. amplitude modulation controlled by the data in *<mod1>*
2. frequency modulation controlled by the data in *<mod1>*
3. amplitude and frequency modulation controlled by the data in *<mod1>* and *<mod2>*, respectively

<mod1> and *<mod2>* are pipes. One value is read from the pipe(s) for each value output by **WAVEFORM**. Modulation values are interpreted as signed binary fractions; they are multiplied by the base amplitude or frequency to obtain the amplitude or frequency.

An alternate method for changing the amplitude or frequency of **WAVEFORM** during execution uses a DAPL variable as the *<amplitude>* or *<period>* parameter of **WAVEFORM**. The value of this variable can be changed during execution using a **LET** command. This is efficient, but cannot adjust the amplitude or frequency continuously, and changes are detected and applied asychronously.

Examples

```
WAVEFORM (2, 1000, 100, P2)
```

Generate a sine wave with values ranging from -1000 to 1000, with a period of 100 samples.

```
WAVEFORM (2, 1000, 100, P2, 2, P3)
```

Generate the same sine wave, with frequency modulation controlled by the data in pipe `P3`.

See Also

COSINEWAVE, **SAWTOOTH**, **SINEWAVE**, **SQUAREWAVE**, **TRIANGLE**

18. DAPL 2000 Messages

The following is a listing of error and explanatory messages produced by DAPL system software.

A message beginning with the word `Error` or the word `Warning` results from system level processing, and is generated by the DAPL system when a configuration or operating error occurs. An error message beginning with `<task>,` where `<task>` is a command name, is generated when the error results in the context of a running task.

Error Messages 0-99 - System Errors

`Error 2: fatal system error`

`Error 3: fatal system error`

`...`

`Error 32: fatal system error`

The DAPL system was notified of a serious fault condition by the processor hardware. Operation is not recoverable and the system must be reloaded and restarted. If the application does not use custom DAPL commands or modules, please report this error to Microstar Laboratories Customer Support. It is possible for applications using a 16-bit custom command or 32-bit downloadable module to experience faults of this kind after: improper use of pointers or indexing, improper use of segment registers, and stack and dynamic memory range errors. Examine the code for custom commands and modules carefully, checking for problems in these areas.

Error Messages 1000-1049 - Configuration Errors

`Error 1004:  illegal command - '...'`
A command was used in an inappropriate context. For example, a **SET** command that is not in an input procedure is an illegal command.

`Error 1006:  illegal parenthesis nesting`
A DAPL expression has mismatched parentheses.

`Error 1009:  illegal symbol type`
A named element has the wrong type for its context. For example, attempting to assign a value to a pipe in a **LET** command where a constant or variable name is required.

`Error 1010:  expression is too complex`
The DAPL expression contains too many operators or too many levels of parentheses.

`Error 1023:  undefined symbol - '...'`
A command contains an undefined symbol name. Usually this error results from neglecting to define a data element before referencing it in a processing task parameter list. This error can also result from spelling a name incorrectly.

`Error 1026:  unmatched parentheses - '...'`
In a command such as **VECTOR** with an initializer list, the parenthesis that terminates the list is missing at the end.

`Error 1029:  illegal decimal point specification - '...'`
In a **FORMAT** command, the decimal point specifier for a parameter is negative or out of range.

`Error 1034:  unmatched parentheses or quotation marks - '...'`
In a situation where either parentheses or quotation marks could occur in an expression, the terminating close parenthesis or quotation mark character is missing. Check that all parentheses or quotation marks match correctly.

`Error 1035:  time is too large`
The maximum sampling or updating time specified is beyond the range supportable by the Data Acquisition Processor model. See the **TIME** command.

`Error 1037: procedure generation failed`

Insufficient system memory is available when attempting to activate an input, output or processing procedure with the **START** command. If the error message suggests a reset, enter a **RESET** command. Otherwise, enter a **STOP** command and then try again to run the configuration.

`Error 1038: input channel pipe number is out of range`

Input channel pipe numbers must be between 0 and n-1, where "n" is the number of channel pipes established by the **CHANNELS** or **GROUPS** command in the input procedure.

`Error 1040: illegal input pin identifier`

An incorrect pin identifier is specified on a **SET** command.

`Error 1041: sampling time is too small`

The sampling time is less than the shortest conversion time supported by the A/D converters on this Data Acquisition Processor model.

`Error 1043: procedure name must be an input or output procedure`

An **EDIT** command can modify input or output procedures configurations but not processing configurations.

`Error 1044: procedure currently is active`

The **EDIT** command cannot modify an input or output configuration that is currently running. Apply a **STOP** or **RESET** command and try again.

`Error 1045: procedure generation failed - RESET the system`

The system ran out of heap memory and cannot complete the activation of an input, output, or processing procedure. Use the **RESET** command to release memory taken during previous operation but never released.

`Error 1046: expecting '=' after pipe name`

A task definition has an invalid syntax. The command begins with a pipe name, but to be a correct DAPL expression, an assignment operator must follow. This error can occur if a pipe name unintentionally appears as the first element in a task definition command line.

`Error 1049: too many output channel pipes`

An output procedure attempts to define more than the maximum number of output channel pipes supportable by the Data Acquisition Processor model.

Error Messages 1050-1099 - Configuration Errors

`Error 1054:  output channel pipe number is out of range`
Output channel pipe numbers must be between 0 and n-1, where "n" is the number of channel pipes established in the output procedure.

`Error 1055:  illegal output pin specification`
A pin identifier on a **SET** command in an output procedure configuration is not supported by the Data Acquisition Processor model.

`Error 1057:  update time is too small`
Output update time is smaller than the digital to analog output converters can support on this Data Acquisition Processor model.

`Error 1060:  illegal input clocking parameter - '...'`
An input clocking parameter must be either `INTERNAL` or `EXTERNAL`.

`Error 1062:  illegal EDIT procedure option - '...'`
See the `EDIT` documentation for a list of accepted **EDIT** options for procedures.

`Error 1063:  command name is undefined - '...'`
When defining a task, an undefined command name was entered. The command must be provided by the DAPL system, by a 16-bit custom command that was previously downloaded, or by a 32-bit module that was previously installed.

`Error 1064:  illegal output com pipe - '...'`
The `OUTPUT=` option in a **FORMAT** command line can only be used to redirect output to an output communication pipe.

`Error 1068:  illegal output clocking parameter - '...'`
An output clocking parameter must be either `INTERNAL` or `EXTERNAL`.

`Error 1069:  illegal output hardware trigger parameter - '...'`
An output hardware trigger parameter must be either `GATED`, `ONESHOT`, or `OFF`.

`Error 1070:  insufficient memory to start input procedure`
All heap memory is filled. Issue a **STOP** command and try again.

`Error 1071:  insufficient memory to start output procedure`
All heap memory is filled. Issue a **STOP** command and try again.

`Error 1072: task creation attempt failed`
A procedure could not be started due to lack of heap memory. Issue a **STOP** command and try again.

`Error 1073: active input procedure does not have a COUNT specification`
The **SAMPLEHOLD** command cannot be used with the currently active input procedure. The procedure must include a **COUNT** specification. Without this, the procedure would never terminate.

`Error 1074: undefined procedure - '...'`
A procedure name given in the parameter list of **START** or **STOP** is not known to the system. The message will show one or more task names beginning with the procedure name that was not recognized.

`Error 1075: procedure already is active - '...'`
An attempt was made to start a procedure that already has been started.

`Error 1077: procedure is not active - '...'`
A procedure that is not started cannot be stopped.

`Error 1078: illegal display parameter - '...'`
The display option specified on a **DISPLAY** command is not recognized. See the **DISPLAY** documentation for a list of supported display options.

`Error 1079: undefined symbol name - '...'`
An **SDISPLAY** or **EDIT** command line specifies an undefined symbol name.

`Error 1080: illegal TASK parameter - '...'`
This message can appear when an incorrect parameter is specified on a `TASKSTAT` command line in older DAPL operating system versions. Use the **STATISTICS** command instead.

`Error 1082: illegal pipe parameters`
A command line option is invalid on a **PIPES** command. For example, the `MAXSIZE` parameter does not specify a valid positive number.

`Error 1084: illegal blocking size`
The transfer blocking size for a communication pipe is out of range. The blocking must be a number of elements greater than zero and less than the maximum buffering size of the output communications pipe.

`Error 1085:  illegal pipe type`
The characteristics of a pipe being modified by an **EDIT** command are inconsistent with the new pipe characteristics specified. The pipe characteristics are not changed.

`Error 1086:  at least one task must be signaled`
Trigger events asserted by a task are not received by any other task.

`Error 1089:  illegal VECTOR syntax`
Unrecognized text appears in a **VECTOR** command line.

`Error 1090:  vector has too many elements`
The number of terms in a vector exceeds the maximum of 8,192 elements.

`Error 1091:  number of wait samples is too large`
The **OUTPUTWAIT** count in an output procedure is larger than available memory resources can support.

`Error 1092:  illegal com pipe number`
Communication pipe numbers must be in the range of zero to 119.

`Error 1094:  PC NUM=<num> must be specified`
A communication pipe definition must specify a communication port.

`Error 1098:  MAXSIZE parameter is too large`
The maximum number of elements for buffering in the user-defined pipe is too large for available pipe storage.

`Error 1099:  inconsistent pipe definition parameters`
Inconsistent pipe options were specified for a communication pipe, for instance, `BINARY` and `ECHO`.

Error Messages 1100-1149 - Configuration Errors

`Error 1101:  expecting '=' after option name`
When changing a system option, the name of the option to be changed must be followed by an equals sign.

`Error 1102:  TYPE parameter is out of range`
`TYPE` parameter in a **`BDOWNLOAD`** command for a 16-bit custom command must be zero if specified.

`Error 1103:  custom command length must be even when reading a word com pipe`
The length parameter in a **`BDOWNLOAD`** command must be an even number of bytes if the input pipe is a word pipe.

`Error 1105:  illegal com pipe edit parameter - '...'`
The parameters of a communication pipe that can be modified by an **`EDIT`** command are `ECHO` , `NOECHO` , `BLOCKING` and `WIDTH`.

`Error 1106:  undefined pipe - '...'`
An **`EMPTY`** or **`FILL`** command line specifies an undefined pipe name.

`Error 1109:  illegal WIDTH parameter - '...'`
The `WIDTH` property parameter on a `PIPE` definition must specify a primitive data type: `BYTE`, `WORD`, `LONG`, `FLOAT` or `DOUBLE`.

`Error 1110:  illegal parameter - '...'`
A **`CPIPE`** command contained an illegal parameter.

`Error 1111:  illegal OPTION command - '...'`
See the **`OPTIONS`** documentation for a list of configurable system options.

`Error 1112:  illegal boolean value - '...'`
The Boolean values used with the **`OPTIONS`** command are `ON`, `OFF`, `YES`, and `NO`.

`Error 1114:  undefined byte or word pipe - '...'`
A **`BDOWNLOAD`** command line specifies the name of a pipe that is not a defined byte or word pipe.

`Error 1117:  missing parameter(s)`
A command line is incomplete. For example, the time interval was omitted from a **`TIME`** command line; the symbol to modify is missing from the **`EDIT`** command line; no sample count is specified on the **`COUNT`** command line; etc.

`Error 1118:  extraneous characters at end of line - '...'`
Unexpected text appears at a point where a command would be complete without the unrecognized text. This error can occur when an optional command parameter is incorrectly typed, a list of comma-separated items is missing a separator comma, or additional non-comment text follows the elements that should terminate the command.

`Error 1120:  missing name`
In a command such as **PIPES** or **LET**, where the user is expected to specify the name of an element, the name is missing from the command line. This error can occur if a required name field is omitted, if there is a duplicated separator such as a comma in a list of names, or if an extra separator character such as a comma appears after the last element of a list.

`Error 1122:  symbol already is defined`
An attempt was made to redefine a symbol. Use **RESET** to remove all previous symbol definitions, or use the **ERASE** command to remove symbol names selectively.

`Error 1127:  this symbol cannot be edited`
Only communication pipes, input procedures, and output procedures can be modified by the **EDIT** command.

`Error 1128:  number is out of range - '...'`
The number entered was too large or too small to be valid in its context. For a command such as **CONSTANTS**, this would mean that the attempt to convert the command text string into a binary number failed because the number was too large to be representable. For a command such as **COUNT**, this could mean that the number is negative or too large.

`Error 1129:  illegal decimal number - '...'`
An element specifying a number either contains invalid characters or has a numerical value that cannot be represented. Decimal numbers can contain only the characters 0 to 9, optionally preceded by a minus sign.

`Error 1130:  illegal hexadecimal word - '...'`
Invalid characters are present in a hexadecimal number notation, or the value represented by the notation exceed the range of a `WORD` data element.

`Error 1131:  illegal hexadecimal long word - '...'`
Invalid characters are present in a hexadecimal number notation, or the value represented by the notation exceed the range of a `LONG` data element.

`Error 1132:  undefined constant - '...'`
While looking for a number value, the command line found a symbol name that might be intended as a named constant, but no constant has been defined with that name.

`Error 1133:  number is negative - '...'`
Only positive numbers are allowed in this context.

`Error 1134:  number is too large - '...'`
The number entered was too large to be valid.

`Error 1139:  CYCLE count is out of range`
The `CYCLE` count in an output procedure must be a nonnegative integer and must not exceed the system memory resource limitation.

`Error 1142:  simultaneous input and output exceeds maximum rate`
Input sampling or output updating rate exceeds the maximum rate allowed when both input and output procedures are active.

`Error 1145:  hardware error`
A hardware error occurred during the system initialization. Contact Microstar Laboratories Technical Support.

`Error 1147:  input procedures must have at least one input channel pipe`
Input procedure configurations must define one or more channel pipes.

`Error 1148:  maximum number of counter/timer channel pipes is exceeded`
The number of counter/timer channel pipes specified in an input configuration is greater than the maximum of 12.

`Error 1149:  a single counter/timer channel pipe is illegal`
A counter/timer configuration is invalid, and data either cannot be captured or read. See the **SET** command and the hardware manual for the counter/timer board for more information.

Error Messages 1150-1199 - Configuration Errors

`Error 1152:  undefined DIAGNOSTIC test`
The test code specified in the **DIAGNOSTIC** command line is not supported by the command. Currently only blank, 0 and 10 are supported.

`Error 1154:  number of wait samples is too small`
The **OUTPUTWAIT** count in an output procedure is unreasonably small. Increase it and try again.

`Error 1157:  illegal UPDATE parameter - '...'`
The **UPDATE** parameter must be either BURST or CONTINUOUS.

`Error 1158:  one or more procedures still are active`
The command entered is not allowed to execute when there are active procedures. Issue a **STOP** command and try again.

`Error 1159:  count must be a multiple of the number of channel pipes`
The **COUNT** parameter must be an integral multiple of the number of channel pipes.

`Error 1160:  output count is less than cycle size`
If both **COUNT** and **CYCLE** are present in an output procedure, **COUNT** must be greater than or equal to **CYCLE** multiplied by the number of channel pipes.

`Error 1161:  count parameter is less than the number of wait samples`
The output updating configuration can never produce output because the **COUNT** command will terminate the updating task before a sufficient number of samples is available to satisfy the **OUTPUTWAIT** command. Increase the number of samples on the **COUNT** command or decrease the number of samples on the **OUTPUTWAIT** command and try again.

`Error 1162:  input burst mode can only be used with HTRIGGER and COUNT`
Input burst mode works only with **HTRIGGER** ONESHOT and a non-zero input **COUNT**.

`Error 1165:  illegal OUTPORT syntax`
The range specification in an **OUTPORT**command line is invalid. Verify that the channel range number and the dot-dot separator are typed correctly.

`Error 1166:  illegal output expansion board type`
An incorrect output expansion board type code was entered. Check the expansion board specification for its correct type code.

`Error 1167:  illegal output expansion port(s)`
An incorrect output expansion board address was entered.

`Error 1168:  pipe already is reserved by a user task`
An attempt was made to use a **FILL** command on a pipe that has been reserved by a user task. Stop the procedure that contains the task and try again.

`Error 1170:  output procedures must have at least one output channel pipe`
Output procedure configurations must define one or more channel pipes.

`Error 1171:  previous input procedure still is active`
An attempt was made to start an input procedure while another input procedure still was active. Stop the active input procedure and try again.

`Error 1172:  task(s) reading from input channel pipe(s) still are active`
An attempt was made to start an input procedure while another input procedure still was active. Stop the active input procedure and try again.

`Error 1173:  previous output procedure still is active`
An attempt was made to start an output procedure while tasks reading data generated by the previous input procedure were still active. Stop the procedure that contains these tasks and try again.

`Error 1174:  task(s) writing to output channel pipe(s) still are active`
An attempt was made to start an output procedure while tasks writing to the output channel pipes of the previous output procedure still were active. Stop the procedure that contains these tasks and try again.

`Error 1175:  pipe is not empty`
An attempt was made to **EDIT** the width of a communication pipe while it is not empty. Empty the pipe and try again.

`Error 1176:  cannot FILL an input com pipe - '...'`
The **FILL** command is not available for input communication pipes. It can only be used to fill a user-defined pipe or an output communication pipe.

`Error 1178:  output count must be a multiple of output channel pipes`

The **COUNT** parameter in an output procedure must be an integral multiple of the number of output channel pipes.

`Error 1180:  outputwait must be a multiple of the number of channel pipes`

The **OUTPUTWAIT** parameter in an output procedure must be an integral multiple of the number of output channel pipes.

`Error 1183:  cannot deallocate a communication pipe`

An attempt was made to delete a communication pipe using an **ERASE** command. For most Data Acquisition Processor models, communications pipes can only be removed through the services of the driver control panel application.

`Error 1184:  slave mode does not support input burst mode`

If the clock source for an input procedure is the master Data Acquisition Processor in a multiple Data Acquisition Processor system, **UPDATE** cannot be set to burst mode for that procedure. See the **SLAVE** and **UPDATE** commands for additional information.

`Error 1185:  slave mode does not support output burst mode`

If the clock source for an output procedure is the master Data Acquisition Processor in a multiple Data Acquisition Processor system, **UPDATE** cannot be set to burst mode for that procedure. See the **SLAVE** and **UPDATE** commands for additional information.

`Error 1187:  cannot FILL a system pipe - '...'`

The **FILL** command is not available for system command pipes. **FILL** can only be used to fill a user-defined pipe or an output communication pipe.

`Error 1188:  illegal system pipe edit parameter - '...'`

When editing the predefined `$CMDIN` pipe, only the `ECHO` or `NOECHO` option may be changed.

`Error 1189:  invalid configuration for fast input sampling - increase the TIME parameter`

In order to use “fast input sampling” to sample faster than the maximum analog sampling rate, you need to interleave enough digital channels between each analog channel so that the analog sampling interval is not smaller than minimum interval allowed. Either rewrite the procedure to use an adequate number of digital channels, or increase the **TIME** parameter.

`Error 1190: all the output channels in the procedure must be specified`

The number of output channels specified in the **ODEFINE** command does not match the number of channels defined by **SET** commands in the output procedure body. Unlike input procedures, in which channels may be skipped, output procedures must assign each channel to a physical device.

`Error 1191: output burst mode cannot be used with CYCLE`

Remove the **CYCLE** parameter in the output procedure and try again.

`Error 1192: fast input sampling is not available with CLCLOCKING and external clocks`

If an input procedure enables both channel list clocking and external clocking, no fast-input-sampling configuration is valid. Increase **TIME** and try again.

`Error 1193: illegal CLCLOCKING parameter - '...'`

An invalid option is specified on the **CLCLOCKING** command line in an input or output configuration. The only valid parameters are `ON` or `OFF`.

`Error 1195: illegal input buffer parameter - '...'`

In the **BUFFERS** command, only the `STATIC` or `DYNAMIC` options are valid.

`Error 1199: cannot have multiple output channel pipes in one task`

A task can write to only one output channel pipe. In most cases, the desired results can be achieved using a single output channel pipe with a channel list. Either multiplex the data for the output channel pipes as the data is loaded into the output channel pipe, or transfer the data through multiple user pipes and combine the streams using a **MERGE** command.

Error Messages 1200-1499 - Task Operating Errors

`Error 1200: <task> - too many parameters (exceeds max line length)`

The **FORMAT** task is unable to format the specified data list for printing because the resulting line length would exceed the available length of 236 characters. Try a shorter output list, or use the slash notation to break the output line into a sequence of shorter lines.

`Error 1206: <task> - too many tasks writing to a pipe`

Multiple processing tasks are attempting to write data to the same pipe. In the case of the output channel pipe, only one task can place data into an output channel pipe. Usually this error occurs because of an invalid processing configuration, but it can also occur if processing procedures are started without first using the **STOP** command to end previous processing.

`Error 1207: <task> - cannot read from an output COM pipe`

A processing procedure is attempting to open an output communication pipe for reading data. Output `COM` pipes transmit data to output communication ports; data cannot be read from these pipes.

`Error 1208: <task> - cannot write to an input COM pipe`

A processing procedure is attempting to open an input communication pipe for writing data. Input `COM` pipes receive data only from input communication ports. Data cannot be written to these pipes.

`Error 1209: <task> - too many tasks being signaled`

When a trigger is defined, and the legacy notation is used that specifies the exact number of tasks that must receive the signal, the specified number is too large.

`Error 1210: <task> - too many tasks asserting a trigger`

When starting a processing configuration, the DAPL system detected that multiple tasks were attempting to assert trigger events into the same trigger pipe.

`Error 1211: <task> - channel number out of range`

During task initialization, the system determined that a task was trying to access an input channel pipe that is not within the range defined by the current input procedure. Check the range of channel pipe parameters for all processing tasks in the configuration.

`Error 1214: <task> - parameter n - '...' should not be a ...`
The nth parameter to a task has the wrong type. (The correct type could not be determined.)

`Error 1215: <task> - too few parameters`
A task determined that the wrong number of parameters were supplied.

`Error 1216: <task> - too many parameters`
A task determined that the wrong number of parameters were supplied.

`Error 1217: <task> - parameter error`
A task found an unspecified error during parameter processing.

`Error 1218: <task> - stack overflow error`
A custom task has insufficient stack to continue executing. Verify that the task is not pushing data onto the processor stack without releasing it, and verify that the stack size is sufficient to support task operation.

`Error 1221: <task> - out of memory`
All memory is used and a request to allocate more memory cannot be granted. Task execution is terminated. Issue a **STOP** command before running the configuration again.

`Error 1223: <task> - cannot write to an input channel pipe`
A processing procedure is attempting to open an input communication pipe for writing data. Input channel pipes receive data from digital or analog-to-digital input ports; data cannot be written to these pipes.

`Error 1224: <task> - cannot read from an output channel pipe`
A processing procedure is attempting to open an output channel pipe for reading data. Output channel pipes transmit data to digital or digital-to-analog output ports; data cannot be read from these pipes.

`Error 1225: <task> - tasks in more than one procedure are reading from a pipe`
Processing tasks in two or more different processing procedures are attempting to read from the same pipe source. All tasks reading from one pipe must reside in one procedure.

`Error 1226: <task> - illegal pipe open option`
A custom command uses an invalid option in the `OPEN_PIPE` function parameter list. See the Developer's Toolkit for DAPL manual for information about the options available for opening a pipe.

`Error 1228: <task> - expecting a user defined or output com pipe`

Only a user-defined pipe or an output communication pipe can be emptied using the **EMPTY** command.

`Error 1229: <task> - coprocessor floating point is not available`

A custom command attempted to access a math coprocessor on a Data Acquisition Processor model and DAPL operating system that does not support floating-point math.

`Error 1230: <task> - this service is not available`

There was an attempt to do an operation other than a pipe `GET`, pipe `PUT`, or pipe `NUM` on a system command pipe. Other pipe operations such as `OPEN`, `CLOSE`, `block GET`, `block PUT`, `REM` and `PURGE` are not allowed on system command pipes.

`Error 1231: <task> - cannot re-start input tasks without re-starting the input procedure`

After stopping a procedure that contained tasks reading data from input channel pipes, restart the procedure only after restarting an input procedure.

`Error 1232: <task> - cannot re-start output tasks without re-starting the output procedure`

After stopping a procedure that contained tasks writing data to output channel pipes, restart the procedure only after restarting an output procedure.

`Error 1234: <task> - data flag value out of range`

A data source or destination flag in a data stream processed by a **SEPARATEF** command is inconsistent with the number of destination pipes specified in the task parameter list.

`Error 1235: <task> - expecting a complete input channel list pipe parameter`

The task requires an input channel parameter to be a complete channel list. For example, if an input procedure has 5 channels, only the channel list "`IPIPES(0..4)`" or "`IPIPES(0,1,2,3,4)`" is valid for the task. This restriction applies to tasks using specialized non-blocking pipe access operations provided by the Developer's Toolkit for DAPL.

`Error 1236: <task> - parameter <number> - ...`

A task found an unspecified error in the indicated parameter position.

`Error 1237: <task> - this task is not compatible with the operating system version`

An attempt was made to execute a custom command task that is not supported by the version of DAPL installed on the Data Acquisition Processor. Call Microstar Laboratories Customer Support and ask for information about recompiling the command code using a newer version of the Developer's Toolkit for DAPL.

`Error 1248: <CmdName> - cannot restart trigger, already active`

If any task using a software trigger is stopped, all tasks using that trigger must stop before the trigger becomes available to new tasks.

`Error 1249: <CmdName> - out of 16-bit selectors; task terminated`

For applications using 16-bit custom commands, making a large number of dynamic memory allocations can result in exhausting all of the processor's selector table entries available for this purpose. If this occurs, consider converting to a 32-bit downloadable module. This eliminates almost all memory allocation restrictions.

Warning Messages 1500-1599

`Warning 1501: time is being rounded down to XXX.XX uS`
The time interval value specified on a **TIME** command is not an exact multiple of the time resolution interval supported by this Data Acquisition Processor model. For example, some models might have a time resolution of 0.1 microsecond, while others might have a resolution of 0.02 microseconds. The sampling time is rounded down to the nearest multiple of the time resolution interval.

`Warning 1504: more than one input procedure`
More than one input procedure exists when a **START** command is invoked without parameters. Only one of the input procedures can start.

`Warning 1505: more than one output procedure`
More than one output procedure exists when a **START** command is issued without parameters. Only one of the output procedures can start.

`Warning 1509: earliest saved options overwritten`
When using an **OPTIONS** SAVE command, the depth of the option stack is 5, and the stack is full. The earliest saved option is overwritten.

`Warning 1510: nothing to restore`
An **OPTIONS** RESTORE command was issued before any **OPTIONS** SAVE command. The option stack is empty; so nothing is done.

`Warning 1530: input channel pipe overflow at sample #`
All buffer memory has been filled by the active input procedure. See Chapter 11. The input sampling procedure is terminated, but samples already collected remain buffered in memory for processing.

`Warning 1531: output channel pipe underflow at sample #`
No data were available at the time when synchronous outputs were supposed to be updated. The output updating procedure is terminated.

`Warning 1532: maximum number of counter/timer channel pipes is exceeded`
Too many channel pipes are configured for using a counter/timer accessory card. Check the hardware manual for the timer/counter board for information about configuring the signal channels.

`Warning 1534: input hardware overflow at sample # <number>`
The DAPL system was unable to respond to a hardware interrupt for servicing a buffering device quickly enough, and this broke the continuity of the sampling.

The hardware sampling procedure is shut down, but processing of data samples already in memory can continue. The sample number indicates the sequence number of the last retained data sample, where samples are numbered starting with zero. This error should never occur unless a custom control task interferes with interrupt processing, leading to a timing fault. If this error occurs for any other reason, please call Microstar Laboratories Customer Support.

`Warning 1535: counter/timer not supported in SLAVE mode`
Data Acquisition Processors operating in **SLAVE** mode cannot access timer/counter accessories.

`Warning 1536: ... hardware overflow at sample # ...`
This is basically the same error condition as Warning 1534, except it is used for the iDSC series boards.

`Warning 1537: option value is not supported by this model`
This message can appear in special circumstances when using the **OPTIONS** command.

- The `FLOATERROR` option is enabled for versions of DAPL and Data Acquisition Processor models that do not support hardware-based Floating Point processing.
- The `AINEXPAND` option is selected for Data Acquisition Processor models that do not support input expansion pin remapping.

`Warning 1538: redundant or conflicting specification`
A configuration command duplicates, overrides, or contradicts a previous specification. For example, this message might occur if a legacy form of the **IDEFINE** command specifies a number of input channels, but later the **CHANNELS** command also define this same number. This warning could mean that the configuration will not behave as expected.

Error Messages 2201-2272 - Configuration Errors

`Error 2201:  requires DAPL floating point extensions`

An operation that requires floating-point extensions was attempted using an incompatible Data Acquisition Processor.

`Error 2202:  division by zero interrupt`

This error should not occur when using built-in DAPL commands. The error occurs when a custom command attempts to divide by zero or attempts to divide a 32-bit number by a 16-bit number and gets a 32-bit result.

`Error 2203:  interrupt #<number> at <segment>:<offset>`

A spurious interrupt occurred — it could be the result of incorrect custom command or downloadable module programming, otherwise it could indicate a serious system error. Contact Microstar Laboratories Customer Support.

`Error 2204:  general protection at <segment>:<offset> Error code = <hex>`

This message indicates a location where an illegal memory access occurred. If the application does not use custom DAPL commands or modules, please report this error to Microstar Laboratories Customer Support. Faults of this kind usually occur through: improper use of pointers or indexing, memory allocations of the wrong size, or initializations that were not performed as intended. Examine the code for custom commands and modules carefully, checking for problems in these areas.

`Error 2205:  fault exception #<number> at <segment>:<offset> Error code = <hex>`

A hardware-related fault condition was detected. This message displays the additional processor-generated fault code as part of the message text. If the application does not use custom DAPL commands or modules, please report this error to Microstar Laboratories Customer Support. An application using a 16-bit custom command or 32-bit downloadable module can cause faults of this kind through: improper use of pointers or indexing, improper use of segment registers, stack and dynamic memory range errors, and some floating point control instructions. Examine the code for custom commands and modules carefully, checking for problems in these areas.

`Error 2206:  undefined interrupt`

A software-requested interrupt occurred. This usually results from execution of invalid program code, such as an invalid downloadable module or assembler coding with an invalid branch instruction. If the application does not use custom

DAPL commands or modules, please report this error to Microstar Laboratories Customer Support.

`Error 2207: floating point exception 16 : ...`

A floating point exception occurred after the **OPTIONS** FLOATERROR=ON was set, or after a custom command adjusted the floating point unit control word directly. DAPL 2000 catches the interrupt and produces this message. No mechanism is available for vectoring the interrupt to relocatable user-defined code. Setting **OPTIONS** FLOATERROR=OFF or directly masking the interrupts in a custom command will avoid floating point errors and allow the task to continue.

`Error 2209: command format is not compatible with the operating system version`

An attempt was made to download 16-bit custom command binary code for a version of the operating system prior to DAPL 2000. Recompile the custom command using an updated version of the Developer's Toolkit for DAPL and try again.

`Error 2211: IPIPES or OPIPES used in the wrong context`

An attempt was made to use IPIPES to define channels in an output procedure or OPIPES to define channels in an input procedure.

`Error 2213: invalid channel range specification`

The channel range notation in a channel list has terms that are not valid integer values within feasible ranges.

`Error 2214: channel numbers in a channel list must be in ascending order`

Some channel pipe numbers in a channel list are not in ascending order. Reorganize the terms in the channel list.

`Error 2217: illegal SCHEDULING option - '...'`

The only options available for **OPTIONS** SCHEDULING are ADAPTIVE and FIXED.

`Error 2218: illegal BUFFERING option - '...'`

The only options available for **OPTIONS** BUFFERING are OFF, MEDIUM, and LARGE.

`Error 2219: scheduling quantum must be smaller than <number>uS`

The QUANTUM option must be set in the range 100μs to 5,000μs. The range depends on the Data Acquisition Processor model.

`Error 2220: scheduling quantum must be larger than <number>uS`
The `QUANTUM` option must be set in the range 100μs to 5,000μs. The range depends on the Data Acquisition Processor model.

`Error 2221: out of memory`
The command to define a new element or task cannot be completed because all memory was previously allocated for other purposes and not released. Issue a **`STOP`** command and try again.

`Error 2222: out of 16-bit custom command stack memory`
There is a 64K heap-space limitation for all 16-bit custom command tasks combined. Either do not run as many instances of the custom command tasks, or reduce the custom command stack sizes.

`Error 2223: illegal channel group specification`
This error occurs when an incorrect `IPIPES` specification is found in a **`SET`** command for a DAP 3400a, DAP 4400a, or DAP 5400a series Data Acquisition Processor input configuration. The specifications for simultaneous sampling groups allow only certain restricted channel sets, so the most likely problem is an incorrect channel number in the channel list notations.

`Error 2224: all channels in the procedure must be specified`
In an input processing configuration for a DAP 3400a, DAP 4400a or DAP 5400a series Data Acquisition Processor, one or more input channel groups specified by the **`GROUPS`** command does not have a corresponding input pin group assigned by a **`SET`** command.

`Error 2225: auto cycle must be longer than holdoff interval`
For a software trigger definition with `AUTO` mode, the **`CYCLE`** property must not be less than the `HOLDOFF` property, otherwise, the trigger would attempt to generate artificial events during the time that events are to be suppressed.

`Error 2226: holdoff must be specified for DEFERRED mode`
The `DEFERRED` mode of a software trigger requires the `HOLDOFF` property to determines when to assert delayed events.

`Error 2227: property '...' not available or already defined`
In a software trigger, either a trigger property is specified that is not compatible with the selected triggering mode, or a redundant property specification attempts to override a value previously defined.

`Error 2228: invalid trigger option - '...'`
There is an incorrect number value assignment for one of the optional parameters on a **`TRIGGERS`** command line.

`Error 2229:  previously specified - '...'`
There is a redundant and possibly contradictory specification. A command line parameter appears more than once. For example, `MAXSIZE=1000` and `MAXSIZE=2000` are both present on the same `PIPE` definition line.

`Error 2230:  error in command keyword syntax`
Unexpected text was encountered in a processing definition command line where a keyword notation in the form "`<keyword> = <value>`" was expected.

`Error 2231:  unrecognized command syntax = '...'`
The command interpreter was not able to interpret the text found in a command line. Check spelling and structure of the command line.

`Error 2232:  invalid constant - '...'`
There is a command syntax error. A numeric or named constant value was expected, but an unrecognized notation was encountered instead. Check the command description in the "DAPL Commands" chapter of the manual or help file.

`Error 2233:  time is too small for counter/timer inputs`
A sampling rate for an input sampling configuration is within the capabilities of the Data Acquisition Processor but too fast for operating a counter/timer card. Use a longer sampling interval for the input sampling configuration. See the counter/timer hardware documentation for information about sampling rate constraints.

`Error 2234:  unable to shutdown communication; DAPL is not restarted`
A **RESTART** command cannot be executed when communications activity is underway on communication channel pipes. If **RESTART** is necessary, shut down all communications activity first.

`Error 2235:  cannot deallocate a system default pipe`
A request to remove one of the system command interpreter channels is denied. These built-in channels are reserved and cannot be removed.

`Error 2236:  name '...' is too long`
In a DAPL command, a user-assigned name for a configuration element such as a pipe or procedure has too many characters. Specify a name with 23 or fewer characters.

`Error 2237:  name '...' contains invalid character(s)`
In a DAPL command, a user-assigned name for a configuration element such as a pipe or procedure contains invalid special characters. Check the DAPL command syntax documentation for information about assigned names.

`Error 2239:  invalid number of channels or channel groups`
The number of channels specified in an **`IDEFINE`** or **`ODEFINE`** command is not a value within the range supported by the Data Acquisition Processor hardware.

`Error 2240:  invalid time interval syntax '...'`
In a **`TIME`** command, either in an input sampling configuration or an output updating configuration, the numeric notation for the time interval is not recognized.

`Error 2241:  invalid IPIPE or OPIPE specifier`
In a **`SET`** command, there is an unrecognized specification where an `IPIPE` or `OPIPE` channel or channel group specifier is expected. See the "Channel Pipe Notations" section of Chapter 2 for information about channel specifiers and acceptable abbreviations.

`Error 2242:  invalid channel pipe syntax - '...'`
The channel pipe range notation following the `IPIPE` or `OPIPE` keyword is invalid. Check that the values cover the correct ranges and are properly enclosed in parentheses.

`Error 2243:  invalid internal clock specifier - '...'`
To use the counter/timer card's internal oscillator as a timing source, a **`SET`** command can accept an additional positional keyword `ICLOCK` after the CT pin specifier. Other options are not allowed with CT pin specifiers.

`Error 2245:  invalid gain specifier`
On a **`SET`** command in an input sampling configuration, the optional notation analog input gain is unrecognized or has an invalid value.

`Error 2246:  invalid one-shot gating option - '...'`
When attempting to edit the `GATE` setting of a software trigger configured in `MANUAL` operating mode, only the keywords `ARMED` or `DISARMED` are allowed.

`Error 2247:  invalid numeric value - '...'`
In a DAPL command, an expression that was expected to evaluate to a number value is not recognized .

`Error 2248: invalid command stack size`
A command stack specifier field in a **BDOWNLOAD** command is not recognized or evaluates to a value that is out-of-range.

`Error 2249: invalid command code size`
A code size specifier field in a **BDOWNLOAD** command is not recognized or evaluates to a value that is out-of-range.

`Error 2250: invalid hardware trigger option - '...'`
The **HTRIGGER** command in an input sampling or output updating configuration can accept only the options `GATED`, `ONESHOT`, or `OFF`.

`Error 2251: ... is not a communication pipe`
An attempt to remove a communication pipe from service failed because the specified pipe was not one of the communications pipes currently configured in the DAPL system.

`Error 2252: symbol '...' reserved or used, cannot erase`
An **ERASE** command is not able to remove the specified procedure, command or data element because the element is currently active, or because the element is a built-in part of the operating system.

`Error 2253: temporary access conflict '...', could not be accessed`
There is a temporary resource conflict. The specified object cannot be accessed at this moment.

`Error 2254: temporary access conflict '...', could not be created`
There is a temporary resource conflict. The specified object cannot be created at this moment.

`Error 2255: temporary access conflict '...', could not be removed`
There is a temporary resource conflict. The specified object cannot be removed at this moment.

```
Error 2259: <CmdName> - memory page fault
      Cause        : ...
      Code address : <segment>:<offset>
      Fault address: DS:<offset> (linear address <address>)
      Access mode  : ...
      Task terminated
```
This message provides diagnostic information for a serious memory access fault detected at the hardware level when memory expected by a task is not available.

This message usually indicates a programming error involving memory allocation, pointer initialization or memory indexing in a custom module. If this message names a DAPL system built-in task, please contact Microstar Laboratories Customer Support.

`Error 2260:  out of selectors`
16-bit custom command tasks made too many requests for dynamic memory allocations of intermediate size memory blocks, and the processor's selector table was exhausted. Try allocating larger blocks with fewer dynamic allocations. Also consider converting to 32-bit module form by recompiling with a newer version of the Developer's Toolkit for DAPL.

`Error 2261:  invalid floating point value`

`Error 2262:  invalid word integer value`

`Error 2263:  invalid long integer value`
A number specifier, even if it has the right syntactic form, cannot be evaluated to produce a representable value of the required data type. This usually indicates a numeric value that is out of range.

`Error 2264:  inconsistent data type`
In a command that needs values of a certain data type, the values provided are of a different type. For example, a floating point value is not acceptable where a 16-bit fixed point constant is needed.

`Error 2265:  missing initializer expression`
In a command such as **VECTOR** where an initialization value or list is required, that list is missing.

`Error 2266:  invalid left-side term for DAPL expression`
A DAPL expression specifies on its left hand side an element such as a constant or input communication pipe that cannot accept computed expression results.

`Error 2267:  bitwise operation with floating point operand`
Bitwise operations such as shift and or logical OR cannot be applied to floating point terms or intermediate values that are floating point. If bit operations of this sort must be performed, produce an intermediate output in a fixed point form, then use a separate DAPL expression to operate upon the data bits of the intermediate values.

`Error 2268:  unexpected end-of-line`
This error occurs when a command line ends without specifying all required elements. This error can occur if a separator character promises that additional elements will appear in the command line, but these are omitted.

`Error 2269:  bad command syntax near '...'"`
The DAPL system is not able to interpret the command line because of invalid text appearing at, shortly before, or shortly after the displayed text.

`Error 2270:  cannot read from output com or channel pipe '...'`
An output communication pipe or output channel pipe appears in a processing command or DAPL expression at a place where a pipe must supply data.

`Error 2271:  symbol '...' has invalid type`
An otherwise valid symbol name is used in a manner that is not meaningful or not allowed. For example, a pipe name cannot be used to specify an initial value for a variable.

`Error 2272:  MAXSIZE parameter is too small`
The storage size requested by a `MAXSIZE` parameter in a **`PIPES`** command is too small. Try increasing this value.

Error Messages 2273-2282 - Downloadable Module Errors

`Error 2273:  installing '...' failed`

There is an unspecified structure error in the downloaded module. This usually indicates that the module binary file is corrupted.

`Error 2274:  installing '...' failed: insufficient memory`

The amount of storage necessary to install the module in the DAPL system memory is unavailable. Verify that a **STOP** or **RESET** command has been used to free storage prior to loading the module. If possible, use smaller storage arrays within the module.

`Error 2275:  installing '...' failed: bad module name`

The name assigned to a module must conform to the requirements of other DAPL system names. The name can contain only alphanumeric characters and underscores.

`Error 2276:  installing '...' failed: bad binary image`

The binary content areas are missing or corrupted in the downloadable module file.

`Error 2277:  installing '...' failed: missing dependency module '...'`

This module cannot be loaded until another module providing required data or functional elements is loaded first.

`Error 2278:  installing '...' failed: could not find '...' export '...'`

A resource expected to be imported from another installed module is not available from that module. This is usually a problem with incompatible versions.

`Error 2280:  installing '...' failed: could not register one of its components`

An element provided by the module could not be recorded in the DAPL system, so that element is not accessible. This usually occurs because of a name conflict. For example, a variable with the same name already exists, or the programmer has assigned the same name to a module and a command defined within that module.

`Error 2281:  installing '...' failed: incompatible with one of the system components'`

When a downloaded module contains a validation function to be called by the DAPL system after module loading, and this validation function reports an error, this error message is issued. This is usually a problem with incompatible versions.

```
Error 2282:  installing '...' failed: incompatible with '...'
interface version ... build version ...
```

When a downloaded module contains a version compatibility function to be called by the DAPL system after module loading, and this compatibility function reports an error, this error message is issued. This is usually a problem with incompatible versions.

Error Messages 2283-2288 - Information Channel Query Errors

`Error 2283:  query buffer is too small; need at least ... bytes of space`

A request packet did not provide for sufficient storage space to return all of the requested information.

`Error 2284:  invalid processing command property structure`

A request packet did not provide for sufficient storage space to return all of the requested information about command properties.

`Error 2285:  missing query parameter(s)`

A request packet did not specify the serial number or range for the elements whose properties are to be accessed.

`Error 2286:  symbol serial number '...' is undefined`

A request packet asks for information about a symbol that does not exist. In some cases, other independent processes can delete the symbol before a query can be completed.

`Error 2287:  '...' does not own the '...' property`

A request packet requests property information about a symbol that does not have the specified property.

`Error 2288:  '...' symbol query does not support the '%s' property`

The information query feature does not have any means to locate the information requested. This message can occur if the property name in the request is misspelled.

Error Messages 2289-2399 - General Errors

`Error 2289:  illegal STATISTICS parameter - '...'`
A **STATISTICS** command line specifies a command option that is not `ON`, `OFF`, `CLEAR`, or `DISPLAYALL`.

`Error 2290:  STATISTICS collection is currently OFF`
An attempt was made to display a **STATISTICS** report without first starting task statistics collection using the `STATISTICS ON` command line.

`Error 2291:  invalid VRANGE specification`
For a Data Acquisition Processor model that supports software-programmable (as opposed to jumper-selectable) voltage range settings, the voltage range specified by the **VRANGE** command is not one of the range configurations supported by the hardware model.

`Error 2292:  GROUPS or CHANNELS command required`
The number of channels for an input sampling configuration was not specified. Add a **CHANNELS** command to the configuration if your Data Acquisition Processor model uses multiplexed single-channel sampling, or a **GROUPS** command if your Data Acquisition Processor model uses simultaneous grouped input channel sampling.

`Error 2293:  specified FORMAT line exceeds ... character limit`
The formatted line is too large to fit the output line buffer size. Try adjusting the field widths, or use the slash notation in the parameter list to split the displayed data into multiple lines.

19. Appendix A. Previous Versions of DAPL

This appendix summarizes obsolete DAPL features that either cannot be supported now or will necessarily become unavailable in future releases.

The DAPL operating system has evolved substantially in response to users' requirements. In the course of this evolution, considerable effort has gone into maintaining compatibility among operating system versions to protect users' investments in applications software. Yet some changes are inevitable, as new generations of high-performance components with greatly increased speed and capacity replace older components that are no longer available. Usually, new DAPL systems cannot both take advantage of the new device capabilities and yet operate within all of the constraints of older devices. Where possible, the DAPL system supports obsolete features by providing special software hooks, allowing applications to continue to operate much as before. It is only a matter of time, however, until the obsolete features become too difficult to maintain in this manner and then they will cease to be available.

New applications should not use the obsolete features listed in this chapter, even if they seem to work. Unless otherwise noted, the features listed are still available and applications that have depended on them in the past will continue to work as before, at least for a while. Existing applications requiring commands or other features that are no longer available must be modified to work with current DAPL system versions. In most cases, there is no loss of functionality because equivalent or superior functionality is available using newer commands.

Reference sheets for selected obsolete commands are provided at the end of this appendix.

System Commands Now Obsolete

BDOWNDLOAD	16-bit custom commands are becoming obsolete as all new systems support 32-bit modules. Applications can download commands using the DAPIO32 programming interface or available utility software such as CDLOAD32.EXE
BUFFERS	This command was introduced to optimize latency in extremely demanding 16-bit custom command optimizations. This command has no benefits in 32-bit systems.
CPIPE	*NO LONGER AVAILABLE.* This previously was needed to support ISA-bus Data Acquisition Processor interfaces.
DIAGNOSTIC	*NO LONGER AVAILABLE.* This command provides no useful information in 32-bit systems.
RESTART	The **RESET** command now covers this. Control of system initialization is now under the Accel32 application rather than the DAPL command interpreter.
TASKSTAT	The new **STATISTICS** command is a general replacement and superior in all regards.

Processing Commands Now Obsolete

AUTORANGE	*NO LONGER AVAILABLE*. The command did not perform well in 32-bit systems and had no users.
FFT mode 7	While the results of mode 7 had a plausible interpretation, nobody used this. If you think you need this, use mode 4 and multiply every output term by 2 using a DAPL expression.
FFT32	**FFT** replaces this. If you specify FFT32, what you will actually get is **FFT**. The previous version of **FFT** used a reduced accuracy algorithm for higher speed, but improved 32-bit numerical processing obviated the speed disadvantage. If you think you need the old algorithm exactly as before, use undocumented processing command FFT16. This links to the old algorithm code and does not support new data types.
MAXTIME	Obsolete. You can achieve the same effect using the **TRIGGERS** CYCLE property when a trigger is defined.
MINTIME	Obsolete. You can achieve the same effect using the **TRIGGERS** HOLDOFF property when a trigger is defined.
RETRIGGER	Obsolete. You can achieve the same effect using the **TRIGGERS** AUTO property when a trigger is defined.

Sampling Procedure Notations

The **IDEFINE** command previously required specification of a number as a command parameter. This parameter is now obsolete. The new **CHANNELS** or **GROUPS** input configuration command should be used instead.

When assigning a CT pin type to an input channel using a **SET** command, the keyword notation ICLOCK should be used to activate the counter-timer board internal clock rather than the *"magic number"* notation. The *"magic number"* notation was a workaround for parsing ambiguities corrected long ago.

Processing Command Changes

In some cases, old commands have been renamed for consistency with new commands or applications. In other cases, new commands have different syntax to allow for

additional options. In all cases, the functionality of the old commands is available using new commands described in the main body of this manual.

The following table lists synonyms or replacements for old commands:

Old	New
CORRELAT	**CORRELATE**
DELTA2	Use DAPL expression to subtract
DISPLAY SAMPLE	**DISPLAY ICOUNT**
DISPLAY OSAMPLE	**DISPLAY OCOUNT**
EXTRACT	Use DAPL expression bitwise operations
FILTER	**FIRFILTER**
LOWPASS	**FIRLOWPASS**
PID	**PID1**
SCALE	Use DAPL expression arithmetic

Support for the **BPRINT** command has changed. In previous version of DAPL **BPRINT** previously had two undocumented, but supported, command forms. These undocumented behaviors are not supported in DAPL 2000.

not supported	equivalent
BPRINT	COPY (<in_pipe>, $BINOUT)
BPRINT (<in_pipe>, <out_pipe>)	COPY (<in_pipe>, <out_pipe>)

Old TRIGGERS Command Syntax

The **TRIGGERS** command in earlier version of DAPL and DAPL 2000 also supported a different command syntax. The old syntax required a single optional number parameter *<readers>* specifying the number of trigger reader tasks to expect. The triggering operating modes and properties were not available. After enough tasks were started to satisfy the *<readers>* specification, the trigger began to allocate and release memory dynamically to support continuous processing. It is not supported for new applications.

Name Conflicts

Older applications might define symbol names that conflict with new DAPL keyword and command names. Such conflicts can be solved by changing the user defined symbol names to other names not reserved by the DAPL system.

Options

The default setting for **OPTIONS SCHEDULING** changed. For Data Acquisition Processor boards for the ISA bus, SCHEDULING=ADAPTIVE has been the default. For current Data Acquisition Processor boards for the PCI bus, SCHEDULING=FIXED is the default setting.

The following **OPTIONS** are obsolete.

FFTSIZE	No longer needed. The DAPL system defers construction of FFT coefficient tables until an FFT task is created.
LLATENCY	Newer options for buffer configuration and scheduling provide better control.
OPTIMIZE	This was misleading. Newer options for buffer configuration and scheduling provide better control.
ROUNDING	The option is accepted but ignored. Less accurate rounding no longer provides any speed advantage.

Hexadecimal Notations

The **OPTIONS DECIMAL=OFF** command previously determined the interpretation of all numerical constants for configuration, input, and output. This led to ambiguous interpretations of DAPL commands and dependencies of current system state upon prior system operation, despite intervening **RESET** commands. In newer versions of the DAPL system, command grammars are not influenced by the DECIMAL option. To use hexadecimal constants when defining a configuration, use the "$" prefix notation. To control displays generated by the **FORMAT** command, use its HEX command line option.

Hexadecimal notations in DAPL expression are now treated as 32-bit representations of bit patterns, where in prior DAPL versions the data type varied depending on the number of digits in the representation. The 32-bit pattern can be converted to a 16-bit number if the high-order 16 bits form a valid sign extension. For example, the constant $FFFFFFF0 would be the equivalent of the decimal value –16, while the constant $FFFF means the same thing as $0000FFFF and would be the equivalent of the value 65535. As 65535 has no 16-bit representation, the DAPL expression would saturate the numerical value to +32767.

Low Latency Tasks

The DAPL system command **OPTIONS BUFFERING=OFF** is treated as advisory by all tasks. Buffering improves system throughput, but it typically results in real-time delays. If there is a means by which a command can move small amounts of data through the system quickly in response to events, then those means will be employed when BUFFERING=OFF is specified. If a command has no such means, or if moving small amounts of data is not the purpose of the command (for example, an FFT always processes data in blocks), the command can ignore this configuration option.

Variables in Parameter Lists

Some DAPL commands that require constants also accept variables in their parameter lists for compatibility with earlier versions of DAPL. The value of the variable is read when the task starts. Further changes to the variable have no effect on the task. New applications always should use constants for these commands to avoid confusion about their operation. The following commands fall into this category.

CHANGE	**DECIBEL**	**EXTRACT**
FREQUENCY	**MINTIME**	**POLAR**
TFUNCTION1	**TFUNCTION2**	**WAIT**

Obsolete Commands

On the following pages, reference sheets for several newly obsolete commands are provided as a courtesy to customers who must support legacy applications. Use of these commands in new and updated systems is not recommended. See the tables in this chapter for information on what should be used in place of these commands.

BDOWNLOAD (obsolete command)

Download a 16-bit custom command binary to the Data Acquisition Processor onboard RAM.

BDOWNLOAD *<command_name> <stacksize> <length> <in_pipe>*

Parameters

<command_name>
Name of custom command to be downloaded.
Alphanumeric text, one to eleven characters

<stacksize>
Maximum stack size of the custom command.
`WORD CONSTANT`

<length>
Number of binary data bytes that are read from *<in_pipe>*.
`WORD CONSTANT`

<in_pipe>
Input data pipe.
`BYTE PIPE | WORD PIPE`

Description

`BDOWNLOAD` defines a command that is downloaded from the PC to the Data Acquisition Processor onboard RAM. The name of the command is specified, followed by the command's maximum stack size and the length of the command's code, in bytes. These parameters are followed by the name of a byte or word pipe.

`BDOWNLOAD` reads *<length>* bytes of binary data from *<in_pipe>*. The bytes contain the code for the custom command. The binary codes are generated using the tools provided in the Developer's Toolkit for DAPL.

In most situations, it is easiest to use the downloading utilities such as DAPview for Windows or CDLOAD32. The CDLOAD32 utility is available with the source code in the DAPDEV\EXAMPLES install directory on the DAPtools CD. These utilities will generate a `BDOWNLOAD` command and format the data stream automatically.

If *<in_pipe>* is a word pipe, *<length>* must be an even number.

Note: **RESET** does not remove any custom command definitions.
ERASE removes custom command definitions selectively.

Example

```
BDOWNLOAD TEST 800 100 $BININ
```

Define the command `TEST` with a stack size of 800 bytes and a code size of 100 bytes; and then read 100 bytes of data from the `$BININ` communication pipe.

See Also

ERASE

BUFFERS (obsolete command)

Specify the input buffer mode of an input configuration.

BUFFERS *<type>*

Parameters

<type>
A keyword, either `STATIC` or `DYNAMIC`

Description

`BUFFERS` specifies the input buffer mode of an input configuration. *<type>* is `STATIC` or `DYNAMIC`. The default mode is `DYNAMIC`.

In `DYNAMIC` input buffer mode, a system task allocates input buffers for an input configuration on demand. The memory used by buffers in the input configuration grows and shrinks dynamically. This is the normal input buffer mode and is appropriate for almost all applications.

In `STATIC` input buffer mode, DAPL pre-allocates a certain number of input buffers for an input configuration. Input processing cycles through these buffers continuously until the input configuration stops or an overflow occurs.

The `STATIC` command is used with certain obsolete multitasking control functions provided in earlier versions of the Developer's Toolkit for DAPL.

Example

```
BUFFERS STATIC
```

Set the input buffer mode to `STATIC`.

CPIPE (obsolete command)

Note: This command is used only for older models of Data Acquisition Processors that use an ISA bus interface.

Define a binary communication pipe.

CPIPE *<name> PC <port> <direction> BINARY [<options>]*

Define a text communication pipe.

CPIPE *<name> PC <port> <direction> TEXT [<options>]*

<options> = [<data_size>] [<echo>] [<buffer_size>] [<blocking>]

Parameters

<name>
Assigned communication pipe name.

<port>
A keyword expression `NUM` = *<n>* where *<n>* is a `WORD CONSTANT` that specifies the DAPL com pipe number.

<direction>
A keyword specifying direction of data transfer, `INPUT | OUTPUT`

<data_size>
A keyword option selecting a data type `BYTE | WORD | LONG | FLOAT`. `BYTE` can be used only with text pipes. The other types can be used only with binary pipes. Defaults are `BYTE` for text pipes and `WORD` for binary pipes.

<echo>
A keyword selecting a command echo option, `ECHO | NOECHO`

<buffer_size>
An option specifying the maximum number of data storage locations of size *<data_size>* available to the com pipe. The default is 2K.

<blocking>
For binary pipes only, the number of data that must be available in the com pipe buffer before data transmission begins. The default is 1.

Description

CPIPE creates a communications pipe. The command must specify the hardware communication port as PC for all Data Acquisition Processors currently supported by DAPL 2000. The PC option specifies high-speed parallel communication over the PC bus.

For PC, the NUM=<*n*> option also must be specified. <*n*> is an integer that specifies the DAPL com pipe number, which is used to establish a unique path to a corresponding com pipe in the PC. Com pipe numbers range from 0 to 119. Com pipe number 0 is reserved for system communication pipes, and com pipe number 1 is reserved for binary communication pipes. The ACCEL driver in the PC supports com pipe numbers from 0 to 31.

INPUT specifies that DAPL receives data from the com pipe. OUTPUT specifies that DAPL sends data to the com pipe.

BINARY or TEXT selects the type of data contained in the com pipe. Text com pipes contain data formatted into lines terminated by carriage returns. DAPL transmits only whole lines from text com pipes.

BYTE, WORD, or LONG selects the width of the data stored in the pipe. Text pipes default to BYTE width, and cannot be declared as WORD or LONG pipes. The com pipe width must match the width of the corresponding PC com pipe. Communications pipes declared as FLOAT can be used with custom commands created with the Developer's Toolkit for DAPL.

ECHO or NOECHO select whether an input text com pipe echoes each received character to the system output pipe $SYSOUT. The ECHO parameter is allowed only for input text com pipes.

The MAXSIZE parameter specifies the maximum number of entries in a com pipe. DAPL statically allocates enough buffer memory to hold MAXSIZE values. MAXSIZE should be large enough to allow efficient communication, but not so large as to waste buffer memory. Typically, a communication buffer should permit between 512 and 2048 entries. Input com pipes on a-Series boards must hold at least 1024 bytes.

The BLOCKING parameter selects whether DAPL forces an output com pipe to contain a minimum number of values before data transmission begins. Using blocking reduces communication overhead in applications that generate large quantities of output data. BLOCKING can be specified only for binary pipes, since text pipes always perform blocking in units of lines.

Parameter defaults for CPIPE are OUTPUT, TEXT, BYTE, NOECHO, and BLOCKING=1. MAXSIZE defaults to 1024.

Note: On all DAPs except for PCI DAPs, communication pipes are removed by **RESTART**. Com pipes are not removed by **RESET**, so **RESET** does not cause loss of buffered data.

Example

```
CPIPE EKGOUT PC NUM=10 OUTPUT BINARY LONG BLOCKING=16
```

Define an output com pipe named `EKGOUT` that sends its data to the PC. The com pipe's number is 10 and it contains 32-bit binary data. Data are transferred to the PC in blocks of 16 values (64 bytes).

See Also

PIPES

DIAGNOSTIC (obsolete command)

The DIAGNOSTIC command is available only on older Data Acquisition Processor models that use an ISA bus interface.

Test Data Acquisition Processor hardware.

DIAGNOSTIC

DIAG

Description

DIAGNOSTIC tests Data Acquisition Processor hardware. The hardware tests take up to four seconds, depending on the model of Data Acquisition Processor. After hardware tests are completed, DIAGNOSTIC sends an error flag to the PC. The error flag is formatted as a single integer; zero indicates no hardware errors, a nonzero number indicates a hardware error. A description of error numbers is provided in the file ERR.TXT on the DAP Software diskettes.

If DIAGNOSTIC detects a memory error, it sends an error message to the PC before the error flag. The error message takes the following form:

```
!! YY ZZ XXXXX
```

YY, ZZ, and XXXXX are hexadecimal numbers. XXXXX is the memory address of an error, YY is the incorrect value, and ZZ is the correct value.

All communication should be finished before sending a DIAGNOSTIC command. After DIAGNOSTIC finishes, a **RESTART** is performed. This is equivalent to a power-up reset. The PC should not attempt to communicate with a Data Acquisition Processor until one second after the error flag is returned. After **RESTART**, all data and symbols in the Data Acquisition Processor are lost.

The **RESTART** command that follows a DIAGNOSTIC command resets several system options such as command echoing. If a DIAGNOSTIC command is sent from DAPview for Windows, these options must be restored. The easiest way to restore the system options is to exit DAPview and run it again.

MAXTIME (obsolete command)

Define a task that restricts the maximum number of samples between trigger assertions.

MAXTIME *(<t1>, <max>, <t2>)*

Parameters

<t1>

Input trigger containing the original event sequence.
`TRIGGER`

<max>

A value that specifies the maximum number of sample counts to wait for a trigger assertion to occur.
`WORD CONSTANT | LONG CONSTANT`

<t2>

Output trigger receiving the modified event sequence.
`TRIGGER`

Description

`MAXTIME` enforces a maximum number of samples between trigger assertions. This command can add extra trigger assertions during periods when there are no events. Each trigger assertion is passed from trigger *<t1>* to trigger *<t2>*. If no trigger assertion occurs within *<max>* sample counts of the previous assertion, an extra trigger assertion is generated each *<max>* sample counts after the previous trigger assertion.

Note: New applications can achieve the same result without using this command by configuring the software trigger `MODE=AUTO`.

Example

```
MAXTIME (T1, 100, T2)
```

Pass each trigger assertion from trigger `T1` to trigger `T2`, adding additional trigger assertions to make sure a trigger assertion occurs at least 100 samples after each previous assertion.

See Also

MINTIME, **NTH**, **RETRIGGER**, **WAIT**, **TRIGGERS**

MINTIME (obsolete command)

Define a task that enforces a minimum number of samples between trigger assertions.

MINTIME *(<t1>, <min>, <t2>)*

Parameters

<t1>
The trigger assertion containing the original event sequence.
`TRIGGER`

<min>
Minimum number of sample counts that the triggers must be separated by.
`WORD CONSTANT | LONG CONSTANT`

<t2>
The trigger receiving the modified event sequence.
`TRIGGER`

Description

`MINTIME` enforces a minimum number of samples between trigger assertions. This command can be used to remove excess trigger assertions. A trigger assertion is passed from trigger *<t1>* to trigger *<t2>* if the trigger assertion is separated from the previous assertion by at least *<min>* sample counts.

Note: For new applications it is better to use the `HOLDOFF` property of a trigger operating in `NORMAL` or `DEFERRED` mode rather than use this command. This suppresses trigger events immediately rather than creating them and later removing them.

Example

```
MINTIME (T1, 100, T2)
```

For each assertion from trigger `T1`, pass the assertion to trigger `T2` if the assertion occurs at least 100 samples after the previous assertion.

See Also

MAXTIME, **NTH**, **RETRIGGER**, **WAIT**

RESTART (obsolete command)

Note: This command only applies to Data Acquisition Processor models that use the ISA bus interface. Other models might respond, in a limited way, by clearing **OPTIONS** to defaults or performing a **RESET**.

Stop all configurations and perform a power-up system restart.

RESTART

Description

RESTART stops all configurations and performs a power-up system restart. User-defined communication pipes and downloaded commands are erased. All other communication pipes are reinitialized. All buffered data are lost. Jumper configurations are read and all **OPTIONS** are set to their default values. Most applications should use the **RESET** command rather than the RESTART command.

All communication should be finished before sending a RESTART command. After a RESTART, the host PC should not send data for one second to allow the Data Acquisition Processor to reinitialize communication.

Note that when RESTART sets the options to their default values, the Data Acquisition Processor is no longer in interactive mode. Echoing and prompting are turned off. If RESTART is issued from DAPview for Windows, the Data Acquisition Processor does not show any response to characters typed from the keyboard until the options are restored to their interactive settings. This can be performed by exiting and reentering DAPview for Windows, or by entering the following option line:

```
OPTIONS PROMPT=ON,SYSINECHO=ON,TERMINAL=ON,\
OVERFLOWQ=OFF,ERRORQ=OFF
```

See Also

EMPTY, **ERASE**, **RESET**

RETRIGGER (obsolete command)

Define a task that reads trigger events and creates a modified trigger sequence.

RETRIGGER *(<t1>, <len>, <t2>)*

Parameters

<t1>
Input trigger.
TRIGGER

<len>
Maximum number of samples to delay a secondary trigger event.
WORD CONSTANT

<t2>
New modified trigger sequence.
TRIGGER

Description

Note: In most applications, it is easier to configure a trigger with MODE=DEFERRED, which achieves exactly the same effect.

A RETRIGGER task reads trigger events from trigger *<t1>* and creates a modified trigger sequence in trigger *<t2>*. The purpose of RETRIGGER is to ensure that data blocks cover clusters of trigger events. If trigger events occur close together, the data for those events can be covered by multiple data blocks so that all data is retained. RETRIGGER normally is used with a **WAIT** task.

RETRIGGER modifies the position of secondary trigger events to ensure that **WAIT** transfers at least pre-trigger and post-trigger values for every trigger. **WAIT** ignores any trigger events that occur while transferring a block of data. If RETRIGGER is used, any trigger event where **WAIT** already is transferring values for its pre-trigger count and post-trigger count causes a new trigger event after that data block. The following description explains how RETRIGGER works with a **WAIT** task:

- The first trigger event in *<t1>* is passed to *<t2>* unchanged.
- If one additional trigger event occurs within *<len>* samples of the previous trigger event, a new trigger is asserted in *<t2>*. The sample count for this assertion occurs *<len>* samples after the previous assertion, which is the first sample count that the **WAIT** task will accept. This causes **WAIT** to transfer data with no gaps or overlaps.
- If two or more trigger events occur within *<len>* samples of the previous trigger event, one new trigger is asserted in *<t2>* to cover all of these events.
- If a trigger event is greater than *<len>* samples past the previous event, the new event is passed to *<t2>* unchanged.

Example

```
CONSTANT PT5=5, PT10=10, PT15=15
...
RETRIGGER(T1, PT15, T2)
WAIT(P1, T2, PT5, PT10, P2)
```

`RETRIGGER` reads trigger events from trigger `T1` and replaces them with trigger events in `T2`, making sure that all trigger events are separated by at least `PT15` samples so that the **WAIT** command covers all trigger events. **WAIT** reads data from `P1` in blocks of 15, then places the values in `P2`.

See Also

MINTIME, **NTH**, **WAIT**

TASKSTAT (obsolete command)

Report information about running tasks.

TASKSTAT *CLEAR | STATUS <option>*

TS *CLEAR | STATUS <option>*

Parameters

<option>

Keyword to select operation, must be CLEAR or STATUS

Description

The TASKSTAT command prints statistics about CPU utilization.

The command TASKSTAT CLEAR resets all CPU utilization variables. This command normally is issued after all configurations have been started. Then, after a time interval, the TASKSTAT STATUS command can be used to examine task activity during the interval.

While an application's configurations are active, a TASKSTAT STATUS command prints a table similar to the following:

Task	CPU Time Used (in ms)
DAPL	394
OVR_CHK	272
UND_CHK	272
MEM_TSK	407
AVERAGE	1096
ALARM	1102
system idle/overhead	6101
Average task cycle latency (in ms):	594

The table lists the total time in milliseconds used by the CPU for each task since the last TASKSTAT CLEAR command. The first four tasks are system tasks that always are executing. All remaining tasks are defined in active processing procedures.

A large value for system idle/overhead typically indicates that the Data Acquisition Processor is performing well. When the Data Acquisition Processor has excess

computational power for an application, it spends much of its time switching tasks while it waits for data. As the demands on the Data Acquisition Processor increase, it spends more time processing data and less time switching tasks. Its efficiency increases as the amount of time spent on system overhead decreases. On the other hand, a heavily-loaded CPU cannot respond to real-time events as quickly.

When **OPTIONS** SCHEDULING=FIXED has been selected, TASKSTAT STATUS also displays the average task latency. The average task latency is the typical number of microseconds of real time between activation of a particular task. See the **OPTIONS** command and Chapter 13.

20. Glossary

The following definitions explain terms that refer to various hardware and software features of the Data Acquisition Processor.

`a Series`
: a Series refers to Data Acquisition Processor models that use the letter ‘a’ in the model name.

`Analog Input`
: An analog input is a hardware pin that connects a continuous voltage signal to the input amplifiers that precede an analog-to-digital converter. When analog input expansion boards are used, the number of available analog input pins is increased.

`Analog Output`
: An analog output is a hardware pin where a continuous voltage is driven by a digital-to-analog converter. A Data Acquisition Processor provides two onboard analog outputs. When analog output expansion boards are used, the number of available analog output pins is increased.

`Asynchronous`
: This is a descriptive term for processes, events or activities that are not coordinated by a sampling or updating clock. Updates of analog outputs using the **DACOUT** command are asynchronous because updates could occur in rapid irregular bursts depending on data arrival time and the number of samples received. Changes of variable values are asynchronous because tasks might process more or less data before they see a change in the variable value, causing the apparent time that the change takes effect to be indeterminate.

`Binary Fraction`
: A fixed point number can be interpreted as a fraction, where the most-significant bit indicates sign, and a binary point is assumed just after the sign bit. The first bit after the binary point is equivalent to a value of 1/2. The next bit is equivalent to a value of 1/2 to the second power, or 1/4. The next bit is equivalent to a value of 1/2 to the third power, or 1/8, and so forth. This pattern continues down to the last available bit. As an example, the binary fraction interpretation is useful when representing FFT windows as a vector of fixed point numbers.

`Block`
: Used as a noun, a block is a collection of associated samples. The samples can be closely related, in the manner of spectrum blocks produced by an **FFT** command,

or the association can be temporary in the manner of data buffered for a bulk transfer. Used as a verb, a task is blocked if it cannot proceed with execution because data is unavailable or because processing of another task prevents it.

`Built-in Command`
The task definition commands provided with each distribution of the DAPL operating system and described in this manual are called built-in because they are loaded by default when the DAPL system starts. There is usually much difficulty and little to be gained from unloading the command module that provides these commands.

`Burst Mode`
Burst mode is a method for input sampling or output updating. During burst mode, a hardware trigger signals for input or output to begin, and sampling or updating continues for a predefined number of samples. When another hardware trigger event occurs, input or output starts again. So, data is received or sent in "bursts."

`Channel Group`
For Data Acquisition Processor models that support simultaneous sampling groups, a channel group is a logical assignment of channels in the input sample pipe to simultaneously-captured samples from an associated input pin group. The channels are numbered consecutively, and correspond in fixed order to the sampled pins. Restrictions on defining channel groups are discussed with the **SET** command.

`Communication Pipe`
A communication pipe (abbreviated "com pipe") is a special pipe used to coordinate transfers of data between the Data Acquisition Processor and its host PC through the PC bus.

`Custom Command Module`
A custom command module is a dynamically downloadable 32-bit code module providing specialized processing commands. Tasks defined by commands in a custom module are equivalent in status to tasks defined by built-in processing commands. They have the same access to internal system features.

`DAPL Expression`
A DAPL expression defines a task that reads from pipes, input channel pipes, or variables, performs arithmetic and bitwise operations, and puts results into a pipe, output channel pipe or variable. DAPL expressions provide flexible means for performing arithmetic and logical operations on data streams.

`DAPL Symbols`
: DAPL symbols are names assigned to elements and recorded within the DAPL system. DAPL symbols can refer to system variables, processing commands, user-defined variables, constants, pipes, and triggers.

`DAPL System Tasks`
: DAPL system tasks are hidden tasks, including the DAPL command interpreter and various system tasks that manage buffers, gather statistics, and perform run-time optimizations.

`Differential Input`
: A differential input is a pair of analog input pins. One of the pins is designated the 'positive' input pin and the other is designated the 'negative' or 'inverting' input pin. The voltage difference between the positive input pin and the negative input pin is measured.

`Digital Input Port`
: A digital input port is a set of digital input pins, typically 8 or 16, that are captured simultaneously. The bits in a fixed-point number represent the state of the pins when sampled. Notations for associating an input channel with a digital input port are discussed with the **SET** command. The number of available digital input ports is increased when digital input expansion boards are used.

`Digital Output Port`
: A digital output port is a set of digital input pins, typically 8 or 16, that are updated simultaneously. For applications that must control output bits individually, some processing commands provide 'masking' ability, so that the updates change the values only of specified pins within the output port. The number of available digital output ports is increased when digital output expansion boards are used.

`Fast Input Sampling`
: Fast input sampling is an interleaving strategy for **SET** commands in an input procedure. By defining a sampling order such that each analog input pin is preceded by a sufficient number of digital input pins, some of the setup time associated with the analog channels can be overlapped with the sampling of the digital channels, allowing a smaller `TIME` interval than sampling of analog channels alone would allow.

`Input Channel Pipe`
: An input channel pipe is a special pipe into which Data Acquisition Processor hardware places analog conversion values and digital input data. Each input channel is associated with an analog input pin, input pin group or digital input port. Voltages at input pins or pin groups are captured and digitized, and the

values stored in multiplexed order. There does not have to be a one-to-one relationship between analog inputs and input channels. Some input channels may be ignored, and some input channels may result from repeated sampling of the same input pin, possibly at different gains. There is really only one input channel pipe, corresponding to the active input procedure, but any number of tasks can read data from the same input channel pipe in different combinations, making it appear as if multiple input channel pipes exist.

`Input Communication Pipe`
: An input communication pipe is a communication pipe that accepts data from the host PC for transfer to the DAPL system.

`Input Configuration Command`
: An input configuration command is a command located within a command group between an **`IDEFINE`** command and its associated **`END`** command. An input configuration command controls the sampling configuration of the Data Acquisition Processor.

`Input Pin Group`
: An input pin group is a set of single-ended analog input pins, determined by the Data Acquisition Processor hardware architecture, for which voltages are captured simultaneously.

`Input Procedure`
: An input procedure is a set of commands between an **`IDEFINE`** command and its associated **`END`** command. An input procedure specifies a configuration that samples various combinations of analog and digital inputs, placing the digitized data into the input channel pipe.

`Multiplexed Input`
: Multiplexed input is a configuration of digital or analog input pins, sampled by Data Acquisition Processor architecture in sequence. Samples from input pins appear interleaved in the captured data stream. When the architecture organizes pins into input pin groups, the data blocks from pin groups are multiplexed within the data stream. When an input channel pipe is used with a channel list notation in a task definition parameter list, data from those channels appear in a multiplexed sequence.

`Multitasking`
: Multitasking is a capability of an operating system such as DAPL to allow multiple processing tasks to execute as though they were completely independent and running simultaneously. The physical processor can only execute one instruction sequence at any given time, so a multitasking system provides a means

of temporarily suspending some processing while resuming other processing, thus allowing all tasks to make progress even if they do not truly run at the same time.

`Output Communication Pipe`
An output communication pipe is a communication pipe that transmits data from the DAPL system to the host PC.

`Output Channel Pipe`
An output channel pipe is a special pipe from which Data Acquisition Processor hardware takes values for clocked digital-to-analog conversion or clocked digital output. Each channel in the output channel pipe is associated either with an analog output pin or with a digital output port.

`Output Configuration Command`
An output configuration command is a command located within a command group between an **`ODEFINE`** command and its associated **`END`** command. An output configuration command controls the output update configuration of the Data Acquisition Processor.

`Output Procedure`
An output procedure is a set of commands between an **`ODEFINE`** command and its associated **`END`** command. An output procedure specifies a configuration that updates analog or digital outputs in any combination.

`Pipe`
A pipe is a high level first-in-first-out buffer for temporary storage of data. Com pipes, input channel pipes, and output channel pipes are special types of pipes. In concept, data are added to one end of a pipe and removed from the other end. The size of elements within the pipe depends on the data type of the pipe. Storage space is allocated and released automatically, allowing a pipe to grow or shrink as required. If data are added to a pipe faster than they are removed, the size of the pipe increases up to its maximum capacity. Attempting to place data into a pipe that has reached its capacity limit, or attempting to remove data from a pipe that has no data, results in the requesting task being blocked. Each task receives data from the pipe in the order in which the data entered. If multiple tasks read data from a pipe, each reader task sees a complete independent copy as if the other readers were not there.

`Predefined Pipe`
When the DAPL system is started, several communication pipes are automatically established: `$SYSIN`, `$SYSOUT`, `$BININ`, and `$BINOUT`. The DAPL interpreter reads commands from the `$SYSIN` input com pipe. Text output produced by the DAPL interpreter and by processing commands is sent to the `$SYSOUT` output com

pipe. $BININ is an input com pipe and $BINOUT is an output com pipe reserved for high-speed binary data transfers between the DAPL system and the PC host.

Print
: A task is said to print data when it sends the data to the PC encoded in a text form. Printing sends data to the screen of the PC or to a printer only under control of a PC application program or utility such as Microstar Laboratories DAPview for Windows.

Processing Procedure
: A processing procedure is a set of commands between a **PDEFINE** command and its associated **END** command. Each command within a processing procedure defines a processing task. Tasks can be defined using built-in processing commands, DAPL expressions, or commands from user-developed custom command modules. All the tasks in a processing procedure are started and stopped together.

Prompt Character
: When DAPL is requesting input from the user in an interactive mode of operation, a character is displayed at the beginning of the input line. This character, usually "#", is called the prompt character. The prompt character changes within input, output and processing procedure definitions.

Scheduling
: Scheduling is the strategy and its application for selecting and running tasks in a multitasking operating system. The DAPL system uses two scheduling strategies. 'Round robin scheduling' allows tasks each to run in sequence. 'Preemptive scheduling' temporarily suspends tasks using too much computing time, so that other tasks can run.

Scheduling Quantum
: The scheduling time quantum is a parameter configured by the **OPTIONS QUANTUM** command. The time quantum specifies how much computing time a task can consume at each opportunity before the preemptive scheduling policy takes effect. System throughput is typically better if a longer time quantum is specified, so that processing completes with the least interruption. However, real-time response is typically better when the scheduling quantum is small, so that tasks do not delay response to a critical real-time event.

Single-ended Input
: A single-ended input is an analog input for which voltages are measured with respect to a common reference ground.

Synchronize
: In the context of task scheduling, synchronization is a process of temporarily imposing sequence constraints on tasks that might otherwise run independently. For example, a task that reads data from a pipe must wait until some other task has placed the data into the pipe. Access policies to preserve referential integrity are sometimes called synchronization. For example, a request to delete a pipe must be denied if a processing configuration uses that pipe. In the context of data streams, two streams are considered synchronized if effects observed at approximately the same place in both streams correspond to the same real-time event. In the context of sampling and updating, synchronization means using a precision oscillator to establish timing intervals for sampling and updating events.

String
: In the DAPL system environment, a string is a sequence of characters enclosed in quotation marks (`" "`). DAPL converts all characters in these strings to upper case. Strings used locally in a custom module programming environment can contain a mix of upper and lower case characters.

Task
: A task is a unit of data processing that occurs when a processing configuration is running. Tasks are defined by commands in a processing definition, but the tasks do not exist until the processing runs. Once processing is started, a task can access data from user-defined, input channel, or communication pipes; modify the data; generate new data; update shared variable values; adjust certain direct outputs asynchronously; process software trigger events; generate message texts; and place results in user-defined, output, or communication pipes.

Task Definition Command
: Each command following the **PDEFINE** command up to its corresponding **END** command is a task definition. Tasks can be defined using built-in processing commands, DAPL expressions, or commands from user-developed custom command modules. A task definition command can be entered several times with different parameters to define separate tasks that execute independently.

Timestamp
: A timestamp is a sample number, determined by a cumulative count of all samples that pass through a pipe. In most applications, the timestamp also equals the number of samples captured by the active input procedure and processed by the task that asserts a trigger.

Trigger
: In the context of hardware, a trigger is a digital logic signal that controls when sampling or updating activity is to start. Subsequent to the trigger event, sampling

activity can continue under control of an independent clock. In the context of processing software, a trigger is a special pipe for synchronizing task data processing. When a trigger is asserted, the sample number of the data value that caused the assertion is placed in the trigger. One task can assert a trigger, and one or more tasks can wait for the assertion of the trigger. When a waiting task receives a trigger assertion, it processes data relative to the exact sample number of the trigger event. Because the triggering is relative to positions within a data stream, rather than real time, a software trigger event can be used to locate data before or after the position of the trigger event.

`Trigger Assertion`
A trigger assertion is recognition by a task of a special condition, typically a data sequence that satisfies special properties, followed by posting of a trigger event in a trigger pipe.

`Trigger Event`
In the context of a hardware trigger, a trigger event is the occurrence of a signal feature that activates the triggering hardware. For software triggering, a trigger event is a trigger assertion indicated by the presence of an event timestamp in a trigger pipe.

`Truncation, Saturation`
In some cases it is impossible to fit the result of a calculation in the storage space allocated for the result. For example, if a DAPL expression adds the values 30000 and 31000 from two word data pipes, the sum of 61000 is greater than the maximum +32767 that can be represented by a 16-bit word value. The range limit depends on the data type. If the data is made to fit by chopping away some of the bits, this is called truncation. For example, if a DAPL expression computes a 32-bit bitwise expression and then stores the results in a 16-bit word pipe, the higher-order 16 bits are truncated. If the data is made to fit by finding the nearest value that is representable, this is called saturation. For example, if a DAPL expression computes a floating point value of 32800.0 and assigns this to a word value, the word value is saturated to +32767 because that is the largest available positive number.

Almost all DAPL operations use saturation, which correctly indicates the sign of the result, and does not yield an artificially small number when the correct result is large.

`Variable`
In the context of the DAPL system, a variable is an element of storage defined by the **`VARIABLES`** command. The storage of a variable is available for shared access by DAPL processing tasks and PC applications. In the context of a custom module

programming environment, a variable is storage known only to the task, and it is not accessible by other tasks. A variable always contains the most recent value transferred to it.

Index