DAPL 3000 Manual

Data Acquisition Processor
Analog Accelerator Series

Version 1.00

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 MSDAPL3000M100

Contents

Section I. Overview

1. Introduction

A Data Acquisition Processor (DAP) from Microstar Laboratories is a self-contained data acquisition system configured and operated by a host PC. The xDAP family includes semi-autonomous DAP products that connect to the PC through an external connection, like a USB port. Other Data Acquisition Processor models mount in peripheral bus slots in the host PC. Both kinds of DAP products are suitable for a wide range of applications in laboratory and industrial data acquisition and control.

Four other manuals complement this DAPL Manual:

- The Data Acquisition Processor Installation Guide contains installation instructions.
- A Data Acquisition Hardware manual describes physical connections and hardware-configurable options.
- The DAPstudio Manual describes using the DAPL Development Studio software to develop and test application configurations.
- The Applications Manual shows how to apply DAPL onboard processing for a wide variety of data acquisition applications.

Scope of This Manual

This edition of the DAPL Manual contains information for version 1.00 and above of DAPL 3000. Many functions are consistent with DAPL 2000 and earlier versions of the DAPL operating system, but there are some fundamental differences:

- DAPL 3000 has better support for high channel count and high data rate applications.
- DAPL 3000 supports multi-priority processing, offering new possibilities for real-time applications.
- Extended precision data types are widely supported. If a computation with floating point data is meaningful, it is almost always supported.
- Several new processing commands are added. Old commands are improved.
- The chapter on Previous Versions of DAPL discusses older command forms that are superseded in current versions of the DAPL 3000 system.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. Utility software such as *DAP Measurement Studio* (DAPstudio) can assist with preparing and running the configurations. 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.

For processing data, DAPL system enforces scheduling rules that grant priority to tasks with critical real-time actions, while otherwise enforcing “scheduling fairness” so that every process that needs to perform a processing action gets an opportunity to do so. If any task exceeds its allowed time allocation or cannot proceed for any other reason, the DAPL system goes on to schedule the next task.

Most input and output activity is driven by a stable digital clock, at configurable time intervals. Some input and output actions, however, must be done “as soon as possible” and cannot wait for the usual buffering and timing mechanisms. For these special cases, it is possible to transfer directly to the appropriate hardware port “asynchronously,” bypassing the usual timing intervals.

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 correspond 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 with a user-defined processing procedure `ANALYZE`.

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

Element Definition Commands

These commands define shared data elements used by processing tasks.

The element definition 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 a DAPL configuration maintain a fixed value while the configuration is running. VARIABLE elements 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.

STRING elements can provide a means of encoding complex information, such as operating modes and options, not easily delivered in the form of number values and data streams. Custom-developed processing commands can use a string to identify a physical process, for example “Condenser 2 Pressure Valve,” and this text can then be used in control and status messages to identify the affected system element.

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
  CHANNELS  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. A processing 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. Or, you can think of each task as being an “independent copy” of the command with a unique collection of associated data elements.

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
```

While this looks very much like an assignment statement in an ordinary programming language, 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

Command names can be given in upper case or lower case letters. For example, the names "skip" and "SKIP" are treated the same within the DAPL system.

Names

Names assigned to shared elements must begin with an alphabetic character and may contain of 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.

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 text line can be terminated by a combination of a carriage-return character followed immediately by a linefeed character, or a combination of a linefeed character followed immediately by a carriage-return character. Otherwise, each carriage-return or linefeed character terminates a separate line. These conventions make DAPL configuration scripts compatible with text conventions on most operating systems. Output texts such as messages are terminated by a carriage return character followed by a linefeed character. Most operating systems can display this text without difficulty, though some might show an extra blank line between actual text lines.

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. For example:

```
TIME  $1F     : 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. Channels do not have to be in a strictly ascending order, but it is most efficient to arrange them that way so that the DAPL system does not need to shuffle the data for transfers with input or output processing. Channel 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 conventional 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 of speed and efficiency. When the application runs, what is the speed penalty, compared to (say) a hand coded application?

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

Recall, the high level commands for configuring 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. 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 `$SysIn` command pipe under software control, using the software facilities of the `DAPIO32.DLL` programming interface. Utility programs like DAPstudio, or applications that you write, all use this approach. But it is also possible to transfer configuration information manually and directly to the DAPL interpreter interactively. For this kind of manual operation, typically the interpreter is put into the `SYSINECHO` mode using the `OPTIONS` command. Any input text is echoed back to the sender through the `$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 many pre-defined processing operations in parallel, 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 more tightly bounded.

Custom command modules are written in the C++ programming language, compiled into a binary code image, and downloaded to a Data Acquisition Processor using a utilities such as CDLOAD32, an application generator such as DAPstudio, or manually using the Accel Console Command Application. Tools for preparing the command code, all 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:

`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
`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

`BUFFERS`	specify the input buffer mode
`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
`SAMPLE`	select continuous or burst input operations
`SET`	associate a channel pipe with a physical pin or pin group
`TIME`	select the sampling interval
`VRANGE`	set configurable input voltage range limits

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. These commands are not applicable for models that do not support output signal generation.

`ODEFINE`	begin an output configuration definition
`END`	complete an output configuration definition

`CHANNELS`	configure the number of channels delivering data
`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

A *processing configuration* is a group of commands that defines a sets of interrelated tasks to be executed together. A processing procedure begins with a `PDEFINE` statement, ends with an `END` statement, and contains commands that define the processing tasks. An execution priority for the group of tasks can also be assigned. 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, vector, or string data, or report software trigger events.

Task Definition Command Summary

The following task-defining commands are provided with the DAPL operating system:

PDEFINE	begin a processing configuration
END	complete a processing configuration
PRIORITY	establish the execution priority of tasks defined in a processing procedure
ABS	compute absolute values
ALARM	generate digital alarm signals
AVERAGE	average pipe data
BAVERAGE	perform block averaging
BMERGE	merge blocked data
BMERGEF	merge blocked data with identifying flags
BOUND	saturate data stream range within adjustable limits
CANGLE	compute polar angle (phase angle)
CHIRP	generate a swept frequency cosine waveform
CMAG	compute polar magnitude (complex absolute value)
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
DFT	compute discrete Fourier transform terms directly
DIGITALOUT	send data to a digital output port
DLIMIT	scan data for slopes that are out of range
FFT	compute fast Fourier transforms
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
INTERP	interpolate with a lookup table
LOADING	simulate CPU loading for data flow and latency studies
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 multiple pipes
MERGEF	merge data from multiple, adding identifying flags

NMERGE	merge different size data blocks from multiple pipes
NTH	remove excess trigger events
PCASSERT	generate triggers based on PC control
PCOUNT	count the number of values placed in a pipe
PID	apply closed-loop regulation to a dynamic process
PIDLATCH	override for smoothly activating/deactivating PID control
PIDRAMP	smoothed transition shapes for control loop level changes
PIDSCRAM	safe and smooth emergency shutdown control override
POLAR	convert from complex rectangular to polar form
PULSECOUNT	count the number of digital input pulses
PVALUE	determine the most recent value in a pipe
PWM	perform pulse width modulation
QDCOUNT	convert 16-bit quadrature decoder board counts into a running sum
QDECODE	directly count quadrature encoder pulses without a decoder board
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
RSUM	running sums and integration, with scaling and decimation
SAWTOOTH	generate sawtooth waveforms
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
STDDEV	compute standard deviation values
TAND	combine triggers with logical ‘and’
TCOLLATE	combine triggers producing a combined event stream
TFUNCTION1	calculate transfer characteristics using direct methods
TFUNCTION2	calculate transfer characteristics using 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

Functional Categories of Tasks

The following lists identify tasks that are most likely to be useful for each given category of application requirements.

Data Selection

Tasks useful for extracting data of special interest from data streams.

DLIMIT	scan data for slopes that are out of range
FINDMAX	find the locations of maxima in blocks of data
LIMIT	scan data for values that are out of range
LOGIC	scan binary data for bit transitions
NTH	remove excess trigger events
RANGE	remove data values that are out of range
RANGE	remove data values that are out of range
SEPARATEF	separate flagged merged data
SKIP	delete selected blocks of data
TAND	combine triggers with logical 'and'
TOGGLE	test trigger events for alternating ON and OFF events
TOGGWT	collect data between alternating ON and OFF events
WAIT	wait for a trigger event and transfer trigger data to a pipe

Data Reduction and Statistics

Tasks that reduce raw data to statistical information.

AVERAGE	average pipe data
BAVERAGE	perform block averaging
COMPRESS	compress data flow for inputs that change infrequently
FINDMAX	find the locations of maxima in blocks of data
FIRFILTER	apply digital filtering and reduce data volume
FIRLOWPASS	apply predefined lowpass digital filtering
HIGH	compute maxima of blocks of data
LOW	compute minima of blocks of data
PCOUNT	count the number of values placed in a pipe
PVALUE	determine the most recent value in a pipe
RANGE	remove data values that are out of range
RSUM	running sums and integration, with scaling and decimation
RMS	compute root mean square values
SKIP	delete selected blocks of data
STDDEV	compute standard deviation values
VARIANCE	compute the statistical variance of pipe data

Data Management

Tasks that move, replicate, combine, separate, remove data.

BMERGE	merge blocked data

BMERGEF	merge blocked data with identifying flags
LOADING	simulate CPU loading for data flow and latency studies
MERGE	merge data from multiple pipes
MERGEF	merge data from multiple, adding identifying flags
NMERGE	merge different size data blocks from multiple pipes
COPY	copy the data in a pipe into several other pipes
COMPRESS	compress data flow for inputs that change infrequently
COPYVEC	copy data from a vector to a pipe
MERGE	merge data from multiple pipes
MERGEF	merge data from multiple, adding identifying flags
REPLICATE	copy data, repeating each value a specified number of times
SEPARATE	separate merged data
SEPARATEF	separate flagged merged data
SKIP	delete selected blocks of data

Waveform Generation

Tasks useful for generating data sample sequences of real-time waveforms.

CHIRP	generate a swept frequency cosine waveform
COPYVEC	copy data from a vector to a pipe
COSINEWAVE	generate cosine waveforms
SINEWAVE	generate sine waveforms
TRIANGLE	generate triangle waveforms
SQUAREWAVE	generate square waveforms
SAWTOOTH	generate sawtooth waveforms
RANDOM	generate pseudorandom numbers
FIRFILTER	apply digital filtering and reduce data volume

Event Processing

Tasks that identify special event conditions and apply selected processing in response.

ALARM	generate digital alarm signals
DACOUT	send data to an analog output port
DIGITALOUT	send data to a digital output port
DLIMIT	scan data for slopes that are out of range
LIMIT	scan data for values that are out of range
FINDMAX	find the locations of maxima in blocks of data
LOGIC	scan binary data for bit transitions
NTH	remove excess trigger events
PCASSERT	generate triggers based on PC control
TAND	combine triggers with logical 'and'
TCOLLATE	combine triggers producing a combined event stream
TGEN	generate periodic triggers
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'
TRIGARM	allow a task to arm or disarm software triggers

TRIGSEND	transfer trigger information to another DAP
TRIGRECV	recover encoded software trigger information
TRIGSCALE	modify a stream of trigger events
TSTAMP	convert trigger assertions to time stamps
WAIT	wait for a trigger event and transfer trigger data to a pipe

Digital Signal Processing

Tasks related to numerically-intensive DSP processing.

CMAG	compute polar magnitude (complex absolute value)
CANGLE	compute polar angle (phase angle)
CORRELATE	compute cross correlations
CROSSPOWER	compute cross power spectrum
DECIBEL	convert positive values to decibels
FINDMAX	find the locations of maxima in blocks of data
FIRFILTER	apply digital filtering and reduce data volume
FIRLOWPASS	apply predefined lowpass digital filtering
FFT	compute fast Fourier transforms
DFT	compute discrete Fourier transform terms directly
TFUNCTION1	calculate transfer characteristics using direct methods
TFUNCTION2	calculate transfer characteristics using power spectra

Signal Conversion and Filtering

Tasks that modify the contents of data streams.

ABS	compute absolute values
BOUND	saturate data stream range within adjustable limits
CANGLE	compute polar angle (phase angle)
CMAG	compute polar magnitude (complex absolute value)
DELTA	compute the differences of the values in a pipe
FIRFILTER	apply digital filtering and reduce data volume
FREQUENCY	determine frequencies of trigger assertions
POLAR	convert from complex rectangular to polar form
PULSECOUNT	count the number of digital input pulses
PWM	perform pulse width modulation
QDCOUNT	convert 16-bit quadrature decoder board counts into a running sum
QDECODE	directly count quadrature encoder pulses without a decoder board
RAVERAGE	compute running averages of pipe data
RSUM	running sums and integration, with scaling and decimation
SKIP	delete selected blocks of data
SQRT	compute square roots
THERMO	perform thermocouple linearization on data
INTERP	interpolate with a lookup table

Monitoring and Process Control

Tasks for stimulus-response processing in real time.

`ALARM`	generate digital alarm signals
`DACOUT`	send data to an analog output port
`DIGITALOUT`	send data to a digital output port
`FIRFILTER`	apply digital filtering and reduce data volume
`LIMIT`	scan data for values that are out of range
`DLIMIT`	scan data for slopes that are out of range
`PID`	apply closed-loop regulation to a dynamic process
`PIDLATCH`	override for smoothly activating/deactivating PID control
`PIDRAMP`	smoothed transition shapes for control loop level changes
`PWM`	perform pulse width modulation

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 the bits of the left-side expression by the number of positions specified in the right-side expression. The bits can be shifted left or right. The result of a shift is a bitwise value.

The left side expression is a bitwise value, or a fixed point value treated as if it were a bitwise value. Shift operations cannot be applied to floating point operands or floating point intermediate results. The number of bit positions to shift can be a fixed point number, or a bit pattern interpreted as a fixed point number, in the range 0-31. For the special case of a zero shift, there is no change to the original left operand value, but the result is always considered to be a bitwise pattern.

The shift operations include the following:

shift left	<<
shift right	>>

Example: Shift the high-order 4 bits of a 16-bit number to the low order position, and force all other bits to zero.

```
        P2 = (P1 >> 12) & $0000000F
```

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.

DAPL Expression Application Examples

Example 1: Complement the four low-order bits you receive from a digital port, so that instead of “active low” they are “active high.”

Use the “exclusive OR” operator. Apply a “bit mask” with 1-bits in the positions to be inverted, and with 0-bits elsewhere.

```
        P2 = P1 ^ $0000000F
```

Example 2: Produced statistically unbiased white noise from random numbers.

The `RANDOM` command produces random numbers in the range 0 to 32767 with uniform probability. If you subtract 16384 from every value, this is almost balanced, but not quite, since the adjusted range is from –16384 to +16383. To get unbiased random numbers (with triangular distribution on interval –32767 to +32767), generate two random number streams and use the following expression to combine them.

```
    rand   =  rand1 - rand2
```

Example 3: Apply nonlinear sensor calibration.

Many sensors show nearly linear, but not perfectly linear, characteristics. For example, a desired strain measurement is almost proportional to the observed voltage, but not exactly. You can fit a low-order polynomial curve that will produce better conversion accuracy than a simple linear approximation.

```
    strain   =  ST1   +   Vmeas * ( ST2 + Vmeas * ST3 ) )
```

Example 4: Simulate a real-time signal by combining multiple waveforms.

DAPL processing commands for computing waveforms have no timing constraints, and they could deliver data in massive, irregular bursts. To deliver data on a regular basis, use the arrival of an input sample to regulate the timing. The DAPL expression cannot evaluate until each input sample arrives, but when it does, you don’t need the value of the sample so force it to zero.

```
    realtime   =  wave1   +   wave2  + noise   +   IP1*0
```

Example 5: Hide the bits on a digital port so they cannot produce a trigger event until the supervisory software sets an enabling variable.

Initialize variable `HOSTMASK` to zero. This overrides the values that the following DAPL expression places into pipe `maskedpipe`, forcing all bits to zero and hiding readings of the digital port received through channel pipe `IP(5)`.

To make the bits from the digital port visible, the host sends the DAPL system a '`LET  HOSTMASK = $ffffffff`' command. This allows the bits received from the digital port to pass through to `maskedpipe` unchanged.

```
   maskedpipe  =  IP(5) & HOSTMASK
```

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 numbers are related.

Analog Input Voltages

The input voltage for each analog input pin is tracked, amplified by the pin's gain factor, and fed to the analog-to-digital converter. The gain factor is from 1 to 40, and is specified independently for each input channel pipe. Each gain setting is independently calibrated, so only a few special values are allowed. After the amplification, the resulting voltage levels must be within the range limits configured for the analog to digital converters.

Different Data Acquisition Processor models support different voltage ranges, so check your hardware manual. Most models support the bipolar voltage ranges of -5 volts to +5 volts and -10 volts to +10 volts. Unipolar voltage ranges start at 0 volts and go up to +5 volts or +10 volts. Unipolar voltages can be measured using half of a bipolar converter range, and all measurements results will be non-negative values (unless disturbed by offsets or noise). On DAP models that provide hardware-configurable unipolar ranges, the full converter range is applied to the positive voltages to preserve an additional bit of precision.

Voltage range selection require hardware configuration, so check your Data Acquisition Processor Hardware Manual about how to select the voltage range.

The result of sampling an analog voltage is that an analog-to-digital converter generates a binary pattern of bits. The bit patterns can be interpreted as numbers identifying equally spaced intervals within the voltage range limits.

Digital Input Voltages

Digital inputs are voltage signals with only two significant levels, `low` and `high`. `Low` is represented by a bit with value 0, and `high` is represented by a bit with value 1. Data Acquisition Processor inputs follow the standard TTL specification that any voltage between 0.0 and 0.8 volts is `low`, and any voltage between 2.0 volts and the supply voltage, approximately 5.0 volts, is `high`. Digital signals that originate from TTL-compatible components and connect through a low impedance cable should never be a problem. Otherwise, voltages applied to digital input pins must be between 0.0 and 5.0 volts. Digital input devices might be damaged if this precaution is not observed. Voltages between 0.8 volts and 2.0 volts are regarded as transition voltages, and may be sensed as either low or high. Avoid voltages in this range except during fast logic level transitions. This reduces the risk of a metastable latch-up state in which digital circuits can be damaged.

There are pull-up resistors on all digital inputs. Because of these pull-up resistors, unconnected inputs appear as 1's when the Data Acquisition Processor measures them.

When digital inputs are captured by a Data Acquisition Processor, 16 input pins of a digital input port are captured simultaneously. Bit positions are conventionally numbered from 15 down to 0. Though the result is actually a collection of 16 independent bits, the collection can be processed like a 16-bit number.

Hexadecimal notations are often useful for compactly representing the patterns of bits on a digital port.

Conversion Values

The pattern of bits produced by an analog-to-digital converter can be interpreted as an integer value. With a 14-bit converter, there are 16384 discrete measurement values possible. With a 16-bit converter, there are 65536 discrete measurement values possible. These values are called conversion values. All conversion values are scaled so that a reading of zero represents zero volts, and 32768 counts corresponds to the supply voltage range. Due to limitations of the fixed-point number system, the actual measurement limits are –32768 to +32767. For unipolar inputs, negative values are not used, so the range of the conversion values is 0 to 32767. To obtain consistent scaling when converters have fewer than 16 bits, padding bits with value zero fill the low order bits to make a total of 16 bits.

With a 14-bit converter, the padding causes each increment of the converter to increase the numerical conversion value by 4. For a 16-bit converter, each increase of the converter count by 1 increases its numerical conversion value by 1, and there is no padding.

The following formula relates the voltage, conversion value, and full scale. Let `X` represent a conversion value, and let `F` represent the full scale voltage — 5 volts or 10 volts. The voltage `V` that the conversion value `X` represents is given by:

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

The range of converter values is not exactly symmetrical. For a 14-bit converter, there is one converter value for 0, and 8192 converter values for negative numbers, leaving 8191 values for positive numbers. Similarly, for a 16-bit converter, there is one conversion value for 0, and 32768 converter values for negative numbers, leaving 32767 values for positive numbers. Thus, conversion values are able to reach the negative conversion limit exactly, but they stop one step short of the positive conversion limit.

- -32768 to +32766 for 14-bit Data Acquisition Processors conversions
- -32768 to +32767 for 16-bit Data Acquisition Processors conversions

Binary Representation of Numbers

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”.

All Data Acquisition Processors are calibrated so that binary `0000 0000 0000 0000` represents zero volts (ground) in all ranges.

Incrementing the zero value by 1 gives a slightly positive binary result of

```
0000 0000 0000 0001
```

Decrementing the zero value to obtain –1 produces a “borrow” action that flips the low order bit and all of the rest of the bits along with it.

```
1111 1111 1111 1111
```

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. Thus we conventionally interpret the first bit as a “sign bit.”

```
s xxx xxxx xxxx xxxx
```

In the normal bipolar operating mode, the converters will supply the sign bit and as many of the remaining bits as possible. Any bits it cannot fill are padded with 0 values. Thus, a 14-bit converter would need two padding bits.

```
s xxx xxxx xxxx xx00
```

In a unipolar operating mode, the sign bit will always be 0, so this bit is predeteremined. The coverter can supply as many of the remaining bits as possible. For this mode, a 14-bit converter would need only one padding bit.

```
0 xxx xxxx xxxx xxx0
```

Hexadecimal Representation of Numbers

The binary representation is helpful for illustrating digital lines, but it is too bulky to use directly. For a more compact notation, the binary notations can be grouped into 4-bit patterns, and each pattern assigned a hexadecimal (base 16) numerical code.

0000	**0**	0001	**1**	0010	**2**	0011	**3**
0100	**4**	0101	**5**	0110	**6**	0111	**7**
1000	**8**	1001	**9**	1010	**A**	1011	**B**

1100	**C**	1101	**D**	1110	**E**	1111	**F**

With this convention, the following decimal values can be represented using hexadecimal codes as

```
1      0001
20     0014
-1     FFFF
0      0000
```

To distinguish numbers using a hexadecimal notation in the DAPL system, prefix them with a dollar sign notation:

```
$0001
$FFFF
$0000
```

For 16-bit integers, only hex notations $0000 through $FFFF are meaningful. Any bits higher than 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.

Hexadecimal notations are easily extended to cover 32-bit integers. Simply include an additional four hexadecimal characters to represent the additional 16 bits of precision.

```
1      $00000001
20     $00000014
-1     $FFFFFFFF
0      $00000000
```

Both for 16-bit and 32-bit values, the sign bit is covered by the first hexadecimal code. Thus, any hexadecimal notation with the codes 8, 9, or one of the letters for the first position represents a negative value.

When using a hexadecimal notation, any high-order bits that you omit default to zero. This can produce some unexpected side effects if you are not careful. For example, you know that the value –1 is represented by the 16-bit value `$FFFF`. So, if you initialize a 32-bit value with this, you will get a surprise.

```
CONSTANT   WRONGVALUE  LONG =  $FFFF
```

The value of WRONGVALUE is set to 65535, not –1. The unspecified higher-order bits default to zero, including the sign bit, but to be valid as a negative number the sign bit should be a 1. The correct initialization is

```
CONSTANT   RIGHTVALUE  LONG =  $FFFFFFFF
```

There are two places where indeterminate length is a problem for hexadecimal notations, and the behaviors are different. In DAPL expressions, all unspecified hexadecimal notations are presumed to be 32-bit values, because the DAPL expressions compute with 32-bit integers. In task parameter lists, there is no prior information about how a hexadecimal value will be used. 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 as a 16-bit `WORD` value. However, if the hexadecimal notation contains five or more hex digits, even if they are leading zero digits, it is presumed that the notation is intended to define a 32-bit `LONG` value. Consequently, $FFF0 would be interpreted as the `WORD` value –16, while $0FFF0 would be interpreted as the `LONG` value +65520.

Interpreting Conversion Values as Binary Fractions

Suppose that the input range is considered normalized, so that the minimum conversion range limit of –32768 means the value –1.0 and the maximum conversion range bound of +32768 means the value +1.0. Any other value between the range limits represents a fraction equal to the literal binary value divided by 32768.

In a binary notation, dividing by a power of 2 is similar to dividing by a power of 10 in a decimal number system — except that instead of shifting the decimal point, it shifts the "binary point." Dividing by 32768 has the effect of shifting the "binary point" to the position just after the sign bit in the binary representation. Retaining the convention that the first, most significant bit is the "sign bit" we can interpret the binary fraction form of a 14-bit conversion value as

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

or for 16-bit conversion values,

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

In some processing commands such as `FFT` that deal with phase angles, the binary fraction representation has an additional implied scaling factor. For the case of FFT, the implied scaling factor for phase angles is equal to Π, and the normalized range –1 to +1 represents one complete rotation in phase angle.

Number Representations Used by DAPL

The range from -32768 to +32767 produced by converters is too restrictive for some computations. DAPL supports alternative data types that can be useful for intermediate computations and presentation of final results.

WORD data type

If you are using only simple transfers of captured data, or simple processing such as averaging, the processing is most efficient in the original 16-bit form. The 16 bit integer values are called `WORD` values (short signed integers when developing custom on-board processing commands). In computer storage, each `WORD` value occupies two memory bytes. The range from -32768 to +32767 is too restrictive for many computations, and you can quickly reach the limits that `WORD` values can represent. If this is a problem, you should consider converting your data to another data type for processing.

LONG data type

A `LONG` data type (long signed integers when developing custom on-board processing commands) has 32 bits. This extends the representable range –2,147,483,648 to +2,147,483,647. The CPU on the DAP can extend convert between `WORD` and `LONG` forms easily, if you are careful about the value range. Each `LONG` value occupies four bytes of computer Data Acquisition Processor memory. The larger data type uses more storage space slightly more computing power to process. When you need very fine and uniform precision, however, the `LONG` data type is very useful.

FLOAT data type

The DAP main processor supports floating point computations consistent with IEEE Std 754. The speed penalty for floating point representations is largely a relic of the past, because of improved hardware support for floating point computations in current CPU devices.

The `FLOAT` data type has almost the same processing efficiency as the `LONG` data types, with 24 bits of precision and 8 bits of exponential scaling. The `FLOAT` type (also called `float` when developing custom on-board processing commands) is very useful when requirements for precision are moderate but flexible scaling is important. DSP operations with large data blocks or low-level signals typically perform well with `FLOAT` data, able to retain information that might otherwise be lost to truncation noise. This is a very versatile number representation, but be aware that rounding can occur. If you see a result 99999.96 where a value of 100000.00 was expected, do not be surprised.

DOUBLE data type

When very large data blocks must be analyzed, where information can be lost to rounding errors, where values range from very small to very large, or where there are many intermediate results before computing a final value, the `DOUBLE` data type (also called `double` when developing custom on-board processing commands) can be very useful. It provides 12 bits for exponential scaling and 52 bits of precision, both far in excess of anything you will normally require for data acquisition data processing. There is a slight speed penalty compared to the `FLOAT` type, but this is mostly because of the larger element size: each value requires 8 memory bytes. `FLOAT` type is fine for most applications, but in special cases where you must avoid accumulating numerical errors, `DOUBLE` will give you the best quality results.

Complex data types

DAPL does not directly support a complex data type. For processing such as the FFT that accepts or generates complex-valued results, the complex values are represented by pairs of ordinary numbers. The numbers may be in any of the data types, but both parts must be in the same data type. When complex values are represented in a polar form (magnitude and angle) the angle is represented by a binary fraction in integer types, or in the ordinary numerical range with floating point types.

Conversions Between Data Types

You can use a DAPL expression task to convert values between data types. You must be careful, however, to avoid assigning values that cannot be represented exactly, or that cannot be represented at all.

You can convert `WORD` values to `LONG` values for computation in the higher precision 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.”

A `LONG` value can be reduced in precision to a `WORD` value without losing any information provided that the value is not out of the range that 16 bits can represent. To be valid, all of the upper 16 bits must match the bit in the sign bit position of the lower 16 bits. If you try to make an invalid conversion, DAPL processing commands will convert values that are out of range to the nearest value that the 16-bit `WORD` can represent correctly. This range-limiting operation is called “saturation.” Processing commands provided by the DAPL system are careful to saturate their result values, but this is not the default behavior of the arithmetic operations provided by the CPU, so beware of this when preparing custom processing commands that work with a mix of data types.

You can perform conversions between data types using a DAPL expression task.

```
    FLOATPIPE = WORDPIPE
```

Converting to Engineering Units

Voltage readings are often scaled to the largest possible range that remains within converter limits. This produces the best measurement resolution and a good signal to noise ratio. A DAPL expression can be used to convert the digitized values so that they correspond to the physical units of the measured physical process. If integer values are inconvenient, floating point values are useful for the scaled values. For example, if the A/D converter channel 0 measure the range from 0 to 100 degrees centigrade, you can convert the binary range from 0 to 32768 into degrees Fahrenheit using the DAPL expression

```
    DEGF = (IPIPE0/32768.0)*212.0 + 32.0
```

9. Sampling and Updating

While sampling and updating are primarily under hardware control, they are for the most part configured under software control, using features of the DAPL system.

Configuration Commands

Data Acquisition Processor boards have a configuration structure. This is in contrast to almost all other data acquisition devices, which have no such structure. For the other devices, what you get is what you do, process-defined, and has no existence until programmed code operates device components through driver interfaces.

With the DAPL system, you do not need to address any registers or operate any devices. The DAPL system will do all of those things for you. But you still must tell the DAPL system about your configuration requirements, so in a sense this just shifts the location of the problem. The difference is that the configuration takes on a separate existence, a human-readable form that can be validated independently of the rest of the application. This is consistent for all Data Acquisition Processor devices.

The DAPL system partitions its configurations into three functional parts. In addition to these, you can declare shared elements, set global modes and define interconnections. Details about commands discussed in this chapter are available in the Commands Reference chapter.

Input Sampling Configurations

Input sampling is configured by an input sampling procedure, so named for historical reasons. A procedure begins with an `IDEFINE` command that assigns a name, and it terminates with an `END` command.

```
IDEFINE   AssignedName
   // Declaration commands here
END
```

Input signals are sampled by a single multiplexed sampling device or by a set of simultaneously operated sampling devices. Depending on the kind of hardware your Data Acquisition Processor has, you will specify the number of channels or the number of channel groups for your configuration. For a Data Acquisition Processor model with a single multiplexed converter, use a `CHANNELS` command. For a model with multiple parallel converters, use a `GROUPS` command.

```
IDEFINE   AssignedName
   CHANNELS  24
   // Channel declarations here
END
```

For some models, the channel groups are software configurable and you can adjust the size using a `GROUPSIZE` command.

```
IDEFINE   AssignedName
   GROUPS  3
   GROUPSIZE  8
   // Channel declaration commands begin here
END
```

When it runs, the sampling procedure will capture data and place it into a data stream called an input channel pipe. But to do that, you need to specify which hardware signals are used to fill each location in the input channel pipe. You are not restricted to using hardware channels in any particular order, and you are not restricted to using them only once, opening possibilities for adjusting effective data rates by sampling the same hardware signal multiple times. Each channel is assigned a hardware signal pin using a `SET` command.

```
SET  IPIPE0    D1    // First channel, differential pin 1
```

The name of the input channel pipe is `IPIPE`, and the channels in that pipe have the names `IPIPE0`, `IPIPE1`, etc. The channels in the input channel pipe can be listed in any order, but when sampling begins they are processed in the order of the `IPIPE` index numbers. When signals are sampled in specific hardware groups, they have a corresponding group of logical channels that receive values.

```
SET  IPIPE(8..15)   SPG0    // Captures pin group 0
```

The pin designations such as `D1` and `SPG0` in the examples above specify hardware connections. Whether the signal goes directly to the Data Acquisition Processor connector or through one of the expansion boards will depend on your hardware connections. Most DAP models support single-ended analog inputs, which have pin designations beginning with the letter `S` (single-ended) for a single channel, or `SPG` (single-ended pin group) for grouped channels. Some models support differential analog inputs and use the pin designations with the letter `D` (differential).

Some models can capture bit patterns from digital ports with pin designators B (for binary port, to distinguish from the D for differential analog). The differential input configurations use the same hardware pins as the single-ended configurations, so you will want to be careful when wiring combinations of differential and single-ended signals. Unused channels can be ignored, but for completeness, it is good to assign them to signal ground G.

```
IDEFINE   AssignedName
   CHANNELS    6
   SET  IP0    S0
   SET  IP1    S1
   SET  IP2    D3
   SET  IP3    S0
   SET  IP4    G
   SET  IP5    G
   // Operational configuration commands begin here
END
```

It is reasonable to have a variety of inputs wired to your termination boards, but you don't have to use them all. You can select just the signals that you need for taking the current set of measurements.

Signals captured as groups have corresponding grouped positions in the input channel pipe, and grouped hardware pins, each with special designations. In the following example, the first eight locations in the input channel pipe are reserved for a group of eight single-ended analog signal samples.

```
IDEFINE   AssignedName
   GROUPS    1
   SET  IP(0..7)    SPG0
   // Other configuration commands begin here
END
```

After the signals for your current application are assigned positions in the input channel pipe, you can declare additional properties of the sampling configuration. The most important of these is the TIME command, used to define the sample-to-sample time interval in microseconds. Most applications can use defaults for all of the rest.

```
IDEFINE   CompletedConfig
   GROUPS   4
   SET   IP(0..7)     SPG0
   SET   IP(8..15)    SPG1
   SET   IP(16..23)  SPG0
   SET   IP(24..31)  SPG2
   TIME  5.00
   // Special configuration options can go here
END
```

The input sampling declaration specifies a sampling cycle. The input pins are captured and digitized according to their sequence in the input channel pipe – not the order in which the configuration is listed. The time interval between samples is determined by by the sampling time. When the end of the channel list is reached, the cycle repeats. For the last example, with four groups at 5.00 microsecond intervals, the complete channel list with 32 input channels is covered in 20.00 microseconds.

Output Updating Configurations

While input processing is called *sampling*, output processing is called *updating.* Output values are continuous, but adjusted to new output levels at periodic intervals. Output updating configurations are similar to input sampling configurations in many ways. It is easiest to start with a complete configuration and point out the differences.

```
ODEFINE   OutputConfig
   CHANNELS   4
   SET   OPIPE0     A0
   SET   OPIPE1     A1
   SET   OPIPE2     A0
   SET   OPIPE3     B0
   TIME  5.00
   // Special configuration options can go here
END
```

An output procedure is introduced by an `ODEFINE` command. It does not at this time support grouped signal pins, so the channel assignments are always single. Instead of an input channel pipe, there is an output channel pipe `OPIPE`, and the channel assignments describe the locations `OPIPE0`, `OPIPE1`, etc. in the output channel pipe. There currently is not an analog differential output option, so all the analog output pin designations begin with “`A` for *Analog*.” The digital output ports remain “`B` for *Binary*”. And finally, the `TIME` command indicates the time interval between in microsecond between each update operation.

An output configuration must be provided a signal source for each location in the output channel pipe. Similar to the input channel pipes, updating is performed in the order of the channel pipe locations, not the order in which they are declared.

Reader and Writer Constraints

After the input procedure configures the input sources and an output procedure configures the output destinations for data, processing commands define the connections. Processing is defined in one or more processing procedures, beginning with a `PDEFINE` command and ending with an `END` command.

```
PDEFINE   ProcessName
// Define processing tasks here
END
```

All processing in the DAPL system is multi-tasking. That means there is no way to predict the exact order of execution of calculations. If exact sequencing of calculations is critical, they must be packaged as a single monolithic command that embodies all data processing. As long as all of the processing consumes input data at a consistent rate, and supplies data for outputs at the rate needed, the processing can proceed continuously.

Data will remain in memory until all tasks that need to process the data have done so. If for some reason a task is unable to continue, a backlog of data will accumulate in memory.

When a processing configuration is started, it could receive data from the input channel pipe, and it could send data to the output channel pipe. With multi-tasking, it can become quite confusing trying to figure out which task consumes which data from the input channel pipe, and which task provides data to the output channel pipe. To help avoid this confusion, the DAPL system enforces the following rules for reading data from pipes.

1. Multiple tasks can read from data sources, including the input channel pipe.

2. Tasks can read any set of channels from the input channel pipe. A subset of channels is indicated using the `IPIPE` notation. For example, to process channels 0 through 3 and 8 through 11, specify the following pipe name.
   ```
   IPIPE(0..3,8..11)
   ```
   A strict ascending sequence is not mandatory, but processing is most efficient this way because the DAPL system does not need to shuffle the data before sending it to the processing task.

3. Every task that specifies one of the channels will receive a copy of ALL data from that channel.

4. Only tasks in one start group can read data from the same pipe. It is possible to partially circumvent this limitation by copying the data to a separate pipe for transfer to a task in another start group.

When a processing task receives data from multiple channels, the data are organized in a multiplexed fashion, one sample from each channel in the specified channel list, and then this pattern repeats.

Because every task receives its own copy of every sample from every channel it reads, this presents a very convincing illusion that every reference to `IPIPE` spawns a completely separate and independent pipe with its own separate and independent data stream. We often speak of multiple `IPIPE`s, though there is actually only one underlying data stream containing all of the sampled input data.

Sending data between tasks also has an important constraint.

1. Only one task can write to a destination pipe.

References to channels in an output channel pipe act somewhat like references to separate output pipes, so at least in theory you could have more than one. There are many ways to get into trouble with this, however. For example, suppose a single task produces two output sequences, `OUTA` and `OUTB`. The configuration assigns `OP4` to the task's `OUTA` parameter, and `OP3` to the task's `OUTB` parameter. This should deliver the two data streams to the output procedure.

- The processing task performs a write operation to `OUTA`, resulting in data transfer to the output procedure for channel `OP4`. This operation must completed before the processing task can continue and send the data to `OUTB`.

- The output procedure does not have data for channel `OP3`, which arrives via `OUTB`. It cannot use the data from `OP4` and cannot complete the data transfer, so it waits for the channel 3 data to arrive. But because the processing task is already waiting, the needed data will never be sent. The two tasks are deadlocked.

One simple way to avoid this kind of problem is to collect output values and send them to output updating as a single task, in the right sequence.

```
MERGE ( OUTB, OUTA, IFLAG, PIPE4,   OPIPE(0..3) )
```

Starting and Stopping

Configurations can be downloaded to the Data Acquisition Processor board at any time. When the host application is ready to run, it sends a `START` command.

```
START
```

When the application is completed, it sends the `STOP` command.

```
STOP
```

When there is no further use for the configuration, it clears the DAP memory using a `RESET` command.

```
RESET
```

Advanced applications can download multiple configurations and select them by name. For most applications, however, the easiest way to manage configurations is to `RESET` the old configuration and download a new one. A conventional practice, and a good one, is to begin each new configuration with a `RESET` command to clear out any configuration that might have been downloaded to the Data Acquisition Processor previously.

When a configuration is started, the DAPL system will go through the following sequence to locate the procedures to run.

1. If an output procedure is specified by the `START` command, start it. If no output procedure is specified by the `START` command, look for one, and start the first one found.

2. If an input procedure is specified by the `START` command, start it. If no input procedure is specified by the `START` command, look for one, and start the first one found.

3. Start all specified processing procedures. If no processing procedures are specified, look for all configured processing procedures and start them.

You can override this start-up sequence for special situations, but the default sequence is the one that is least likely to result in problems, particularly when using some of the more advanced hardware options.

Output updating can do very little as it starts, because data sources are not yet activated. Output processing suspends until the rest of the start-up sequence provides the data connections and provides some data to send.

Starting the input processing is also relatively fast. Data arrive from external sources so hardware sampling cycles begin almost immediately. The connections have not yet been established between the input, processing, and output, however, so some data must be temporarily buffered in memory until the rest of the start-up initialization completes.

As processing configurations are started, the pipe connections are completed between data sources and to output destinations. Once all of these connections are established for a processing task, it becomes available to be scheduled for execution. When all tasks are running and all connections between data sources and destinations are consistent, the configuration can proceed to operate continuously.

Starting under hardware control

Data Acquisition Processors have hardware features to allow the starting sequence to be controlled by an external digital signal. These features are known as *hardware triggering*.

To enable hardware triggering for input sampling, place an HTRIGGER command in the input procedure definition. To enable hardware triggering for output updating, place an HTRIGGER command in the output procedure definition. For example, an output procedure configured with hardware triggering might look like the following.

```
ODEFINE   OutputConfig
   CHANNELS  1
   SET   OPIPE0     A0
   TIME  5.00
   HTRIGGER    ONESHOT
END
```

The START command is still required: it determines which configurations to run. The startup sequence is basically the same as before, but input sampling suspends at the place where the hardware capture of samples or the hardware updating of output signals would ordinarily begin. Activity begins quickly when the controlling hardware trigger input is received.

The HTRIGGER command can specify a ONESHOT or GATED mode, for activity controlled by a logic transition or by a logic level on the input line.

Stopping under clock control

Sometimes data collection or signal generation is required for only a limited time. For example, in a crash test, after a few milliseconds of intense action, there is nothing left to measure. Or for output signal generation, after ramping the signal through the full range, the test is done. These configurations can finish after a known period of time, which corresponds to a known number of samples or updates. To specify an automatic shutdown, include a COUNT command in the input or output configuration. For example, the following input configuration stops automatically after collecting 10000 measurements for each of two channels.

```
IDEFINE   Collect10000
   CHANNELS  2
   SET   IPIPE0     D0
   SET   IPIPE1     D1
   TIME  40.0
   COUNT   20000
END
```

Involuntary stopping: Underflow and Overflow

Updating and sampling are under the control of a programmable state machine in the DAP system – that is to say, it is a hardware process. The data sampling process expects that it will have a place to store its data. If it does not, this is called an *overflow condition*, and the DAPL system has three alternatives.

1. Fail quietly, overwriting existing data. Most simple acquisition devices will do this. It is forbidden on a DAP system. *DAP systems will never lose data, and never corrupt existing data.*

2. Generate a fault. Unfortunately, it is not in general possible for the DAPL system to know what it should do in response to the fault. *Unprocessed data could be lost, and this is unacceptable in a DAP system.*

3. The DAPL system stops the sampling process, logs a warning, and leaves all other processes running. Other processing can continue so that any samples previously captured are processed normally, until all are processed, and then the processing must suspend.

Memory overflow conditions typically result when trying to perform too many calculations on too many channels, or when trying to transfer more samples to the host than its data bus can handle.

Similar hazards occur with output updating. The updating process expects that it will have an update value ready when its output clocking pulse arrives. If it does not, this is called an *underflow condition*, and the DAP hardware again has only a few alternatives.

1. Fail quietly, placing garbage data on the output port. This is forbidden on a DAP system. *DAP systems will never send undefined update values to an output port.*

2. Generate a fault. Unfortunately, it is not in general possible for the DAPL system to know what it should do to correct the fault.

3. The DAPL system stops the updating process, logs a warning, and leaves all other processes running. Some kind of supervisory follow-up action should be taken to shut down other processing in an orderly fashion before a backlog of unprocessed data forces other tasks to suspend.

Underflow conditions are typically the result of overloading the DAP board's processor – so that the tasks that are supposed to generate the output data do not get sufficient opportunity to run – or providing an incorrect number of update values for the output updating to use.

Premature underflow and OUTPUTWAIT

Because of the uncertainties about the execution order of independent processing tasks, it is uncertain exactly when data will be available for the output updating to use. Suppose that data for output are generated in blocks. Time is required to collect the data for processing a block. Even if calculations are done at a sufficient rate, if a data block arrives too late, the output updating will underflow.

Hardware-based processing cannot tolerate the delays. It must have a sufficient amount of buffered data to weather any short-term delays. The most dangerous time for output underflow is during application start up. Several tasks start in quick succession, each of them establishing pipe connections and performing other setup operations. It is very easy for a task to compute a few values, enough to get the output updating started, but not enough to keep it going while other tasks start.

To avoid these transient underflow events and premature shutdown, an output procedure will not begin delivering output updates until it has in its buffers sufficient data to sustain 1/10 second of activity. The `OUTPUTWAIT` command can be used to adjust this behavior, but most applications should use the default to avoid a premature underflow condition. As a side effect, output updating is always subject to a time delay for cycling through all of the data previously sent for output processing.

Burst mode

There are some situations in which input or output processing is intermittent. For example, input data collection might be initiated by an event such as a “top dead center” pulse for engine ignition. After some period of time, data collection can stop. This processing can occur several times, and it is not convenient to restart the configuration for each firing event. For output processing, a sonar “ping” might require an intense burst of output activity, but then no action until an external activation signal prompts for the next tone burst. In applications such as these, it is convenient to let the input or output procedures stop and then re-initialize themselves, so that the next triggering pulse is the same as starting over from the beginning, but with very low overhead. This kind of intermittent processing is called *burst mode*.

To configure burst-mode processing for an input procedure, include the `HTRIGGER ONESHOT` commands to enable starting under hardware trigger control, the `SAMPLE BURST` command to enable the automatic resets, and a `COUNT` to terminate sampling after the data are collected.

```
IDEFINE   CollectBurst
   CHANNELS  2
   SET   IPIPE0     D0
   SET   IPIPE1     D1
   TIME  2.50
   HTRIGGER  ONESHOT
   SAMPLE      BURST
   COUNT       40000
END
```

A burst mode configuration for output updating is similar, but easier. Use the `UPDATE BURST` command.You don’t need to specify the `COUNT` to stop the updating if you provide a block of data of sufficient size and just let the updating continue until the data are exhausted. This underflow is expected and does not produce warnings. You also don’t need the `HTRIGGER` command if you supply data in a single long block and let the `OUTPUTWAIT` condition initiate the sampling.

```
ODEFINE   SendPulse
   CHANNELS  1
   SET   OPIPE0     A0
   TIME  4.80
   UPDATE   BURST
   OUTPUTWAIT  1000
END
```

Using Software Triggering

Hardware triggering for input sampling is very useful when there is a device already present in the measurement system to generate timing pulse at just the right time. But hardware triggering has limitations.

- *Complexity.* If you don't have the necessary timing pulse, getting one can be a problem.
- *Cost*. Avoiding unnecessary control hardware can mean a considerable cost savings.
- *Test criteria.* For example, how would you fire a hardware logic gate whenever a signal of sufficient strength is present in the band from 220 to 260 Hz?
- *Pre-event analysis*. Hardware triggering suspends sampling activity until an event arrives. If you need to analyze data prior to the event, you won't have it.
- *Controllable duration*. After data collection starts, you have few hardware options for how to stop it.

You can avoid some of the costs and limitations of hardware-only triggering, provided that you have sufficient processing power – which is the normal case with Data Acquisition Processor boards. DAP boards can measure and discard data with extreme efficiency. When something interesting is detected in the data, that part of the data can be kept. This strategy is called *software triggering.*

To use software triggering, start your configuration for continuous operation without any special hardware option. In your processing procedure, you direct the signal where events can be detected into a trigger generating (trigger writing) task such as `LIMIT`. For the example of detecting a 220 to 260 Hz tone, run the signal through a digtal bandpass filter, and then allow `LIMIT` to look for high peaks. When the trigger writer detects an event condition, it posts that information to a `TRIGGER` element, which is very much like a data pipe except that it is dedicated to processing event information. A responding task (trigger reader) such as `WAIT` will extract the relevant data from any number of data channels and route them for further processing.

Internal and External Clocking

After you have a configuration set up and started, it can begin to collect samples and issue outputs. But there is sometimes a question about exactly when these actions should occur. One common special case is rotating equipment, instrumented with a high-resolution optical encoder that produces logic pulses at accurate angular positions for each rotation. At a fixed speed, encoder pulses and fixed digital timer pulses are equivalent. But when the speed is not fixed, samples captured on a fixed timer would no longer represent equal angular positions.

Data Acquisition Processor boards give you options: data capture timed by an external logic signal, data capture with the onboard precision clock, or a combination.

Channel List Clocking

The `CLCLOCKING ON` command tells the DAPL system to capture samples or update its output ports using the on-board precision clock, at time intervals configured by the `TIME` command. If not specified, `CLCLOCKING ON` is assumed by default.

```
ODEFINE   TrackEncoder
   CHANNELS  2
   SET   OPIPE0     B1
   SET   OPIPE0     B2
   TIME  28.8
   CLCLOCKING  ON
END
```

External clock

Use `CLCLOCKING OFF` when an external clock signal rather than the onboard digital timer determines the sampling or updating instants.

With external clocking, each sampling or updating instant is determined by an external logic signal, such as an encoder pulse. To obtain this behavior, place a `CLOCK EXTERNAL` command into your input or output configuration. You also need to override the default timing behavior with a `CLCLOCKING OFF` command.

```
IDEFINE   ClockedSampling
   CHANNELS  1
   SET   IPIPE0     S10
   TIME  20.0
   CLOCK    EXTERNAL
   CLCLOCKING  OFF
END
```

Using the hardware external clocking does not affect various other features such as triggering. The `TIME` command does not determine the sample-by-sample interval, but it informs the DAPL system of approximately the time interval to expect, to assist the system in making good decisions about internal memory allocations and data buffering.

A combination

Data Acquisition Processors allow a unique combination of channel list clocking in combination with external clocking. When there are several signals to be measured, the external clock signal can indicate when to sample the first one, and then the internal clock signal can be used to capture measurements in rapid succession for the rest of the channels in the procedure's channel list.

```
IDEFINE   Measure4Chan
   CHANNELS  4
   SET   IPIPE0     D0
   SET   IPIPE1     D1
   SET   IPIPE2     D2
   SET   IPIPE3     D3
   TIME  2.5
   CLOCK   EXTERNAL
   CLCLOCKING  ON
END
```

Pipelining and Burst Mode

Pipelining is a widely used technique for increasing data capture rates. With it, the input signals are allowed to settle, while simultaneously conversion is underway for another sample and a third sample result is transported across a digital data bus. The processing of these operations, performed at the same time by coordinated electronic elements, is called *pipelined processing*. The net conversion rate is faster, but the side effect is that the value produced on a given clock pulse is not really the current sample! It is really the result of the most recently completed conversion. The real “current sample” is still in electronic form within the conversion pipeline.

When using a burst mode with *onboard clocking*, the DAPL system knows about its pipeline, and it knows about its clock. It can use extra clock pulses to clear out extraneous old history from its pipeline, so that the first value you get out will be the first value actually measured in the current burst. You don’t need to think about it.

With external clocking, however, the DAPL system does not have the same degree of hardware control. The number of samples you get will be the number of samples you clock out. The first one or two samples you clock out could be old history carried forward in the pipeline from a previous burst, far in the past. In rare cases, these arbitrarily delayed samples might be meaningful despite the delay in receiving them. Most of the time, however, the easiest thing to do is collect a few extra samples and discard the extraneous samples from the beginning of the block. You can test your configuration to determine whether pipelining has produced extraneous samples. Or check your Data Acquisition Processor hardware manual for information about how many extraneous samples will be produced from the sampling pipeline in burst mode.

This is not a DAP design problem – it is a universal problem for analog-to-digital converter devices with pipelined input stages. The people who work with converter devices regularly will expect to see this. The rest of us have to be careful.

Fast sampling

The conversion pipeline sometimes provides benefits besides a general boost in throughput. Data Acquisition Processor fast sampling is one of these extra advantages. If you are reading a mix of digital and analog signals, the digital signals need no conversion and hence no pipeline, but they do need settling time. The DAPL system is aware of this. If you have a mix of digital and analog inputs, DAPL allows the analog hardware settling time to overlap the digital hardware settling time. The net effect is that the combination can be allowed to operate at a faster sampling rate than would normally be possible.

Buffering and Latency

Data Acquisition Processor board capacity and the delays in transporting data through the DAPL system are greatly influenced by the degree to which sample values are collected together before processing. The Data Acquisition Processor architecture is designed primarily for acquiring large amounts of data on many channels.

Hardware buffering

To operate conversion devices with accurate timing and without losing any data requires an extraordinary amount of attention from a control processor. All data acquisition devices face problems with hardware control, with different kinds of solutions.

1. *Custom-designed DSP processor solution.* For those who have the time and expertise to develop it, a custom-engineered DSP processing board dedicated to converter device operation will provide the ultimate in performance at the ultimate in costs. The software can be optimized to meet stringent timing requirements – for one application.

2. *Simple hardware buffering solution.* For lowest costs, simple acquisition boards depend on their host processors for almost everything. Workstation systems have their own agenda (an enhanced multimedia experience), and most processing resources are dedicated to those purposes. Because attention from the host is unpredictable and subject to delays, the data must be collected by the hardware into relatively large hardware memory buffers and held there until the host is available. If the buffers are long enough, the probability of losing data is small, but application software in the host must be very careful about process timing.

3. *Intelligent buffering solution.* An intermediate between the two cases above, Data Acquisition Processor systems include hardware-level buffering but also dedicated onboard processing. Thanks to the guaranteed availability of processing, there is less hardware buffering delay. While faster and much more flexible than the elementary hardware buffers of simple acquisition devices, any hardware buffering at all impedes value-by-value processing, so performance cannot match an optimized custom DSP design.

Whether one value or 100 values are obtained from the hardware buffer, the overhead for accessing the buffering hardware is about the same. If a Data Acquisition Processor can meet timing constraints to access one channel, it can probably meet the same timing constraints on 20 or 30 channels operating in parallel.

Pipe system buffering

As the DAPL system removes samples from the input buffer hardware, it moves them into the onboard main processor memory. Sample values are organized by the pipe system, with memory allocated and values indexed so that they can be delivered upon request when an application needs them. Buffering can store vast amounts of data, or very little, depending on the demands of the application.

For moving single values through quickly, the memory allocation and indexing overhead of the pipe system is small, not much more than operation of hardware devices. However, a small amount of processing overhead remains. This is enough to constrain the rates at which the DAPL system can produce individual responses.

Tasking effects

The DAPL system depends on processing tasks to move data. If there is no data analysis onboard, at a minimum the DAPL system must copy input samples to the host, and this takes one processing task.

Each processing task acts like an additional buffering delay. A task will convert its input data stream to produce a new data stream of results, which will use additional pipe system memory. Even if the tasks perform no analysis, the operations to move data into and out of pipes will cause some delays. If there are multiple processing tasks, the execution order of tasks is indeterminate, and this results in corresponding indeterminate time delays before the tasks deliver results.

Buffering delays are influenced by the task scheduling interval. If tasks are allowed to run for a longer time, they tend to collect more data from input sources and deliver results in larger groups, with better processing efficiency but larger buffering delays. To give the tasks less time to collect and buffer data, you can reduce the task scheduling time quantum using the command

```
   OPTIONS   QUANTUM=<interval>
```

where the `<interval>` value can be as small as 200 microseconds.

Having said all of this, the *worst*-case time delays from sampling all the way through to delivery of results will typically be better than host workstation delays at their *best*.

10. Data Transfer

Communication pipes (com pipes) are first-in-first-out buffers for communication between a Data Acquisition Processor and its host. Com pipes allow text or binary communication, though binary is strongly recommended for transferring bulk data.

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 host. 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 host. They are not pre-configured to associate with any particular tasks.

Sending Text to the PC

The `FORMAT` command can be used to encode and send text data to the PC. The encoding process is not efficient. Though text displays are very useful during test and development, to get a *quick peek* at data to verify validity, binary data transfers are better for almost all other purposes.

In addition to streamed data from pipes, a `FORMAT` task can send data from variables or 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 host, simply by streaming data into the `$BinOut` communication pipe. The simplest way to send binary data is the `COPY` command.

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, from sources with similar or different data types, a `MERGE` task can be used to combine and deliver the data. If data enter the pipes at different rates, the command `MERGEF` can be used to tag the data with source identifier codes prior to transfer. If data enter the pipes at different but proportional rates, the command `NMERGE` can be used.

The commands `BMERGE` and `BMERGEF` are more efficient blocked versions of `MERGE` and `MERGEF` for processing bulk data. A `BMERGE` or `BMERGEF` task reads blocks of data from multiple pipes and writes the blocks to the output pipe. The output pipe usually is `$BinOut`. `BMERGEF` adds an identifying flag to each block of data, something like `MERGEF`, but much more efficient.

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.

Custom processing commands that need to receive special instructions from the host in text form can do this by setting up a separate communication pipe, and configuring it to transfer text.

Reading Binary Data from the PC

Binary data can be sent from the PC to the Data Acquisition Processor through the communication 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 communication pipes can be defined. The Data Acquisition Processor can support several binary and text communication pipes transmitting and receiving simultaneously. The additional pipes are configured in the DAPcell control panel application. Communication pipes can also be configured by software applications, using features of the DAPIO32 programming interface.

11. Resource Allocation

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

Processing Time Allocation

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 one of the following reasons:

1. a task used up its maximum CPU time allocation, and another task must be allowed to run;
2. a task has no data to process and explicitly requests a task switch;
3. a system operation supplied data that a higher priority task needed to continue its processing;
4. a system timing operation allowed continuing a task that was previously awaiting the timing event.

More information about task priorities and scheduling is provided in the chapter on Prioritized Multitasking.

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.

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

For information about optimizing processor performance see Chapter 12.

Output Channel Access

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 delivers the combined text data, or how many lines each `FORMAT` statement will send at each opportunity. 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 available in the RAM to satisfy the request.

Command modules will allocate a certain amount of memory to define a shared code area and command management structures at the time a command module is loaded. This is a one-time allocation and it is not released until the command module is unloaded by external utility or by an ERASE command.

When a task is defined invoking one of the available processing commands, and that task is started, it will receive an allocation of memory from a special region of memory called heap memory. The allocated memory will provide a stack area for function and subroutine calls and local memory. It will also contain its own copy of all “global” variable values, so that multiple tasks using the same type of command will not interfere with each other’s internal variables. After starting, a task may request additional memory blocks to be used for the lifetime of the task. To determine the total amount of available heap memory and the amount of memory available for data storage, start the processing configuration and run the command DISPLAY HMEM.

The amount of memory available for data storage can change as a configuration runs, so the displayed value might not be exact. Pipes can temporarily allocate memory blocks for data transferred through pipes. Processing tasks can also create and free dynamic objects at any time during execution. When processing is terminated, these allocations are released automatically.

If there is insufficient free memory to satisfy a request for a new memory allocation, this produces a *memory overflow condition*. The task will be terminated, and an error message will be issued. Termination of a task usually blocks execution of other tasks, leading to a complete shutdown of processing.

12. Optimizing Processor Performance

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

Reducing Processor Load

DAPL 3000-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 uses a small amount of time for each operation that moves a value into a pipe or takes a value out of a pipe. Processing data in large groups tends to be more efficient than processing the same amount of data in small groups.

Processing more data generally takes longer than processing less. 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 Host and then process and reduce the data there. Even if the PC host has a very powerful processor, that processor is mostly occupied with operating graphics, keyboard, pointing device, disk, networking, and other functions besides processing your data.

Digital Signal Processing

Digital signal processing is possible with all Data Acquisition Processor models, but keep in mind the limitations of the processor. `FIRFILTER`, `FFT`, and other digital signal processing tasks are highly optimized, but computationally intensive. For fastest processing, 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 filter taps required, as well as reducing the number of data to process. When computing FFTs, try to compute only the number of blocks and frequency resolution you need. 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 transfers of bulk data from the Data Acquisition Processor to the PC, use binary format. This eliminates the time-consuming format conversion from binary to text. 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 using all channels
2. task with a single channel pipe
3. task with a channel pipe list of some, but not all, channel pipes

Streaming Data to the PC

Some applications require high-speed data acquisition without real-time data processing. The `COPY` command is a good choice when all data streams are the same type. Even better is the `BMERGE` command when processed data can be moved in larger blocks. The `MERGE` command can transfer a mix of data types, but the packing of data is less efficient. Binary transfers stream data to the host in binary format. The most efficient transfers use a single source pipe (such as an input channel pipe with channel list). Microstar Laboratories DAPstudio software can log data to a disk or RAM disk at high speed in real-time without any additional software.

Trigger Performance

Tasks that involve triggers are 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

A 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.

13. Prioritized Multitasking

The DAPL operating system provides some options to control task execution priority. Tasks at higher priority can respond quickly to isolated events in real time. Tasks for bulk data processing, not constrained by real-time deadlines, can operate at a lower priority without interfering with the higher-priority tasks.

Priority Levels in DAPL

The DAPL system supports multiple levels of task priority, with multiple tasks at each priority level. Task scheduling is easier theoretically when each task has its own unique priority level, but the DAPL system must also ensure that data streams flow uniformly and efficiently. Usually, this is easier when tasks can have approximately equal opportunities to execute.

As an example of prioritized processing, consider an application that controls tool contact pressure for a drilling machine tool. To maintain the correct cutting pressure, a strain-sensing device monitors the contact pressure and controls the tool position. The desired cutting pressure is different for a freshly sharpened cutting tool or a more worn one. To determine whether the cutter is sharp or dull, vibrations are monitored with a microphone and an FFT spectrum analyzes frequencies that correlate with dull tools. While tool pressure corrections need to be regular and fast, the FFT data analysis collects large blocks and posts results slowly. You don't want the low-speed analysis to consume all of the processor capacity and interfere with high-speed control response.

Assigning task priorities reduces exposure to delays by reducing the number of tasks that can cause delays. With priorities in effect, tasks at higher priority levels proceed as if the lower priority processing didn't exist. Response delays include:

- scheduling delays caused by any higher priority tasks (if any),
- scheduling delays for other tasks at the same priority,
- a small delay to account for everything else.

DAPL Tasking States

Before discussing priorities further, it is necessary to describe the states of tasks that are able or not able to run.

Every DAPL processing task exists in one of the following states.

- CREATED. The task has been defined in a processing configuration, but the configuration is not started. The task has no system resources and no connections to other tasks.
- RUNNING. The task is started, has established connections to other tasks, and has allocated system resources. It is available to be scheduled for execution.
- BLOCKED. The task would be in a RUNNING state, except a requested resource is not available; for example, it needs data from its input stream. A BLOCKED task is not available to be scheduled for execution.
- SLEEPING. This is very similar to BLOCKED except that the reason the task cannot be scheduled for execution is not a resource conflict. The most common reason is awaiting a timeout interval.
- DEAD. The task has completed its processing and is no longer available for execution.

A RUNNING task is a candidate for execution, but only one task actually runs on the hardware processor at any one instant. Selection of this task is the responsibility of the *task scheduler.*

Tasks enter the RUNNING state when their processing configuration is started. Tasks in the RUNNING state remain schedulable until some action causes them to change to a BLOCKED, SLEEPING, or DEAD state. Some examples:

- The RUNNING task requests data from a data pipe and insufficient data are available to satisfy the request.
- The RUNNING task attempts to write data to a data pipe but insufficient capacity is available for the receiving pipe to accept all of the data.
- The RUNNING task requests a timing interval.

Tasks that are BLOCKED become schedulable again when the resource conflict is resolved. Some examples:

- A task that became BLOCKED because it needed more data becomes a RUNNING task when the writer task supplies the needed data.
- A task that became BLOCKED because a pipe could not accept more data becomes a RUNNING task again when the data reader tasks process some of the data, making pipe capacity available.

Tasks that are SLEEPING become schedulable again when the timing interval ends.

Tasks that are DEAD can never be rescheduled. They revert to a CREATED state when the entire processing configuration is stopped.

Priority Scheduling

The *task scheduler* selects tasks for execution from the list of available running tasks. The rules it applies for making the selection are the *scheduling strategy*.

The Round Robin Strategy

If all processing tasks have the same priority — the default configuration when no priority is explicitly assigned — the scheduling strategy is "round robin" with every task getting an opportunity to execute in turn. This strategy was used in earlier versions of the DAPL operations system.

However, the DAPL system applies an additional heuristic to help optimize data flow. If a task in the round robin sequence is BLOCKED so that it cannot execute, and if the DAPL system can identify the task responsible for the resource conflict, the DAPL system can schedule that other task to execute in place of the BLOCKED task — at least, up to the point that the resource conflict is resolved.

Scheduling Rules

When priorities are assigned, some tasks will run in preference to other tasks. The task scheduling rules for the priority scheduling are:

1. To be schedulable for execution, a task must either be RUNNING, or BLOCKED by another known RUNNING task that provides the resource needed by that BLOCKED task.

2. A task selected for execution must have the highest priority of all tasks that are schedulable. If multiple tasks satisfy this condition, tasks are selected for execution from this group in a fixed "round robin" sequence.

3. If the schedulable task selected for execution is RUNNING, it immediately becomes the currently executing task.

4. If the schedulable task selected for execution is BLOCKED, the RUNNING task that blocks it can execute in its place.

5. If an event or action causes one or more SLEEPING or BLOCKED tasks of higher priority to become RUNNING tasks, the scheduling rules are re-evaluated and one of the higher priority tasks is selected for execution.

6. If the currently executing task is running in place of another task, and resolves the resource conflict between the two tasks, the current task is suspended and rule (5) above is evaluated so that the other task may become the currently executing task.

Priority Inversions and Their Resolution

According to scheduling rule 4, it is possible that a lower-priority task can execute in place of a higher priority task temporarily when there is a resource conflict. Suppose that this is not the case, and a high priority task becomes BLOCKED awaiting data from a low priority task. Even if enough data become available for the low priority task to continue, other intervening low or medium priority tasks could be scheduled first, causing a long delay.

A situation in which a high-priority task waits while various other tasks of lower priority execute is called a *priority inversion*. Priority inversions can seriously degrade the response time of a high-priority task. Though priority inversions can't be avoided entirely, they should be resolved quickly. Executing tasks in place of other tasks is a kind of *priority inheritance* scheme. The low priority tasks temporarily execute at higher priority, at least long enough to resolve the resource conflict. At that point, the low priority task reverts to its usual priority and its usual place in the low priority round-robin list, while the high priority task assumes a RUNNING state and is schedulable at its priority.

Configuring Priorities

A priority is assigned to one or more tasks configured within a PDEFINE / END command group. Priorities range from 1 (the lowest) to 4 (the highest). All commands in the group are assigned the same priority level, hence, tasks that operate at different priorities will be configured in different PDEFINE / END groups. The priority is declared on the PRIORITY command, which appears (if present) immediately after the PDEFINE command. If no PRIORITY command is present, priority 2 is assumed as the default.

For example:

```
PDEFINE  LOWGROUP
  PRIORITY 2
  … normal priority task definition commands here …
  END

PDEFINE  HIGHGROUP
  PRIORITY 3
  … high priority task definition commands here …
  END
```

A START command will start both of these processing groups, unless you specify a list of procedures that deliberately omits one or the other. To avoid pipe access restrictions, you should start all procedures that exchange data across priority levels in one start group; that is, use the same START command.

Prioritized Processing Application

In this section, we will examine two prioritized configurations for the same control application, and contrast their relative advantages and disadvantages.

In the example application, a sensor is controlled to multiple positions, and at each position data are captured. Sensor data are captured rapidly and filtered to reduce noise. A PID command drives the sensor positioning to arrive at each sensing location quickly and hold that position accurately. Regular and fast PID updates are important for accurate and stable settling, but initiation of the position changes is not time-critical because time must be allowed anyway for the mechanical transient to settle. The position levels are sent to the PID command at a slow rate, at intervals derived from the sampling clock.

First Solution

We can observe that responding to the real-time feedback quickly involves three operations.

1. Delay high-priority processing until an input sample is reached when the PID control must be updated.

2. Compute an immediate control response in the PID task.

3. Send the control output to the analog output port using a DACOUT task.

We observe that these activities are all related to the high-speed, regular response that should not be delayed by other processing, so we can package these three processing tasks at a higher priority level.

The feedback signal is oversampled, so we need to select the samples for the feedback control to use. However, we also have the general restriction that all readers of the same pipe must be in processing procedures started at the same time. After selecting the signals needed for high priority processing, the sensor data are sent down to the lower priority processing through a separate data stream.

This must be done carefully, otherwise the DAPL system will spend most of its time spinning through a tight loop, transferring samples one at a time to the lower level processing. That is why we should not use the regular SKIP command. Instead, we can use a customized high-priority processing command. Let's call this command SKIPN. SKIPN will take care of all data selection and copying actions, remaining BLOCKED most of the time until a configurable group of N samples becomes available in each data channel. When the data block is ready, the SKIPN task resumes its RUNNING state. It copies a value of the feedback data signal for high priority processing, and copies the bulk of the sensor data for low priority processing.

First, configure the data acquisition for sampling the position feedback and sensor signals. The sensor channel and the PID feedback channel are each captured at 50000 samples per second. For brevity of presentation, the declarations of pipes, variables and named constants are skipped. Here is the input configuration:

```
IDEFINE  AllChannels
  CHANNELS    2
  SET  IP0   S0       // pid feedback channel
  SET  IP1   S1       // sensor input channel
  TIME  10             // 2 channels each at 50 kHz rate
END
```

The PID loop only requires updates at a rate of 500 updates per second, so the SKIPN task will be configured to accept 100 samples from each channel. The high priority processing is unblocked as each data block of 200 samples arrives, with 100 feedback samples and 100 sensor samples. One feedback value propagates through the PID command and out to the DACOUT task.

```
PDEFINE  HiPriority
    PRIORITY 3
    SKIPN( IP(0,1), 2, 100, PIDfb, IPecho )
    PID( PIDset, PIDfb, Pgain, Igain, Dgain, 500.0, PIDout )
    DACOUT( PIDout, Dac0 )
END
```

Now define the low-priority processing. First, define a sequence of stepped levels for the PID command to track.

```
VECTOR  vPIDlevels   WORD =  ( 1000, 2000, 3000, 4000, 5000, 6000,
    7000, 8000, 9000, 10000, 11000, 12000, 13000, 14000, 15000 )

PDEFINE  LoPriority
…
```

The commands in this lower priority processing group will generate the stepped PID command sequence. Each level is held for 5000 sensor samples (1/10 second). We can obtain an arbitrary data stream at this rate using an ordinary SKIP command to count and select sensor samples.

```
    SKIP( IPecho, 0, 1, 4999, IPslow )
```

We don't care about the values selected, so we will use a DAPL expression to replace them with PID command setpoint levels. Delays of a few milliseconds don't matter.

```
    COPYVEC( vPIDlevels, PIDlevels )
    PIDset = (0*Ipslow) + PIDlevels
```

These PID level commands are sent to the high priority processing. The PID task adjusts its target level whenever a new value is received, never missing an update, even if the new level is a little slow in arriving. In the meantime, the low-priority processing uses a FIRLOWPASS task to smooth the sensor data, decimate by a factor of 5 to reduce the sample rate to 10000 samples per second, and stream the results to the host PC.

```
    FIRLOWPASS( IPecho, 5, $BinOut )
END
```

There is nothing wrong with this solution for this application. It is successful because high priority results propagate one at a time. That means either there is one high priority task able to run, or no high priority tasks can run, and in either case, scheduling is fast and easy. This strategy works in general when processing is fast, and data dependencies are simple. As long as all of the tasks get a fair opportunity to execute, which they do, one pass will finish them all.

Second Solution

In more complex networks of related tasks, high priority processes might take multiple scheduling quanta to complete. A long *dependency chain* of related tasks possibly could delay other tasks by the length of the dependency chain times the size of the time quantum.

In the second solution, as many interdependent tasks as possible are placed into the low priority processing. Configure only DACOUT, the final task at the end of the dependency chain, for high priority. When the lower priority tasks are driving the data to DACOUT, they run in its place at high priority to resolve its BLOCKING condition, so the effect is roughly the same as if they also had high priority.

```
PDEFINE  HiPriority
    PRIORITY 3
    DACOUT( PIDout, Dac0 )
END
```

The DACOUT task needs to get data from the PID control. Because processing of the input channels is in the low-priority configuration, an ordinary SKIP command can select the data for the PID command. The SKIP task alternately takes one feedback sample then is BLOCKED as it ignores the next 99 samples. Without new data, the SKIP task waits, PID task waits, and the DACOUT task waits.

```
PDEFINE  LoPriority
    SKIP(IPipe0,  0, 1, 99,   PIDfb)
    PID( PIDset, PIDfb, Pgain, Igain, Dgain, 500.0, PIDout )
```

The slow 1/10 second intervals for the stepped PID command sequence is produced the same as before, but input channel data can be used directly to establish the timing.

```
    COPYVEC( vPIDlevels, PIDlevels )
    SKIP(Ipipe0,  0, 1, 4999,  IPslow )
    PIDset = (0*Ipslow) + PIDlevels
```

Filtering and decimation of the sensor data can also read input channel data directly.

```
    FIRLOWPASS( IPipe1, 5, $BinOut )
END
```

Comparing the Two Configurations

For this application, the performance difference is negligible for the two solutions. The tasks execute quickly, so it doesn't matter much whether they are actually configured in the high priority task list or just inherit high execution priority. More complex configurations could show many subtle differences in behavior, related to the number of task switching operations, the order of task scheduling, and the efficiency with which accumulations of data are processed.

The second configuration has some small advantages. A custom processing command is not needed to manage processing time at the higher priority. Processing can use the input channel data directly without additional transfer operations. In general, with fewer tasks in the high priority task, there are fewer ways for high priority processes to consume all of the computing power. Keeping a minimum number of tasks in the high priority group makes it easy to bound the worst-case response time: it equals the number of high priority tasks times the scheduling time quantum. The main disadvantage is that the scheduling is a little harder to analyze, because high priority task execution might occur in tasks that are not specifically assigned to the high priority group.

Rescuer Scheduling

Prioritized task configurations are tricky to set up. Configuration or design errors can be particularly perplexing if they result in high priority tasks that consume all of the processing time, so that other tasks never run. Everything looks fine and seems to run fine, except that *nothing happens*. In fact, a runaway high priority task can preempt command and information channel management tasks, so that you can't obtain any information about what is happening, and can't send system commands to do anything about it — other than reload the DAPL system in the host system control panel application.

To assist diagnosing and recovering from these problems, there is a special *rescue task* option. A rescue task is a special task created by the DAPL system that treats all other tasks in the system as if they provided a resource. The rescue task doesn't do anything itself, but because of the artificial resource conflict it temporarily elevates the task priority of every task of priority 2 or higher, in sequence. If it turns out that the task is already running fine without intervention, the rescue task can skip over it and look for other tasks that need help to run. The effect is as if there existed a large round-robin scheduling list giving all tasks a chance to execute when they can't otherwise, partially overriding normal priority protocols.

To control the rescuer task, you need to use three special variants of the `OPTIONS` command.

```
OPTIONS  RESCUE=ON
OPTIONS  RESCUE=OFF
```

The default is `ON`. Setting `RESCUE=OFF` can give maximum multitasking efficiency and minimum response latencies, but you should not do this until you have confidence that your system always operates correctly this way.

The rescuer task is scheduled to run once per each *rescue time interval*.

```
OPTIONS  RINTERVAL=nnnnn
```

Specify the time interval between opportunities for the rescuer task to execute, where the interval length *nnnnn* is expressed in microseconds.

When the rescuer task executes, it is allowed a limited processing time. Specify this time interval as the *rescuer time quantum*.

```
OPTIONS  RQUANTUM=nnnnn
```

The DAPL system establishes default values that will have only a very tiny effect on net real-time performance.

With the rescue task active, you can tell whether tasks produce numerically correct results, verify that data go to the right places, receive error messages, and so forth, even in the presence of configuration problems that cause havoc with process scheduling. The rescue processing adds a slight amount of extra system overhead, and a small additional scheduling uncertainty. It also can create new opportunities for priority inversion. But if your application meets its response and processing requirements, there is no reason to turn off the rescue task. In a well-configured system, where all of the tasks run normally and the rescuer needs to take no action, the only cost is a small delay for a safety check. Thus, the cost is small, while there are some definite risks for not using it.

Using Priorities Effectively

Priorities mean that some tasks are allowed to run in preference to other tasks. That has advantages in that the most urgent things are done immediately, but it can cause complications when tasks at high priority engage in unproductive activity, displacing lower priority processing that could produce useful results. Here are some general rules for getting the best results from tasking priorities.

Run tasks that crunch through large amounts of data at a lower priority.

Run tasks that process small amounts of data to produce responses quickly at a higher priority.

The application example, with a `PID` loop regulating the position of a sensor while a `FIRLOWPASS` task cleans a separate sensor signal, is a good example. Configure the tasks with regular but relatively infrequent control updates to operate at priority 3, either by configuring them there or configuring them so that priority 2 or priority 1 tasks provide the data at boosted priority. Tasks that propagate single values with minimum real-time delay are good candidates for high priority processing. Any CPU capacity not needed for fast response becomes available for other processing.

Drive high priority tasks from low priority tasks for faster response.

There is less delay when driving a high priority task from low priority computation, than from driving a low priority task from a high priority computation. When a high priority task supplies data to a low priority task, the low priority task might not execute immediately. The only time the lower priority task is guaranteed to respond immediately is when its input pipe is already full — priority inheritance will then try to schedule it to facilitate the higher priority processing. But if the pipe is full, that means there was already a long delay to fill it. Furthermore, when a low priority task runs, it might release some other resource conflict, allowing an intervening task to be scheduled, further delaying its response.

On the other hand, when a high priority task receives data from a low priority task, it will respond to all data it receives and ask for more data before any other lower priority tasks can execute. That means the lower priority task also waits for its data at the higher priority. Other tasks at normal priority will not delay it when new data arrive.

Be extra careful with polling loops at high priority.

A polling loop is a program loop that checks a sequence of conditions, performing an action if the corresponding condition is satisfied, or going to the next test in the sequence otherwise. They are most often found in custom-developed dispatch commands that coordinate complex processes. Some coding examples from commands built using the *Developer's Toolkit for DAPL* are shown.

There is nothing fundamentally wrong with polling, as long as the polling is not allowed to deny lower priority tasks any opportunity to execute. For example, consider the following code sequence:

```
// Dispatch loop with polling
while  ( FOREVER )
  {
    if  ( command_pipe_nonempty() )
        ProcessCommandPipe();
    if  ( status_bit_changed() )
```

```
            UpdateLevels();
        check_alarms();
      }
```

If this processing is placed in a low priority command, response might be too irregular because all of the other low priority tasks could interfere. But if this processing is located in a high priority command, this loop will continuously test the command pipe, status bits, and alarm levels, and not release the processor to lower level processing, even if nothing is happening.

It might seem at first that the following change would solve the problem and allow other tasks to execute.

```
// Dispatch loop with voluntary task switching
while  ( FOREVER )
  {
    if  ( command_pipe_nonempty() )
        ProcessCommandPipe();
    if  ( status_bit_changed() )
        UpdateLevels();
    check_alarms();
    task_switch();
  }
```

This helps a little because it allows other tasks at the *same priority* to execute sooner, but lower priority tasks still cannot execute.

In the following sequence, the dispatch loop continues to process until it is certain that all changes are detected and processed. Then, a time delay is allowed before checking again. During that delay, other tasks can run at lower priority.

```
// Dispatch loop
while  ( FOREVER )
  {
    if       ( command_pipe_nonempty() )
             ProcessCommandPipe();
    else if  ( status_bit_changed() )
             UpdateLevels();
    else if  ( check_alarms() )
             /* NIL */;
    else     task_pause( 10 );
  }
```

In this modified loop, if any change condition is detected, the entire checking loop is repeated. But if nothing has changed, a 10 millisecond delay is enforced. During that time, lower priority data processing can execute.

Avoid high priority tasks that depend on low-priority polling.

Ordinarily, a polling task running at low priority can permit other tasks at low priority to execute in round-robin sequence, for example, code in a custom processing command might include the following:

```
// Low priority polling loop
while  ( FOREVER )
```

```
    {
      // Perform necessary actions
      . . .
      // Then allow other tasks to run
      task_switch();
    }
```

But if one of the actions provides data to a high priority task, the high priority task might be BLOCKED waiting for this data, causing the low priority polling task to execute at a boosted priority. The `task_switch()` function then has no effect, because control returns immediately to the polling task, while the other low priority tasks do not get a chance to execute.

Use priorities to regularize output timing at rates different than hardware-controlled output procedures.

As long as you don't overload it with data, a high priority task can generate regular output updates, somewhat like hardware-clocked outputs. It just isn't as fast or as precise as the hardware-controlled updates, due to uncertain task scheduling delays.

A few supplied commands such as `DIGITALOUT` and `DACOUT` provide options to support cyclic processing with timing intervals. Custom processing commands can do the same things. The idea is to prepare for each output but defer it until a regular clock interval expires. During the intervening time, the high priority task will SLEEP. An advantage of this scheme is that there is no harm done if the task runs out of data. Updates will not occur until the data stream resumes. In contrast, a hardware-clocked output sequence is very intolerant of missing data, and will cause an UNDERFLOW condition that stops the output updating.

Avoid data transfers to the host at high priority.

Don't confuse priority with importance. When you operate host system transfers from high priority tasks, all of the delays of the host operating system and bus management will be reflected as delays in all high priority processing. If you must transfer data to the host from a high priority process, consider transferring the data to a low priority task first. In terms of DAPL processing, the low priority tasks might be slow, but in terms of host system responses, the delays will be minimal, and they will not damage critical real-time response.

Don't try to do too much.

Though tasks scheduled at a single priority level have only weak theoretical guarantees of response speed, it often turns out that for a given actual system, there is no way for the configuration to respond slowly. What a round robin scheduling strategy lacks in elegance it can sometimes make up for with simple consistency. It is worth testing a configuration to establish a performance baseline before worrying about the priority levels. If you consistently get the response times you need, without any adjustment of priorities, you should keep your application simple.

14. Overflow and Underflow

Overflow and underflow conditions result when data rates are inconsistent with available data storage. Overflow occurs when sampling is faster than data can be processed and transferred, so that eventually there is no more available memory to store new values. Underflow occurs when output updates deliver data to output ports faster than new data are provided, so that eventually no data are available at the moment when an update cycle is required. Both overflow and underflow are associated with hardware activity that is unable to continue. When underflow or overflow conditions cause the process to stop, the devices are not damaged, and no previous sample values are lost or corrupted. Other processing will continue normally.

If an overflow occurs, processing tasks eventually stop getting new data and cannot continue processing. If an underflow occurs, output processing stops accepting new data, and any new data can fill the transfer pipe so that no more data are accepted. Even though most processing continues normally, an overflow condition or an underflow condition usually leads to an inactive state where all possible processing is done.

Countermeasures include:

- limiting the number of accumulated samples using a `COUNT` option
- reducing the hardware rates specified on the `TIME` command
- intermittent processing using a "burst mode" configuration.

Overflow and underflow conditions are not always errors, and can be applied intentionally. For example, *burst mode output* configurations can accept a block of output data for a complicated output sequence, and after all updates are delivered, output activity stops to await the arrival of a new data burst.

Overflow Messages

The Data Acquisition Processor diagnosis of overflow conditions depends on the `OVERFLOWQ` option. If this option is set to `OFF`, the following overflow message is sent to the host via the `$SysOut` pipe:

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

The `#xxxxx` field of the message gives the sample count at which overflow occurred.

If the `OVERFLOWQ` option is set to `ON`, the overflow message is suppressed. At any time, the host can send the command:

```
DISPLAY OVERFLOWQ
```

The Data Acquisition Processor responds by sending a line containing a single 32-bit integer text via the `$SysOut` pipe. If this number is zero, overflow did not occur. Otherwise, the number indicates the sample number at which overflow occurred.

Example: if a configuration has six-channel input pipe, and overflow occurs after sampling 100,000 times for each channel, the overflow sample number is 600,000.

Preventing Overflow

Overflows are often related to processing bottlenecks. You can check whether data backlogs are occurring in transfer pipes by sampling until just before overflow occurs, and issuing a `DISPLAY PIPES` command. This will display all defined pipes and the number of data values in each pipe. Check for pipes that contain extremely large amounts of data. Another useful tool is the `STATISTICS PIPES` command. It can tell you which pipes reached their capacity, and which tasks are consuming most of the processing time. If a data backlog is present, it is probably due to processing delays at the reader task.

If no user-defined pipe contains excessive data, a backlog of unprocessed data is accumulating in the input channel pipes. Remedies 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 using more efficient processing commands.

WARNING: Hardware sampling in general is faster than the processing that buffers and transfers data across the host bus. Applications that attempt to capture data at maximum Data Acquisition Processor rates and transfer all of that data to the host system should expect overflow problems.

The `COUNT` command can prevent overflow by specifying the number of values to be sampled. Input sampling stops when the sample count is satisfied. The `COUNT` value can be larger that available memory if processing is expected to remove and process part of the data before new samples arrive, but overflow can still occur if the processing is not fast enough. Applications trying to maximize the amount of data capture before overflow should test their configurations carefully to balance between sampling and processing rates.

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 diagnosis of underflow depends on the `UNDERFLOWQ` option. If this option is set to `OFF`, the following message is sent to the host via the `$SysOut` pipe:

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

The `#xxxxx` field of the message gives the update count at which underflow occurred.

If the `UNDERFLOWQ` option is set to `ON`, the underflow message is suppressed. At any time, the host can send the command:

```
    DISPLAY UNDERFLOWQ
```

The Data Acquisition Processor responds by sending a line containing a single 32-bit integer text via the `$SysOut` pipe. If this number is zero, underflow did not occur. Otherwise, the number indicates the update number at which underflow occurred.

Example: if an output configuration has two output channels, and underflow occurs after updating each channel pipe 100,000 times, the underflow update count 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 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.

As output processing starts, there is a race. Who will be first, the task that produces the data, or the output procedure that sends it? If the output processing "won" this race it would immediately underflow and terminate prematurely. If this problem were ignored, the result would be highly unreliable startup performance. To avoid this problem, the DAPL system enforces an "output wait." More precisely, it requires a certain number of output values to be in the buffer and ready before output processing is fully started. An output procedure's initial startup delay is defined using the `OUTPUTWAIT` command. In some instances, increasing an output procedure's initial startup delay allows faster update rates.

Other options are available for avoiding underflow. The `COUNT` command can force an output procedure to stop output updating after a specified number of updates before an underflow condition occurs. The `UPDATE BURST` option of an output procedure lets the underflow happen, and then resumes output updating later when more data are available to satisfy the `OUTPUTWAIT` condition again.

15. 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, you want to dedicate as much CPU power as possible to computation, and as little as possible to other system activities such as task scheduling. Delayed processing is acceptable if it produces better efficiency. In contrast, for better response to real-time events, tasks that must meet response time constraints should execute before other tasks that do not. When they are allowed to run, tasks should do so only for short time intervals to avoid delaying other tasks. Switching tasks more frequently increases system overhead, and makes processing less efficient, reducing the volume of data that can be processed as a tradeoff for reducing the delays.

The two goals of high throughput and fast response are clearly in conflict. Each application should evaluate whether its goal is fast bulk 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 relatively long delays. In contrast, delays of a few milliseconds in a closed loop control application could cause the controlled system to oscillate in a chaotic fashion.

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. Large buffers only make memory management more complicated. The simpler the buffering, the faster the response.

Ordinarily, pipes are `DYNAMIC`, which means that memory will be allocated for new data, and released after the data have been taken. Tasks responding to events in real time typically produce and consume limited amounts of data, often single items. There is no point in continually allocating and releasing memory for just a single item, so pipe operation can be made more efficient by declaring the pipe memory `STATIC`, and configuring an appropriately small `MAXSIZE` property.

Custom processing commands can enforce their own timing strategies. While other processing might try to improve efficiency by bunching samples into larger blocks, thus increasing processing delays, the custom processing logic can deliver a few or just one result per processing cycle, and then release control of the CPU to allow other tasks to respond.

Task Timing Control

The QUANTUM options affect how long and how often a task can run. Shorter time intervals mean less processing efficiency, but shorter delays. DAPL allows each task the same *time quantum* of CPU time at each opportunity to run. In most time-critical applications, this time interval is small, 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.

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. This strategy is not appropriate in all cases, however. 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. Of those four tasks, only the FFT computation takes significant time. The other tasks will complete their processing and release the CPU quickly regardless of the QUANTUM option. The transform doesn't delay anything when it has nothing to do; but when it has data, it produces its results the fastest if there are no tasking interruptions. For this case, the QUANTUM option can be lengthened to allow completion of the transform without interruption by task switching.

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

```
OPTIONS  QUANTUM=2000
```

This is the default quantum value, generally useful for acquisition applications with moderate processing requirements, aimed primarily at efficient throughput. The following option would provide rapid task switching for shorter real-time delays at the expense of data throughput capacity.

```
OPTIONS  QUANTUM=200
```

Evaluating Task Latency

The `STATISTICS` command is useful for evaluating expected task latency. This command will show all active tasks in the system, the amount of time they consume, and the worst case delay (in terms of backlogged samples) experienced by each task.

For systems with so-called "soft" real time requirements, an absolute bound is not very useful, because actual system latency rarely approaches the worst possible theoretical case. The `STATISTICS` command will report the CPU utilization of each active task, and the latency of the entire scheduling sequence. This information can then be used to compare the expected response latency with the observed and theoretical worst cases.

Low Latency Option

Some DAPL commands have special logic that detects the `BUFFERING=OFF` option and selects an appropriate algorithm specialized for that case. This only advises the commands about how to operate: there is no longer any direct effect on the operating mode of the pipes. The `AVERAGE,` and `SKIP,` and DAPL expression commands are examples of commands that are aware of the `BUFFERING=OFF` option.

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. There is system overhead in recognizing that an event has affected the scheduling status, and switching appropriately to a new task. 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. This can mean some complicated programming, but because the operating system's background tasks take little processor time, the custom command will be running most of the time. This approach yields the lowest possible latencies.

If you cannot reduce the processing to one task, perhaps you can partition the processing so that the critical part executes as the only task running at a higher priority. This is almost as good as a single-task solution.

16. 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:

- DLIMIT
- LIMIT
- LOGIC
- PCASSERT
- TGEN
- TOGGLE

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

- 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 start group. 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:

- `NORMAL` Triggered display operation

The `NORMAL` mode simulates normal mode operation of an oscilloscope, in which a display sweep must be completed before responding to another trigger event. It has a `HOLDOFF` property specifying the number of samples until the triggering can occur. The default `HOLDOFF` interval is 0.

- `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.

- NATIVE Deprecated

This mode is equivalent to NORMAL mode, except that it does not allow adjustment of the default property values.

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 TDestruct  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 peak value will be either the most positive or the most negative value observed during a cycle.

```
ABS  (IP0, PA0)
ABS  (IP1, PA1)
ABS  (IP2, PA2)
HIGH (PA0, NCYCLE, PHIGH0)
HIGH (PA1, NCYCLE, PHIGH1)
HIGH (PA1, NCYCLE, PHIGH2)
```

Produce a trigger for this cycle if any of the three peak values is unexpectedly high or unexpectedly low.

```
LIMIT (PHIGH0,  OUTSIDE,17600,26400,  Tph0)
LIMIT (PHIGH1,  OUTSIDE,17600,26400,  Tph1)
LIMIT (PHIGH2,  OUTSIDE,17600,26400,  Tph2)
```

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  (Tph0, Tph1, Tph2, 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 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, samples can be captured on many data channels simultaneously using expansion expansion boards that feature this capability. With many of these boards, however, a single xDAP processor might not be able to collect all of the samples from all of the boards fast enough. Or a single host PC might not have enough data bandwidth to transfer all of the data fast enough. These cases require multiple Data Acquisition Processors synchronized with a master-slave connection.

This presents a challenge for triggering. Typically only one of the processors sees the data from the triggering channel. How can the other processors know when to retain sample 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.

On the MASTER board, a trigger-generating command such as LIMIT generates the events. The triggering information must then be transferred to the other Data Acquisition Processors through communication pipes. Extra communication pipes for this purpose can be set up using the DAPcell 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 DAPcell control panel application, instead of using a configuration set up by the DAPcell 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. Suppose that the first Data Acquisition Processor samples a single high-speed group of eight simultaneous channels. 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 eight data channels, while the master board captures only seven, 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 host applications run at a low priority determined by the host operating system policy. 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 immediate response.

17. 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 computes 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 wide-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 term 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` usually operates like `RAVERAGE`, generating one result for each input sample received. But the commands `FIRFILTER` and `FIRLOWPASS` operate like `AVERAGE` in the sense that they can reduce 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` loads `n-1` values without calculating. Then, after reading the n-th value, `FIRFILTER` calculates another filtered 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 automated and integrated into the DAPstudio software from Microstar Laboratories. After adjusting the controls to obtain a filter characteristic that you think will work, DAPstudio allows you to test the filter on real data interactively, so you can immediately observe the effects and adjust the filter design as necessary.

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. Side effects related to the finite filter length are reduced by multiplying the coefficient values by a window function. This typically is a symmetrical function whose values approach zero near the ends of the block. The DAPstudio software includes the windowing as part of a filter design, and you can switch it in and out of service to observe costs and benefits.

Phase Response and Time Delay

All of the filters designed by DAPstudio software have an odd-numbered length, with symmetry around the center 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; and 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 specify the artificial parameter value -1 and the `FIRFILTER` command will apply the appropriate correction. `FIRFILTER` makes the correction by adding extra samples to the output stream. After this adjustment, features in the filtered data stream will correspond in position to features of the original unfiltered data stream.

This alignment 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, but the first filter computation corresponds to input 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 time shift correction other than `M+1`.

18. Fourier Transforms

The Fourier transform is a mathematical operation that converts data from the *time domain* to the *frequency domain*, or the reverse, from a frequency spectrum to a time sequence. Using the Fourier transform, complex signals are represented in terms of "complex exponentials" consisting of cosine (real) and sine wave (imaginary) parts. While very useful theoretically, these transforms don't in general have a simple finite form that is straightforward to compute.

The discrete Fourier transform (DFT) is a mathematical operation that approximates the Fourier transform for blocks of sampled data. Where a general Fourier Transform has a complex value corresponding to every value of its frequency variable, a Discrete Fourier Transform has defined values only at a finite set of equally-spaced discrete frequencies. The frequencies represented by a DFT are harmonics of one frequency, called the fundamental frequency. The data block used by the DFT has a length equal to one cycle of this fundamental frequency. The DFT can be computed directly from data obtained by sampling a continuous signal. The fast Fourier transform (FFT) is an efficient algorithm for calculating the DFT.

The DFT has two forms, which are mutually inverse operations. The forward transform converts 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 to time domain. Performing the forward transform and then the reverse transform should return the original data set, but as it turns out, there is an extra factor of `N` where `N` is the number of terms in the data block. This extra factor needs to be cancelled somewhere. Because DAPL is used primarily for computations on measurement data, the forward transform is used much more often than the inverse transform. Because the data are better scaled after applying the extra `1/N` factor, the DAPL system applies it to the forward transform.

Even with real-valued input data, the Fourier transform produces complex-valued output data. In many cases the interesting information is in the amplitudes, obtained by combining the parts of the complex output spectrum values. Squaring the amplitudes yields a power spectrum, or more precisely, *power spectral density*. Sometimes the complex values are converted to polar form, yielding amplitude and phase angle.

DFT Command

Though not elegant mathematically, direct computations of the DFT transforms are useful for computing a few frequency terms. For computing a full spectrum, however, you will want to use the `FFT` command.

As a rule of thumb, if the power-of-two size parameter that you would specify for the `FFT` command is `M`, the `DFT` command is worth consideration for computing `2M` or fewer frequencies. It will take less time than computing the full spectrum with the `FFT` command and discarding all of the frequency terms you don't need.

The `FFT` command depends on symmetries that arise when the data blocks are restricted to certain special sizes. The `DFT` command does not depend on those symmetries and consequently you can perform analysis on a data block of arbitrary length. Also, the `FFT` command requires well-structured operations that span across the data block, while the `DFT` analysis can be performed sequentially from the start of the data set to the end, without buffering the input data. This lets a `DFT` analysis process very long data blocks, but with a risk of losing numerical precision.

Like the `FFT` command, the `DFT` command supports windowing of the input data, and it provides various output forms.

FFT Command

The `FFT` command is efficient and versatile. It includes window pre-processing, the actual transform, and post-transform data conversions. The price of this efficiency is that the block size is restricted to be a power of 2. The `FFT` command accepts block sizes from 4 to 16384. For computing spectra over a wide band at high resolution, the `FFT` command is the first choice.

The FFT algorithms require pre-computed tables of coefficients. When FFT sizes are large, these tables become large. To avoid wasted storage, these tables are not allocated in memory 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 larger FFT size first.

Choosing Data Types

When the `1/N` factor is applied to the forward transform, a cosine wave of 10000 units RMS produces a 10000 units RMS peak in the spectrum. But when working with signals that span a broad range of frequencies, the magnitudes of individual terms are much smaller, decreasing approximately inversely as the block length increases.

Most applications involve signals containing various degrees of random noise. If there are initially `K` noisy bits, the noise level tends to increase to about `K+N/2` noisy bits during the DFT computations. The scaling by `1/N` just drops bits that were already lost to noise anyway. For these common cases, direct processing of `WORD-type` data, directly from input samples, is efficient and works just fine. This is the most common way of using the `FFT` command.

For detecting subtle effects in clean signals, however, it is better to avoid loss of numerical precision by performing the FFT calculations using a floating point data type. To use floating point data types, first convert the data type of your input data blocks using a DAPL expression task. A `FLOAT` data type typically works well, and you should try it first. If there are any remaining side effects from scaling or rounding, use a `DOUBLE` data type.

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

The `DFT` command supports a subset of these, the operations related to the forward transforms.

When a forward fast Fourier transform is applied to real data sequences, the transform equations cannot distinguish from the sampled data set whether the real-valued input sequence resulted from sampling a signal below the Nyquist frequency (at term `N`/2 in the spectrum) or above it. More specifically, for blocks of size `N`, 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-valued. The transform calculations assign half of a frequency to the lower frequency and half to the “symmetric image” upper frequency. Because of this symmetry, half of the terms in the FFT spectrum are redundant. Modes 4, 5, and 6 recombines the redundant terms, and returns output blocks half of the size of the input blocks.

Windowing

The underlying assumption of an FFT is that samples in a data block represent one period of a periodic signal. Often, a transform is applied to a data block extracted somewhat arbitrarily 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 sample values in the input data block, term by term, by a vector of coefficients called a window vector. There are some drawbacks, however. 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. Frequency peaks are somewhat widened and reduced in magnitude.

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. Given the block size `N`, the other windows are given by the formulas:

```
1. Hanning(k)    = 0.5 - 0.5 cos(2πk/N)
2. Hamming(k)    = 0.54 - 0.46 cos(2πk/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(2πk/N) + 0.08 cos(4πk/N)
```

All FFT modes accept a window. 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. Predefined window types apply the highest weights to terms near the center of the block and the lowest weights to terms near the end of the block. This works fine for time-domain samples, but it is rarely what you want when applying a window to a frequency spectrum – you typically will want the highest weights near zero frequency and lowest near the Nyquist frequency.

You can design and apply a window of your choosing. Specify the coefficients in a DAPL `VECTOR`, and use the name of the vector rather than a numeric constant in the task definition parameter list. See the description of the `FFT` command for more information about setting up your own windows.

Scaling In FFT Modes

The easiest way to describe the scaling of the various FFT modes is to observe the effects of applying an FFT to a cosine wave of peak amplitude 10000. Pick a frequency that is a harmonic (integer multiple) of the fundamental frequency and less than the Nyquist frequency (half of the sampling frequency).

FFT mode 0 or 1 yields two nonzero spectrum components, each with magnitude 5000. If you combine the power of the two components, $5000^2 + 5000^2$, you will get the value 50000000, the result reported by Mode 4, half of the original magnitude squared. This is consistent with the fact that the mean squared value of a cosine wave is ½ of the peak value squared. Mode 5 reports a value 7071, which is the RMS of a cosine wave with peak magnitude 10000, also equal to the square root of the Mode 4 power value. Mode 6 returns the amplitude along with the phase angle, which would be 0 for a cosine wave.

Modes 2 and 3 are inverse FFT modes. Because the scaling is applied in the forward transform direction, no scaling factor is applied to the reverse FFT. Sometimes this fact can be used to advantage. You can apply a reverse FFT to real data to see the full precision on all terms, noise and all, but keep in mind that the signs will be reversed on all of the imaginary terms.

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

$$x = (x_{[0]},\ x_{[1]}\ \ldots,\ x_{[N-1]}),$$

where $x_{[k]} = u(t_{[k]})$ 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

$$F \quad = \quad 1/T \quad = \quad (1/N)(1/\tau)$$

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

$$f_{[n]} \quad = \quad nF \quad = \quad n/T \quad = (n/N)(1/\tau).$$

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.

Nyquist Frequency

The Nyquist frequency is $f_{[N/2]}$, half the sampling frequency. For a data block of length N, this term corresponds to the term at position N/2 in the DFT spectrum. 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 board 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.

Aliasing

The FFT cannot distinguish frequencies above the Nyquist frequency from frequencies below the Nyquist frequency. In a properly configured application, frequencies above the Nyquist frequency will not be present in the original signal before sampling, hence there is no confusion about what the frequencies in the spectrum represent. Frequencies appearing above the Nyquist frequency in the transform should then be considered equivalent to their symmetric frequencies below the Nyquist frequency. However, if frequencies both above and below the Nyquist frequency are present in the original signal before sampling, you have no way to distinguish them in the sampled data set, and no way to identify the effects in your spectrum. Because of aliasing, the components at frequencies

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

appear to also produce the frequencies

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

Avoid aliasing problems by filtering the input signal to remove the undesirable frequencies at or above the Nyquist frequency before sampling. Later, if you see these frequencies appear in a spectrum, you can then be sure that they are numerical artifacts, and not problems in the original signal. A common and effective filtering technique is to use simple hardware filtering to eliminate very high frequencies, sample at a high rate to get a valid sampling, and then use digital filtering with decimation (such as a DAPL `FIRLOWPASS` task) to decimate the data to the sampling rate required by the FFT analysis.

The General Complex 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. The complex exponential terms are defined by

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

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.

Representing Sampled Data with Cosines and Sines

When a transform is applied to a harmonic frequency that is a cosine wave, the summed sine-cosine product terms cancel to zero, resulting in a transform with only real-valued parts. When a transform is applied to a harmonic frequency that is a sine wave, the summed sine-cosine terms again cancel to zero, leaving a transform with only imaginary-valued parts. This observation gives an alternative interpretation of an FFT as a decomposition into sine and cosine waves. When the waves are sampled at the set of sample times $t = t_{[k]}$, the set of discrete samples from the sine and cosine waves will be

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

We can also observe that cosine functions and sums of cosine functions of any frequency have a mirror image symmetry at times t and -t, while sine functions have a negative symmetry. So the real parts of the transform can be interpreted as the parts of the signal contributing to symmetry at frequency 0, hence also at the Nyquist frequency by extension. Similarly, the imaginary parts of the transform can be interpreted as the parts of the signal contributing to odd symmetry of the waveform at frequency 0 hence, and therefore also at the Nyquist frequency.

This symmetry is closely related to the phenomenon of aliasing. The difference between a sine wave and a cosine wave is only in the phase angle. Depending on what that phase angle is, an undesirable high frequency could produce terms with even symmetry, odd symmetry, or some combination, on the other side of the Nyquist frequency. This is what makes aliasing so unpredictable.

Interpreting the FFT for Real Data

The symmetry relationships become most clear when the input data for the FFT are real-valued. The real parts of the transform satisfy the symmetry relationship

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

for all k, while the imaginary parts of the transform satisfy the negative symmetry relationship

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

From these formulas, it is clear that the n-th and the (N-n)-th transform terms 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 a direct representation of the original real-valued data in terms of sines and cosines. Note that the 0-th term and the N/2-th term are real valued, located at the points of symmetry, and do not have symmetric terms. Thus no additional factor of 2 is required to reconstruct these terms. The 0-th term results in a sequence that equals $X_{[0]}$ for every term. The N/2–th term corresponds to the Nyquist frequency, and results in the sequence $X_{[N/2]}$, $-X_{[N/2]}$, $X_{[N/2]}$, $-X_{[N/2]}$, etc. If you have used proper anti-aliasing techniques for sampling your input signal, the $X_{[N/2]}$ term should be zero or very small.

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. DAPL already takes advantage of symmetry properties, in effect, computing a real-data FFT of size N in approximately the same amount of time as a general complex FFT of size N/2. Because this is done automatically, there is no advantage for you to reformulate your FFT problem in this manner.

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. 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. Transforms are never exact to the very last bit. If you use a floating point number representation, however, the inaccurate bits are probably only low-order bits that are irrelevant anyway.

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 with fixed point data, `m`=8, so do not attach significance to the last 4 bits, or equivalently to differences less than 16. The error in the reverse transform usually is well modeled by “white noise” with a bounded distribution. Averaging can sometimes cancel some of the noise, yielding additional significant bits.

Saturation effects can sometimes be a problem in reverse transforms with fixed-point data. 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, fractional bits are lost after the `1/N` factor is applied, and the rounding effects can appear as various small 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 exactly 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.

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 signal levels are low. 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

19. 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 similar parameters can be repeated, it is followed by a star *.
- If several possible command alternatives exist, the alternatives are separated by vertical bars |.
- 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 commands.

EXAMPLE

An example of a fictitious command with two command forms.

EXAMPLE *(<param1>, <param3>)*

EX *(<param1>, [<param2>,] <param3>)*

Parameters

<param1>

Description of this parameter.
the data types accepted by this parameter

<param2>

Example of an optional parameter accepted only by the second command form.
`WORD CONSTANT`

<param3>

Example of a pipe parameter for output results.
`WORD PIPE | LONG PIPE`

Description

This section provides a comprehensive description of input data required, what the command does, what results are produced, and the purpose of the parameters.

Example

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

Command text in a form that would be accepted by the DAPL command interpreter, with a detailed description of the input data required, configurable options, and the output results produced.

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 | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>

Output data pipe.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`ABS` reads data from *<in_pipe>*, computes absolute values, and places the results in *<out_pipe>*. The data types of the input and output pipes must match.

Example

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

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

See Also

`CMAG`

ALARM

Define a task that sets a bit on a digital output port when input data satisfy a region condition.

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

Parameters

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

<outport>
Address of digital output port.
WORD CONSTANT

<region>
A REGION, specifying the input values that indicate alarm conditions.

<output_bit>
Bit position within the digital output port for signaling alarms.
WORD CONSTANT

<reset>
An integer specifying output mode and a time interval in milliseconds.
WORD CONSTANT | LONG CONSTANT

Description

ALARM reads data values from *<in_pipe>*. When it observes an input value that satisfies the *<region>* condition, it sets the output level of bit *<output_bit>* on the specified *<output_port>* to active high. The tested input data stream can be any basic data type. The two bound numbers of the *<region>* specification (see the "Introduction to DAPL" chapter) must match the data type of the tested input data pipe.

If you do not have digital expansion, specify digital port 0 to send the output directly to the digital connector on the Data Acquisition Processor board. If you are using digital expansion, specify the configured port address of the connector on the expansion board.

The *<output_bit>* position 0 is the least significant bit, and bit 15 is the most significant bit. For compatibility with older hardware, omit the *<output_port>* parameter, and set *<output_bit>* equal to *16 * port address + bit position.*

The *<reset>* parameter specifies the output mode. There are three cases.

1. If the *<reset>* parameter is 0, or omitted, the output has a *tracking* behavior. The duration of the alarm output is indeterminate, but when the alarm condition is no longer present the *<output_bit>* value is returned to inactive low.
2. If the *<reset>* parameter is negative, the output has a *one-shot* behavior. Alarm conditions remain until the system is restarted.

3. If the *<reset>* parameter is positive, the output has a *pulsed* behavior. The active-on condition is held for at least the number of milliseconds specified by the *<reset>* parameter value. Due to uncertainties of task scheduling, the exact duration is indeterminate.

In conditions of high data traffic, the ALARM task will receive data in blocks. It must respond in real time as quickly as possible, so any event within a received block of samples will produce an alarm output immediately. However, the alarm conditions could occur multiple times in one received block, and only one alarm action results. ALARM is not suitable for counting events.

During the times that an alarm event is posted, new data are received and discarded without processing. This avoids a data backlog that could interfere with other processing.

Examples

```
ALARM (P1, 1, OUTSIDE,0,1000, 5)
```

If a value from pipe P1 is outside the range 0 to 1000, bit 5 on digital expansion port 1 is set active high. The digital bit returns to inactive low when the input level no longer satisfies the alarm condition.

```
ALARM (P1, 1, OUTSIDE,0,1000, 5, 10)
```

The same as the previous example, except that the duration of output alarm signals is guaranteed to be at least 10 milliseconds.

See Also

DIGITALOUT, LIMIT

AVERAGE

Define a task that computes the arithmetic mean of a group 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 from which the arithmetic mean is computed.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>

Output pipe for the averaged data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`AVERAGE` computes the arithmetic mean of *<count>* samples. Data values are received from *<in_pipe>* and results are sent to *<out_pipe>*. `AVERAGE` is useful for data compression and noise reduction. The data types for the input and output pipes must match.

Examples

```
AVERAGE (IPIPE0, 10, PAVG)
```

Average groups of 10 values from input channel pipe 0 and send the averages to pipe `PAVG`.

```
AVERAGE (PAVG, 4, P3)
```

Average groups of 4 values from pipe `PAVG` and send the averages to pipe `P3`.

See Also

`BAVERAGE`, `FIRFILTER`, `RAVERAGE`

BAVERAGE

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

BAVERAGE *(<in_pipe>, <nchannels>, <mblocks>, <out_pipe>)*

Parameters

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

<nchannels>
The number of values in a block.
`WORD CONSTANT | LONG CONSTANT`

<mblocks>
The number of blocks of data to be averaged.
`WORD CONSTANT | LONG CONSTANT`

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

Description

<nchannels> and *<mblocks>* are positive nonzero integers. *<nchannels>* indicates the number of values (channels) in the multiplexed block. BAVERAGE reads *<mblocks>* blocks, averages corresponding points in the blocks, and puts one block of multiplexed averages into *<out_pipe>*. For every *<nchannels>***<mblocks>* values read, *<nchannels>* values are put into *<out_pipe>*.

Some common applications of BAVERAGE are

- reducing random noise,
- statistical averaging data from multiple channels in parallel,
- 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

BMERGE

Define a task that merges equal-size blocks of data.

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

Parameters

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

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

<blocksize>
A number that specifies 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 blocked data, such as the outputs of WAIT or FFT tasks, BMERGE is more efficient than MERGE, because data are processed in groups rather than individually, without any sdata conversion.

The data types of pipes *<in_pipe_0>*, *<in_pipe_1>* etc., and pipe *<out_pipe>* must all be the same. The maximum size of *<blocksize>* is 32768. There can be up to 64 data source pipes. Very larger block sizes could result in data transfer inefficiency.

BMERGE should be used only when *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>*, are filled at the same rate. Otherwise, BMERGEF should be used.

Example

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

Read blocks of 1024 data values sequentially from the four data pipes P1, P2, P3, and P4, in sequence, 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 | FLOAT PIPE | DOUBLE PIPE

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

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

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

Description

BMERGEF merges blocks of equal size from *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>* to *<out_pipe>*, adding an identifying flag before each block. Though the block size is fixed, the order in which the blocks will arrive in the destination *<out_pipe>* is indeterminate. For blocked data, such as the outputs of WAIT or FFT tasks, BMERGEF is more efficient than MERGEF.

Data may be of any data type, but the data types of all source pipes and the destination pipe must match.

BMERGEF scans through *<in_pipe_0>*, *<in_pipe_1>*, … , *<in_pipe_n-1>*. If a pipe contains at least *<blocksize>* values, BMERGEF writes an identifying flag to *<out_pipe>*. The identifying flag is a number from 0 to *n-1*, converted to the data type of the *<out_pipe>*. BMERGEF then reads *<blocksize>* values from the identified source pipe and writes those values to *<out_pipe>*. Block sizes must be 16384 or shorter.

A software application program receiving data from BMERGEF should read a tag items to determine the source pipe identity and then read *<blocksize>* values.

Example

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

Read blocks of 1024 data values from pipes P1, P2, P3, and P4 as the data become available. Place an identifying flag before each block, and send the blocks to the PC host.

See Also

BMERGE, MERGE, MERGEF, SEPARATEF

BOUND

Define a task that limits values in a data stream to a range.

BOUND *(<in_pipe1>, <low_bound>, <high_bound>, <out_pipe>)*

Parameters

<in_pipe>

Input data pipe.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<low_bound>

Lower limit of the value range.

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

<high_bound>

Upper limit of the value range.

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

<out_pipe>

Output pipe.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`BOUND` enforces range limits on the values taken from the *<in_pipe>* source.

- Values greater than or equal to *<high_bound>* are replaced by *<high_bound>*.
- Values less than or equal to *<low_bound>* are replaced by *<low_bound>*.
- Values within the range are unchanged.

After enforcing the range limits, the results are placed into output pipe *<out_pipe>*. The data types must be the same on all parameters.

Example

```
BOUND (P1, 0.0, 25000.0, P3)
```

Read the values from pipe `P1`, and adjust values that are beyond the range limits 0.0 to 25000.0 to the nearest range limit. Place the adjusted values into pipe `P3`.

BUFFERS

Specify the input buffer mode of an input configuration.

BUFFERS *<type>*

Parameters

<type>

A keyword.
`STATIC | 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.

Example

```
BUFFERS STATIC
```

For a configuration that operates upon small data blocks on a regular schedule, set the buffer mode to `STATIC` in the input procedure, to slightly reduce worst-case buffer management latency.

CALIBRATE

Calibrate the Data Acquisition Processor input sampling hardware.

CALIBRATE *[<STORE | NOSTORE>]*

Parameters

<STORE>
Store calibration values to onboard nonvolatile memory.

<NOSTORE>
Do not store calibration values to onboard nonvolatile memory. This is the default if not specified.

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. 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*.

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.

See Also

`RESET`

CANGLE

Define a task that computes the phase angle of complex values.

CANGLE *(<in_pipe1>, <in_pipe2>, <out_pipe>)*

Parameters

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

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

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

Description

`CANGLE` computes the phase angle of complex numbers, with real parts taken from pipe *<in_pipe1>* and imaginary parts taken from *<in_pipe2>*. This operation is equivalent to performing a four-quadrant arctangent function such as atan2(y,x) in the C, Pascal, or FORTRAN programming languages. The result is placed into *<out_pipe>*. The input and output data streams must all be the same data type.

The output values are angles in the range of –pi to +pi. Special representations are needed to represent this numerical range with fixed-point numbers. See the discussion of the `POLAR` command.

Note: The phase angles of complex numbers also can be computed using mode 6 of `FFT`, or by the `POLAR` command.

Example

```
CANGLE (P1,P2,P3)
```

Read the real part `x` of a complex value from pipe `P1` , and the imaginary part part `y` of the complex value from pipe `P2`. Compute the phase angle $\tan^{-1}(y/x)$ and place the result in pipe `P3`.

See Also

`ABS`, `FFT`, `POLAR`

CHANNELS (input sampling)

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

CHANNELS *<nchannels>*

Parameters

<nchannels>

The number of input channels to receive data samples.
WORD CONSTANT

Description

The CHANNELS command in an input procedure definition specifies the number of signal channels that will be sampled when the input sampling is started. The characteristics of each channel will be specified later in the configuration. The number of signal channels must be known before other configuration information can be processed. The CHANNELS command should appear as one of the first commands following the IDEFINE command.

Usually, each channel specified by the CHANNELS command will be associated with a signal pin, by specifying *<nchannels>* number of SET commands. However, a configuration will sometimes omit declarations of some of these channels. For example, artificial channels might be included in the sequence for purposes of aligning the samples at certain specific time intervals, and since no samples will be processed for these particular time intervals, the associated SET commands are omitted. All channels occupy memory whether or not an external signal pin is assigned, and whether or not any data processing uses the signal channel.

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.

Example

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

Configure the input sampling to capture data for four signal channels, on a Data Acquisition Processor model that samples signal pins individually. The total number of input data channels is four, but there are only two unique hardware signal paths. The last channel is not used for any data processing, but it is present only to preserve an even 100 microsecond timing interval for capturing the channel list set.

See Also

SET, GROUPS, IDEFINE

CHANNELS (output updating)

Define the number of output channels in an output updating configuration.

CHANNELS *<nchannels>*

Parameters

<nchannels>

The number of output channels to receive clocked updates.
`WORD CONSTANT`

Description

In an output procedure for a Data Acquisition Processor, the `CHANNELS` command must be specified to configure the number of output channels that will receive updates.

An output configuration needs to know the number of signal channels before it can process other configuration information. The `CHANNELS` command should appear as one of the first commands following the `ODEFINE` command. Each of the *<nchannels>* output channels specified by the `CHANNELS` command must be assigned a destination hardware signal pin.

Example

```
ODEFINE Outp2
   CHANNELS 2
   SET IP0  A0
   SET IP1  B0
   TIME 500
END
```

Configure an output updating procedure that will deliver data samples to output signals: one analog output channel and one digital port channel.

See Also

`SET`, `CHANNELS`, `ODEFINE`

CHIRP

Define a task to generate a swept cosine waveform.

CHIRP *(<sweep>, <direction>, <amplitude>, <cycle>, <periodL>, <periodH>, <phase0>, \ <outpipe>)*

Parameters

<sweep>
Frequency sweep function selection.
`STRING`

<direction>
Uni-directional or bi-directional sweep selection.
`STRING`

<amplitude>
Amplitude of waveform.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<cycle>
Number of samples to cover one full sweep cycle.
`WORD CONSTANT | LONG CONSTANT`

<periodL>
Period of cosine waveform at initial sweep frequency.
`WORD CONSTANT | LONG CONSTANT`

<periodH>
Period of cosine waveform at final sweep frequency.
`WORD CONSTANT | LONG CONSTANT`

<phase0>
Initial phase angle of cosine waveform.
`FLOAT CONSTANT | DOUBLE CONSTANT`

<outpipe>
A data pipe where samples of the swept frequency waveform are placed.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`CHIRP` generates a frequency-modulated cosine waveform in pipe *<outpipe>*, where the *<sweep>* identifier specifies the kind of function used to vary the frequency. It must be one of the following strings:

- `“LINEAR”` (abbreviation `“LIN”`)
  Frequency increases proportionally to time
- `“QUADRATIC”` (abbreviation `“QUAD”`)
  Frequency increases with the square of time

- `“LOGARITHMIC” (abbreviation “LOG”)`
  Frequency increases with time as an exponent

The sweep is repeated, with each sweep generating *`<cycle>`* samples.

You can specify an arbitrary starting phase angle *`<phase0>`* for the cosine waveform, in radians. To get a swept `cosine` wave, specify 0.0 as the initial phase angle. To get a swept `sine` wave (starting at signal level 0.0), specify a phase angle of -1.5707963 (equal to `-pi/2`).

The frequencies are characterized by their periods, i.e., the number of samples to span a complete cosine wave cycle. The low frequency range for the sweep is specified by *`<periodL>`* and the high frequency range is specified by *`<periodH>`*, subject to the restriction that *`<periodL>`* and *`<periodH>`* must be 2 or greater to remain below the theoretical Nyquist limit for sampled data. Ordinarily, *`<periodL>`* and *`<periodH>`* will not exceed the length of *`<cycle>`*.

Most applications sweep from low frequency to high. A sweep from high frequency to low can be obtained by making *`<periodH>`* longer than *`<periodL>`*.

During each cycle, the *`<direction>`* parameter determines how the frequency is swept.

- “`UNIDIRECTIONAL`” (abbreviation “`UNI`”). The sweep goes smoothly between *`<periodL>`* and *`<periodH>`*, in the manner of a *sawtooth shape*, jumping back to the initial frequency to begin the next cycle.
- “`BIDIRECTIONAL`” (abbreviation “`BI`”). The sweep goes smoothly between *`<periodL>`* and *`<periodH>`*, then smoothly back again during each cycle, in the manner of a *triangle shape*.

Waveforms are evaluated by taking the function for generating the current frequency, as determined by the *`<sweep>`* type, and integrating it to obtain the current argument to the `cosine` function. After evaluation, the waveform sample is scaled according to the *`<amplitude>`* parameter, which specifies the peak level of the signal in the data type of the desired output signal. Scaled values are placed into *`<outpipe>`*.

Note: It is not possible to meet the constraints of initial sweep frequency, final sweep frequency, and initial phase angle, *while also* meeting the constraint that the final phase angle matches the initial phase angle. A step discontinuity can occur between sweep cycles, particularly when using bi-directional sweeps. Possible solutions: (1) discard response data corresponding to the sweep discontinuity or (2) adjust the sweep cycle length up or down slightly to reduce the magnitude of the phase discontinuity.

Example

```
CHIRP( “linear”, “unidirectional”, 32767, 4000, 800, 8, 0.0, PSWEEP)
```

Generate a repeating swept cosine wave “chirp” of maximum amplitude 32767 that will fit into `WORD` output pipe `PSWEEP`. Each full sweep covers 4000 samples. The period at the initial frequency is 800 samples, meaning that there would be five full waveform cycles if the frequency stayed at the initial frequency. The frequency changes linearly with time, and increases until the period of the final frequency is 8 samples per cycle, 25% of the theoretical Nyquist sampling limit. The sweep is uni-directional, so each sweep begins again with the frequency returning to an 800-sample period.

See Also

`COSINEWAVE`, `RANDOM`

CLCLOCKING

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

CLCLOCKING *<switch>*

Parameters

<switch>

A keyword.

`ON | 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 samples in an input channel group are sampled simultaneously.

Example

```
CLCLOCKING OFF
```

Turn channel list clocking off.

See Also

`CLOCK for input`, `CLOCK for output`, `HTRIGGER for input`, `HTRIGGER for output`

CLOCK (input sampling)

Select the source of the input configuration sample clock.

CLOCK *<source>*

Description

CLOCK specifies the clock source used by an input sampling configuration. *<source>* can be 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 input clocking pin.

See Also

CLCLOCKING, HTRIGGER

CLOCK (output updating)

Select the source of the output configuration sample clock.

CLOCK *<source>*

Description

CLOCK specifies the clock source used by 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 output clocking pin.

See Also

CLCLOCKING, HTRIGGER

CMAG

Define a task that computes the magnitude (absolute value) of complex values.

CMAG *(<in_pipe1>, <in_pipe2>, <out_pipe>)*

Parameters

<in_pipe1>

First input data pipe for real-valued terms.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<in_pipe2>

Second input data pipe for imaginary-valued terms.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>

Output data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`CMAG` computes the absolute values of complex numbers, with real parts taken from pipe *<in_pipe1>* and imaginary parts taken from *<in_pipe2>*. To compute each value, corresponding input terms from the two source pipes are squared and summed. The square root is taken and the result is placed into *<out_pipe>*. The input and output data streams must all be the same data type.

When using large fixed-point numbers, the result can be too large to represent in the output data type, and in such a case, the output value is saturated to the largest representable value. When using small fixed-point numbers, there are very few bits in the number representation and the truncation of the final square root operation can be significant. Numerical problems such as these can be avoided by converting data to floating point with a DAPL expression before applying this operation.

Note: The magnitudes of complex numbers also can be computed using modes 5 or 6 of `FFT`, or by the `POLAR` command.

Example

```
CMAG (P1, P2, P3)
```

Read the real part of a complex value from pipe `P1`, and the imaginary part of the complex value from pipe `P2`. Compute the absolute value of the complex number and place the result in pipe `P3`.

See Also

`ABS`, `FFT`, `POLAR`

COMPRESS

Define a task that encodes data having infrequent changes.

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 number or sequence of numbers that represents the threshold value for which to report changes.
`WORD CONSTANT | WORD VARIABLE`

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

Description

`COMPRESS` encodes 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 stream. If the absolute difference is greater than or equal to the *<threshold>* parameter for that stream, 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 stream.

The first parameter *<in_pipe>* specifies one or more data streams. The first parameter typically is an input channel pipe list, but it can be any data pipe with multiplexed data.

The second parameter *<n>* is a positive constant, less than or equal to 16, specifying the number of data streams read from *<in_pipe>*. If *<in_pipe>* is an input channel pipe list, *<n>* must be equal to the number of input channel pipes. For a single pipe, *<n>* must be 1. The data streams from the input pipe are numbered 0 through *<n>*-1.

Either one or multiple *<threshold>* parameters must appear next in the parameter list. Each *<threshold>* value is a nonnegative value. If a single *<threshold>* parameter is specified, it will apply to each stream in the input pipe list. If more than one *<threshold>* parameter is specified, there must be one *<threshold>* parameter for each of the *<n>* multiplexed data streams, in sequence.

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 stream from which the threshold value was exceeded. A zero indicates the first stream, a 1 indicates the second stream, and so on. The second value is the new 16-bit input data value. The last field is a 32-

bit unsigned integer specifying the sample number of the new value, where the data are counted in one channel starting from sample number 0.

Examples

```
  COMPRESS (IP0, 1, 100, P1)
```
Send data block to pipe P1 when input values change by more than 100.

```
  COMPRESS (IPIPES(0,1,2), 3, 64, 64, 1000, P1)
```
Send data block if input channel pipe 0 or 1 changes by more than 64 or if input channel pipe 2 changes by more than 1000.

See Also
`AVERAGE`, `LIMIT`

CONSTANTS

Define a shared constant data element.

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

```
CONSTANT <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.

If the data type specification is omitted, the data type of the constant defaults to `WORD`, and the initializer expression must be compatible with `WORD` data type.

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>
  First output pipe for copied data.
  `WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe_n>
  Last output pipe for copied data.
  `WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`COPY` transfers values from *<in_pipe>* to one or more output pipes. Up to 64 output pipes are allowed. The data types of the *<in_pipe>* and *<out_pipe>* parameters must all match.

A `COPY` task can be used to circumvent the restriction that data-reading tasks for one pipe must all be in the same start group. The `COPY` command can produce an independent copy of a data stream. A task in another start group can then read that copy.

Example

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

See Also

`COPYVEC`, `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 | FLOAT VECTOR | DOUBLE VECTOR`

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

Description

`COPYVEC` continuously copies data from a vector to a pipe. This is used to generate repetitive waveforms or other specialized test signals. The data types of the pipe and the vector must match.

Example

```
VECTOR VF FLOAT = (1.0, 2.0, 3.0, 4.0)
COPYVEC  (VF, P)
```

Repeatedly 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>, <scaling>, <bsize>, <lead>, <lag>, <window>, <p3>)*

Parameters

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

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

<scale>
A final scaling divisor to apply to each term.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<bsize>
The size of data blocks processed from input streams.
`WORD CONSTANT`

<lead>
The maximum lead time, expressed as a positive number of sample intervals.
`WORD CONSTANT`

<lag>
The maximum lag time, expressed as a positive number of sample intervals.
`WORD CONSTANT`

<window>
A parameter that specifies an FFT window operation.
`WORD CONSTANT |`
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<p3>
Output pipe for correlation values.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE 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>*. To obtain the correlation at this lead or lag, corresponding pairs of values from the two data blocks are multiplied, and the resulting terms are averaged.

The range limits for the time shift T are 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 pair of input data blocks, *<lead>* + *<lag>* + 1 computed correlation values are placed into output pipe *<p3>*, starting with the correlation result for the greatest lead and continuing to the correlation result for the greatest lag.

The correlation calculations are equivalent to a convolution operation, and an FFT is used to compute the convolution efficiently. The FFT block size *<bsize>* used for the correlation analysis must be greater than *<lead>* + *<lag>* + 1 items, and must be an exact power of 2 no greater than the largest block size supported by the FFT command. There is a tradeoff between statistical significance and locality. For effects with localized short term correlation, a longer block will obscure the results by including more noise in the analysis, but for more persistent correlations, a longer block will improve the accuracy of the estimates.

For highly correlated data, the magnitude of the correlation is approximately the square of the magnitude of the data terms, and for some data types this can lead to unrepresentable values. To avoid this, a *<scale>* parameter is provided. In previous versions of the DAPL system, the scale divisor was fixed at 16384. If a value of 0 or 16384 is specified, the scaling will be the same as before. For effects that are less strongly correlated, a lower scaling factor might preserve more bits of precision. When using floating point data, scaling is seldom a problem, and the scaling divisor can be set to 1.0 or to any other useful value. The scaling divisor type must match the processed data type.

<window> specifies a window operation. The CORRELATE command accepts the same pre-defined window codes or user-defined window vectors as the FFT command. See the description of the FFT command for more information about window operations for FFT analysis.

A typical application of CORRELATE is finding the propagation delay 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 the signal to propagate from input to output.

Example

```
CORRELATE (P1, P2, 16384, 1024, 31, 32, 3, P3)
```

Calculate cross correlation between blocks of 1024 points taken from pipes P1 and P2, producing a block of blocks of 64 points for each analysis. In each block of 64 results computed, the first 31 are for lead, the next term is for no lead or lag, and the last 32 terms are for lag, in the order of greatest lead to greatest lag. Use a Bartlett (triangular) window operation (code 3) for the FFT analysis. Place the results into pipe P3.

See Also

FFT

COSINEWAVE

Define a task that generates cosine wave data.

COSINEWAVE *(<amplitude>, <period>, <out_pipe>*
[, <mod_type>, <mod_pipe] [, <mod_type>, <mod_pipe])

Parameters

<amplitude>
The absolute magnitude of the waveform peak.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

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

<mod_type>
An optional modulation selector keyword.
`STRING`

<mod_pipe>
Pipe for a modulation signal, when a selector keyword is specified.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`COSINEWAVE` generates cosine wave data and places the data in *<out_pipe>*. The *<period>* is the number of sample values in each wave cycle. The *<amplitude>* is the absolute magnitude (positive one half the peak to peak range) of the output wave.

The *<amplitude>* does not have to match the output data type exactly, but it is restricted to representable values of the output data type. For example, the integer 100000 would be valid as the amplitude for a `float` output signal, but 61472133 would be invalid because it is not representable without rounding.

The cycle length specified by *<period>* is the number of samples for one complete waveform cycle. For fixed point output data types, the cycle length must be a fixed point number. For floating point data types, the cycle length does not need to be an exact integer, which means that it can produce frequencies that are not harmonically related to the sampling frequency. Aribtrarily long values of *<period>* are allowed. The most efficient operation is achieved using fixed point data and a fixed waveform length of 2048 terms or less.

Note: The `COSINEWAVE` is identical to `SINEWAVE` except for the phase of the signal.

There are two optional modulations. No modulation, one modulation option, or both modulation options can be specified. Each modulation specification consists of a selector string, followed by a pipe name identifying the modulation stream. The two modulation options are:

1. *Amplitude modulation. Modulation type "`AMPLITUDE`".*
   Values of the modulation signal multiply the corresponding values of the original waveform. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary.
2. *Frequency modulation. Modulation type "`FREQUENCY`".*
   Values of the modulation signal act as a multiplier on the waveform frequency. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary but typically will be in the range 0.0 to 1.0. For example, if the *`<period>`* is 500 and the values from the frequency modulation pipe are all 0.5, the frequency is cut in half and the effect is the same as setting *`<period>`* equal to 1000.

When applying modulation, one modulation value is used for each waveform value generated.

Examples

```
COSINEWAVE (1000, 100, P2)
```

Generate a cosine wave with values ranging from -1000 to 1000, with a period of 100 samples. Place the waveform data into pipe P2.

```
COSINEWAVE (32767, 400, PMOD, "AMPLITUDE", PAMPL)
```

Generate a cosine wave with output values ranging from –32767 to 32767, and with a cycle length 400. Apply amplitude modulation, taking the multipliers from pipe `PAMPL`, and placing modulated waveform data into pipe `PMOD`.

See Also

`SAWTOOTH`, `SINEWAVE`, `SQUAREWAVE`, `TRIANGLE`

COUNT (for input sampling)

Specify the amount of data to collect during input sampling.

COUNT *<sample_count>*

Parameters

<sample_count>
An integer specifying the number of samples to collect before stopping.
`WORD CONSTANT | LONG CONSTANT | extended sample count`

Description

A `COUNT` command is used in an input sampling configuration to stop sampling after a predetermined number of samples. The *<sample_count>* parameter specifies the total number of samples collected in all input channels before stopping. Divide *<sample_count>* by the number of input channels in the configuration to obtain the number of times each individual channel is sampled.

<sample_count> must be a nonzero, positive integer, and a multiple of the number of channel pipes. For extended-time sampling, it can be as large as 9.22×10^{18}. For Data Acquisition Processor models that provide simultaneous sampling of multiple signal lines, the *<sample_count>* must be an integer multiple of the number of channel groups times the sampling group size.

When the `SAMPLE` command specifies the `BURST` sampling mode, the `COUNT` command is used to end data collection in each burst.

Example

```
CHANNELS  100
COUNT   100000
```

When the input configuration is started, the Data Acquisition Processor acquires 100000 samples, or 1000 samples for each input channel. Then the configuration stops.

See Also

`CYCLE`, `SAMPLE`, `HTRIGGER for input`, `COUNT for output`

COUNT (for output updating)

Specify the number of updates for an output configuration to generate.

COUNT `<update_count>`

Parameters

<update_count>

An integer that specifies the number of updating operations.

`WORD CONSTANT | LONG CONSTANT | extended sample count`

Description

A `COUNT` command in an output configuration definition specifies the number of updates that the output configuration produces before stopping. The *<update_count>* parameter specifies the total number of updates generated in all output channels. *<update_count>* must be a nonzero, positive integer, and a multiple of the number of channel pipes. For extended-time signal generation, it can be as large as 9.22×10^{18} . Divide *<update_count>* by the number of output channels in the configuration to obtain the number of times each individual channel is updated.

When the `UPDATE` command specifies the `BURSTCYCLE` output mode, a `COUNT` command is required to specify the length of the output burst.

A `COUNT` command can be used in conjuction with a `CYCLE` command to generate multiple copies of a wave cycle before stopping. The `CYCLE` command specifies the number of passes through the channel list to generate one cycle of a waveform in every output channel. The number of updates produced by this cycle is the number of channels times the `CYCLE` value. To get an exact integer number of cycles, multiply that value by the number of cycles to generate, and specify the result value in the *<update_count>* field.

Example

```
CHANNELS  10
COUNT  10000
```

When the output configuration is started, the Data Acquisition Processor delivers a sequence of 10000 output updates and then stops the updating process. Because there are 10 channels in the configuration, each channel will be updated with 1000 output values.

See Also

`CYCLE`, `HTRIGGER for output`, `UPDATE`, `COUNT for input`

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 | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p2>

An input pipe that represents the output signal from the device under test.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<m>

A number specifying the size of the transform data block.
`WORD CONSTANT`

<window>

A windowing vector or a number that specifies a predefined window.
`WORD CONSTANT |`
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<p3>

A output pipe that contains the real part of the crosspower spectrum.
`LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p4>

A output pipe that contains the imaginary part of the crosspower spectrum.
`LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p5>

An optional pipe that contains the autopower spectrum of *<p1>*.
`LONG PIPE | FLOAT PIPE | DOUBLE 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`. Under some definitions, reactive power is the `CROSSPOWER` result.

When using `LONG`, `FLOAT`, or `DOUBLE` input data streams, the input and output types must match. When the input data type is `WORD`, this is a special case because there there is very little chance that the computed results will fit into `WORD` data type without overflowing the representable range. The `CROSSPOWER` command will require the output data type `LONG` for the special but very common case of input data type of type `WORD`. The 32-bit `LONG` data streams also have range limitations, but unfortunately no extended precision fixed-point data type for returning large results, so you should avoid `LONG` input data for most applications.

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` which contains the autopower spectrum of *<p1>*. The autopower spectrum always is real.

The parameters *<m>* and *<window>* configure the FFT and have the same meaning as in the `FFT` command. *<m>* is the size of the transform. *<window>* is a window vector specification or predefined window selection code. See the `FFT` command for details.

`CROSSPOWER` can be used in conjunction with `TFUNCTION2` for calculating the transfer function of a “black box” from input and output data. The advantage of using crosspower and autopower spectra is that these are insensitve to phase and can be averaged to reduce noise prior to computing the transfer characteristic.

Example

```
  CROSSPOWER (P1, P2, 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 of `P1` in pipes `P3` and `P4`, and the autopower spectrum of `P1` in pipe `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 which 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 which 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, and write the results to word pipe `P1`.

See Also

`CTCOUNT`

CYCLE

Specify that an output configuration generates a repeating pattern.

CYCLE *<n>*

Parameters

<n>

The number of output updates for each output channel pipe before repeating.
`WORD CONSTANT | LONG CONSTANT`

Description

The `CYCLE` command specifies that an output configuration must produce a repeating pattern. The parameter *<n>* specifies the number of passes through the output channel list. Each output channel will receive *<n>* updates before the cycle repeats. *<n>* can take any positive value up to the maximum memory space available. Typical applications are the generation of complex waveforms and repetitive control sequences.

Note: `CYCLE` is different from other output configurations commands, such as `COUNT`. `CYCLE` specifies the number of channel lists processed while `COUNT` specifies the number of update values processed.

Because of the repeating data cycles, the output processing requires only *<n>* values per channel from its data source. These values can then be used many times from memory to generate an output burst. This makes signal generation very efficient. When the output updating mode as specified by the `UPDATE` command is `CONTINUOUS` or `BURSTCYCLE`, the output processing accepts no additional data and the supplier task is effectively shut down after delivering these data. Other tasks should not depend on the supplier task as a continuing data source. It is very common to use waveform synthesis commands such as `SINEWAVE` or `SAWTOOTH` to produce the data for the output sequences.

When the `BURST` or `BURSTCYCLE` updating mode is specified by the `UPDATE` command, and the `CYCLE` command is specified as well, the output updating activity starts when each channel has received the *<n>* update values per channel to cover the `CYCLE` specification. The burst ends when the `COUNT` command is satisfied. For the case of `BURST` updating mode, an additional *<n>* new update values per channel, to cover the `CYCLE` specification again, are required before another burst can occur. In the `CONTINUOUS` updating mode, the cycle repeats indefinitely, until the entire output processing configuration is stopped.

Example

```
CYCLE 1024
```

Specify that each channel in the output configuration repeats a pattern of 1024 updates.

See Also

`UPDATE`, `SAWTOOTH`, `SINEWAVE`, `COSINEWAVE`, `SQUAREWAVE`, `TRIANGLE`

DACOUT

Define a task that updates a digital-to-analog converter independent of output clocking hardware.

```
DACOUT ( [init_value,] <in_pipe>, <dac_number> [, <slew_option>]
   [, <interval> [, <reset_time>]] )
```

```
DACOUT ( <in_var>, <dac_number> [, <interval>] )
```

Parameters

<init_value>
Optional initial output value for digital-to-analog converter prior to updates.
`WORD CONSTANT`

<in_pipe>
Source data pipe for first command form.
`WORD PIPE`

<in_var>
Source variable for second command form.
`WORD CONSTANT | WORD VARIABLE`

<dac_number>
Hardware address of the digital-to-analog converter.
`WORD CONSTANT`

<slew_option>
Case-sensitive keyword (no quotes) specifying how to process a data backlog.
`skip | slew | noskip | skipalign | slewalign`

<interval>
Optional updating interval in microseconds.
`WORD CONSTANT | LONG CONSTANT`

<reset_time>
Delay time in microseconds before resetting aligned counting modes to zero.
`WORD CONSTANT | LONG CONSTANT`

Description

`DACOUT` updates a digital-to-analog converter asynchronously by reading from a data stream <*in_pipe*> or from a shared scalar value <*in_var*>, and writing the value to a converter specified by parameter <*dac_number*>, bypassing the hardware-controlled timing of an output procedure. This command is frequently applied in control and monitoring applications where the delays of hardware data buffers for regularly clocked updates are unacceptable, but the extreme precision of hardware output clocking is not necessary.

The <*dac_number*> parameter specifies a hardware address for the digital-to-analog converter. Hardware addresses are slightly different than the logical channel numbers that would be used in output configuration commands. The address numbers for onboard DACs are 0 and 1. Address number 2 accesses analog output expansion port 0, address number 3 accesses output expansion port 1, and so on.

The behaviors are different for the two kinds of data sources, so there are two command forms.

For the command form with a *shared variable data source*, the value of the variable is used to initialize the output port state when the command starts. Other tasks can change the variable value at any time, and the only way the `DACOUT` command can know whether this has happened is to examine the variable value each time there is an opportunity to execute. If `DACOUT` is running as a high-priority task, this has a danger of soaking up all available CPU capacity, so that other tasks cannot run. To avoid this hazard, you can specify a positive <*interval*> parameter value, in microseconds, setting a minimum time interval between tests. If you do not specify the interval, the DAPL system will select a default.

For the command form with a *pipe data source*, the behaviors of `DACOUT` are a little more complicated. A value might not arrive promptly to set the initial converter output state, so an optional <*init_value*> parameter can specify the initial output state to apply when the command starts, prior to the first update. After that, new values from the pipe can represent continuous streams or occasional bursts.

The <*slew_option*> and <*interval*> parameters work in combination to tell the `DACOUT` command the strategy for obtaining update data and controlling when they are sent to the DAC.

If no <*interval*> value is specified, the `DACOUT` command will send the data out as quickly as possible, any time the task scheduler allows the `DACOUT` command to execute, for minimum real-time delays. If more than one value is received, all values but the most recent one are considered "old history." Use one of the following <*slew_option*> choices.

- `skip` - If more than one update value is available, ignore old values and post the most current one to minimize real-time delays. This is the default behavior with no <*slew_option*> specified.
- `slew` - Almost the same as the `skip` option, but use all data, and spin through all of the updates quickly. The digital-to-analog converters might not be able to track this rapid sequence, however. Accuracy of the output stream during these transient conditions is not assured, so use this mode with caution.

If you specify an <*interval*> value, the `DACOUT` command will try to send updates at the specified interval in a regular sequence. Timing starts with the arrival of the first update sample. The interval is determined by a clock that is independent of the input and output configurations. Regular updating intervals are possible, but not with the speed and timing accuracy of hardware control. There is some uncertainty about the exact time of an update when other tasks coexist with the `DACOUT` command at equal priority

There are two possible assumptions about the way that data will arrive at the `DACOUT` command when an <*interval*> value is specified. Under the first assumption, the data will arrive in real time, one value for each update interval. If multiple values are received, this indicates that a delay occurred somewhere. Only the most recent value means anything, and the rest are "old history." Use one of the following three <*slew_option*> modes.

- `noskip` - When a timing interval arrives, if an update value is available, send the first one to the DAC. Never discard any update data. Use this option when a precomputed data block is provided and it is important to reach each level without missing anything.
- `skip` – Discard old data and send only the most recent value to the DAC. This is the same as `skip` mode without the timing interval, except that updates occur at timed locations instead of any time the task happens to run. This is the default when an <*interval*> is specified but a <*slew_option*> mode is not.
- `slew` - The same as the `slew` mode without a timing interval, except that the sequence of rapid updates begins on a timing interval. The same as the `skip` mode with a timing interval, except instead of discarding the old data, spin through them and send them to the DAC in quick sequence.

Under the second assumption, updates are sent to the `DACOUT` command in precomputed blocks, and the values must be collated one-for-one with timing intervals of length <*interval*>. If for any reason updates are missed, the corresponding values are discarded so that the remaining updates remain at the intended locations. These options require a timing <*interval*> and one of the following two <*slew_option*> modes.

- `skipalign` - Skipped updates are detected and counted. When each timing interval arrives, one value is discarded for each missed update time, and then the next value is sent out to the DAC.
- `slewalign` - This is almost the same as the `skipalign` option, except that instead of merely discarding extra update values, they are sent in rapid sequence to the DAC.

In the two aligned modes, a consistent strategy of discarding a value for each missed clock interval only works when the supplied data stream is continuous. If not, the strategy fails because many update cycles are missed and counted, potentially causing a large number of values to be discarded. To avoid this problem, specify a <*reset_time*> parameter value that is larger than <*interval*>. If this <*reset_time*> interval passes without receiving any new data for updates, the missed sample count is reset to zero and no data are discarded to start the next burst of activity. Make <*reset_time*> sufficiently large that activity bursts are clearly distinguished from ordinary scheduling delays.

Here are some application examples for using `DACOUT` command modes.

1. **The application requires the most recent value without delay.** Use no <*interval*>, so that there is no added delay. Select the <*slew_option*> `skip`, so that any extra values are considered “old history” and deleted. The most recent update value is sent to the DAC.
2. **Updates consist of a few short pulses, but a minimum pulse width must be guaranteed for each.** Specify an updating time interval with the <*interval*> parameter, and specify the <*slew_option*> `noskip`. Send the pulse level values that the signal must reach, one value at a time or in a block sequence.
3. **Send a block of precomputed signal values, and release them on clocked update intervals.** Specify an updating time interval with the <*interval*> parameter, and specify the <*slew_option*> `skipalign`.
4. **Receive real-time signal data at a high rate, and send selected samples in real-time at a reduced rate.** Specify the <*interval*> between updates, and specify the <*slew_option*> `skip`. More data will arrive than can be used. At each update interval, send only the newest value.
5. **Deliver intermittent tone bursts.** Values are computed and provided one update at a time during a period of activity, but there are no updates during periods of inactivity. Specify the <*interval*> for the updates, with the <*slew_option*> `skipalign` and a <*reset_time*> equal to 10 times <*interval*>. If 10 or more updates are missed, that indicates the end of an activity burst and the count of late samples is cleared to zero.

Note: The voltages on the digital-to-analog converters remain unchanged after `STOP` and `RESET` commands.

Examples

```
DACOUT (P1, 0)
```

Reads the most recent values from pipe `P1`, and writes them to `DAC 0`, using the default strategy `skip` of sending the most current value with minimum delay.

```
DACOUT (V1, 0, 1000)
```

Reads the current value of variable `V1` at time intervals of approximately 1000 microseconds (one millisecond) and updates the value on `DAC 0`.

```
DACOUT (P1, 5,slew)
```

Reads update values from pipe `P1`, and writes them to expansion `DAC 5`, using the strategy `slew` of rapidly posting all backlogged old values.

```
DACOUT (1000, P1, 0)
```

First initializes the output state of `DAC 0` to a value of 1000. After that, reads the most recent values from pipe `P1`, and writes the most current one to `DAC 0`, using the default strategy `skip` of sending only the most recent update value at each opportunity to execute.

```
DACOUT (P1, 1, noskip, 1000)
```

At intervals of 1000 microseconds (1 millisecond) attempts to read a value from pipe `P1`, and, if it finds one, it updates `DAC 1` using the `noskip` strategy of sending the next available value at each opportunity to execute.

```
DACOUT (-1620, P1, 2,skipalign, 1000, 4500)
```

Initializes expansion `DAC 2` to a value of -1620. Tries to read a value from pipe `P1` at intervals of 1000 microseconds (one millisecond) for updating the value at `DAC 2`. If scheduling is slightly delayed, so that an update is missed and multiple values arrive later, the `skipalign` strategy discards old data up to the best-aligned value for the current time interval. If data are not available for more than 4 update intervals of 1000 microseconds, presumes that the activity providing data has ended, and resets the interval counting so that no data are discarded at the start of the next data burst.

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 | FLOAT PIPE | DOUBLE PIPE`

<reference>
An input value corresponding to 0 dB.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<scale>
An integer that represents a scale factor for the output.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

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

Description

`DECIBEL` converts data from *<in_pipe>* to decibel units. *<reference>* is the input value corresponding to 0 dB. Typical reference levels are:

- 32767 (maximum `WORD` value) for `WORD` pipes
- 2147483647 (maximum `LONG` value) for `LONG` pipes
- 1.0 for FLOAT or DOUBLE pipes

<scale> is a scale factor for the output. For the fixed point data types, *<scale>* is an integer multiplier value, typically with values 1, 10, or 100. With *<scale>* equal to 100, for example, 3.25 dB is output as the number 325, preserving two digits of precision in the fraction. *<scale>* can be used with the decimal point specification of `FORMAT` to produce formatted decibel output with zero, one, or two decimal places displayed.

The output of a `DECIBEL` task is given by

```
20 * <scale> * log10 (X/<reference>),
```

where `X` is the value of the input. The output range is limited by the range of the input data type.

The decibel operation is defined only for positive input values. When an input value is zero or negative, the `DECIBEL` command generates a special output value equal to the most negative value for fixed point streams, or special floating point number NAN for floating point streams.

Fixed point conversions are approximate, using a lookup table for speed. For `WORD` data, accuracy is roughly equal to a 14-bit converter. `LONG` evaluations extend the range and add an additional fraction digit to the accuracy. If you need maximum accuracy, use floating point data types.

Examples

```
PIPE  P1  WORD, PDB WORD
DECIBEL (P1, 32767, 100, PDB)
```

Read fixed point values from pipe `P1`, convert data to decibel units with reference level of 32767 defining the 0 dB signal level. For purposes of display, multiply by 100 to preserve two fractional digits in the results, and write results to pipe `PDB`.

```
PIPE  P1  FLOAT, PDB FLOAT
DECIBEL (P1, 100.0, 1.0, PDB)
```

Read floating point values from pipe `P1`. Convert data to decibel units relative to a reference level of 100.0. Do not apply any additional scaling to the results. Send results directoy to pipe `PDB`.

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 | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>

Output pipe for difference data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`DELTA` reads from *<in_pipe>*, computes the forward differences of the data, and puts the result values into *<out_pipe>*. A forward difference is computed by subtracting each *<in_pipe>* value from its successor. That means one difference value is generated for each data value read from *<in_pipe>,* with the exception of the first.

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 differences, and place the results in `P2`.

See Also

DAPL Expressions Chapter

DEXPAND

Define a task that provides output expansion on a digital port.

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 pipe for output data.
`WORD PIPE`

<type>
An optional parameter that specifies the type of output expansion boards.
`WORD CONSTANT`

Description

`DEXPAND` provides output expansion on the digital port. *<output_vector>* is a vector containing a list of the expanded output pins to which data should be sent. *<in_pipe>* is a word pipe that contains data to be sent to the output pins; data must appear in the order of the list in *<output_vector>*. For each data value read from *<in_pipe>*, four words specifying the output pin number and the data are written to *<out_pipe>*.

Optional parameter *<type>* specifies the type of output expansion boards. A value of 0 specifies digital output expansion boards. A value of 1 specifies analog output expansion boards. See the hardware documentation for details on output expansion boards.

`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: 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

```
DEXPAND(P1, (4, 5, 6, 7), OPIPE0)
```

Prepare data from pipe `P1` for output to digital expansion ports 4, 5, 6, and 7, and send the data to synchronous output channel pipe 0.

See Also

DIGITALOUT, DACOUT, ODEFINE, OPTIONS, OUTPORT

DFT

Define a task to calculate selected terms of a Discrete Fourier Transform efficiently.

```
DFT ( <mode>, <N>, <window>, <terms>, <pipein>, <pipeout1> [,<pipeout2>]  )
```

Parameters

<mode>
A constant enumerator in the range 0 to 6 that selects post-processing.
`WORD CONSTANT`

<N>
A number specifying the length of the transform input data block.
`WORD CONSTANT | LONG CONSTANT`

<window>
A constant enumerator or a vector specifying a window operation.
`WORD CONSTANT |`
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<terms>
A vector specifying the frequencies to be computed, by index.
`WORD VECTOR | LONG VECTOR`

<pipein>
Data pipe for a real-valued input data stream.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<pipeout1>, *<pipeout2>*
Data pipes for output results, according to mode.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`DFT` calculates a set of terms of the Discrete Fourier Transform (DFT). The `DFT` command computes a subset of the transform terms computed by the `FFT` command, using a different numerical algorithm.

Similar to the `FFT` command, the *<mode>* parameter identifies the kind of output processing to apply to the transform. In practice, DFT operations for a limited number of terms are rarely used in the inverse direction, and are rarely applied to complex-valued input data. Given these restrictions, the `DFT` command supports only the following subset of the processing modes of the `FFT` command.

Mode parameter	Processing operation
Mode 0	Real data block in, complex value out
Mode 4	Real data block in, power out
Mode 5	Real data block in, magnitude out

Mode 6	Real data block in, magnitude and phase out

The `DFT` command does not apply the same kinds of adjustments to the power computations as the `FFT` command. If you compute, for example, the spectral power at frequency location k, symmetry properties of the transform guarantee that the same amount of power will also appear at location N-k. The `FFT` command takes both terms into account and combines them, so that all of the power in the transform is accounted for. But the `DFT` command will compute either term separately, so it computes the power at only the term that you specify; therefore, a power result is smaller by a factor 2. A similar effect occurs for the mode 5 magnitudes. Because the magnitude is related to the power, the result is smaller by a factor square root of 2 than the magnitude produced by the `FFT` command for real-valued data streams.

The *<N>* parameter specifies the number of terms in each input block. The FFT depends on certain symmetries that result from restricting the block sizes. A DFT does not use these symmetries, and does not need to restrict the block length, so any positive number N is accepted as the input block size, but the larger the block length the more buffer storage is required.

The *<window>* parameter specifies a window operator. Constant enumerators for pre-defined windows are the same ones supported by the `FFT` command. For explicit user-defined window operators, a `VECTOR` can be specified in any basic data type. The data types will be converted internally to the appropriate type for run-time processing. For fixed-point windows, the maximum positive integer value represents the real number 1.0, and smaller integer values proportionally represent fractions. Fractions can be represented using an ordinary decimal notation when using floating-point data types to define the window. See the documentation for the `FFT` command for more information about how to represent a window operator.

Window parameter	**Window type**
0	Rectangular window, i.e., no windowing
1	von Hann window
2	Hamming window
3	Bartlett window
4	Blackman window

The *<terms>* vector specifies the index locations of terms to be computed. DFT index values must be in the range 0 to N-1. A DFT result corresponds to the result you would obtain from an FFT using its *<terms>* value as an index to the FFT output block. Applications typically look for frequencies in a narrow band, so it is most common for the *<terms>* vector to list just a few consecutive index values in order.

The `DFT` command is most useful for computing just a few terms of a forward DFT transform. The `FFT` command is better choice for computing a large number of spectrum terms. The fixed cost of computing the FFT is $K_f \bullet N \bullet \log_2(N)$. If you don't need all of the terms, discard the ones that you don't need. For comparison, the cost to compute $\log_2(N)$ terms using the DFT algorithm is is `Kd•N•log2(N)`. You can see that this has the same form. The theoretical `Kd` is smaller by a factor of 2 and, because of processor and memory access advantages, the DFT tends to gain additional speed. As a quick rule of thumb, if the number of DFT terms in the *<terms>* vector is $2 \bullet \log_2(N)$ or less, the DFT is probably more efficient. For more DFT terms, run a processing rate benchmark to compare the efficiency of the two commands.

The configuration of input and output pipes depends on the operating mode. For complex output or magnitude and phase, both the *<pipeout1>* and the *<pipeout2>* parameters must be specified. For magnitude or power output, only *<pipeout1>* is needed. As a general rule, the data types for output pipes must match the data types for input types. The important exception is mode 4 (power output), when the input data type is `WORD`. For this special case, overflowing the WORD numeric range would occur frequently, and to avoid this the output data type is forced to `LONG` type. A similar fix-up is not available for `LONG` input data type, so it is probably best to avoid `LONG` input data, and convert your `LONG` data streams to `FLOAT` or `DOUBLE` before applying the `DFT` command

Example

```
PIPES  PIN FLOAT, POUTR FLOAT, POUTI FLOAT
VECTOR  TERMS = ( 200, 201, 202, 203, 204, 205, 206 )
. . .
DFT( 0, 1500, 2, TERMS,  PIN, POUTR, POUTI)
```

Use the `DFT` command to detect frequencies in a narrow band. The results of the transform are expressed as complex frequency pairs (mode 0). Each input block from pipe `PIN` has 1500 values. Apply a Hamming window (type 2) to the input data block before applying the DFT analysis. The `TERMS` vector specifies the index values for the seven frequency locations covering the relevant frequency band. For each input block received, 7 complex output results are produced, corresponding to the frequencies

200*2*pi*f / 500, 201*2*pi*f / 500, …, 206*2*pi*f / 500,

where f is the sampling rate used to obtain the input sample stream. The real parts of the seven DFT results are placed into pipe `POUTR`, and the imaginary parts are placed into pipe `POUTI`.

See Also

`FFT`

See the Chapter on “Fast Fourier Transform” for more information about Fourier transforms, window operations, and operating modes.

DIGITALOUT

Define a task that updates a digital output port independent of output clocking hardware.

```
DIGITALOUT ( [init_value,] <in_pipe>, <port_number> [, <slew_option>]
    [, <interval> [, <reset_time>]] )
```

```
DIGITALOUT ( <in_var>, <port_number> [, <interval>] )
```

Parameters

<init_value>
Optional initial output bit pattern for the digital port prior to updates.
`WORD CONSTANT`

<in_pipe>
Source data pipe for first command form.
`WORD PIPE`

<in_var>
Source variable for second command form.
`WORD CONSTANT | WORD VARIABLE`

<port_number>
Hardware address of the digital port.
`WORD CONSTANT`

<slew_option>
Case-sensitive keyword (no quotes) specifying how to process a data backlog.
`skip | slew | noskip | skipalign | slewalign`

<interval>
Optional updating interval in microseconds.
`WORD CONSTANT | LONG CONSTANT`

<reset_time>
Delay time in microseconds before resetting aligned counting modes to zero.
`WORD CONSTANT | LONG CONSTANT`

Description

`DIGITALOUT` updates a digital output port asynchronously by reading from a data stream <*in_pipe*> or from a shared scalar value <*in_var*>*,* and writing the value to a digital port specified by parameter <*port_number*>, bypassing the hardware-controlled timing of an output procedure. This command is frequently applied in control and monitoring applications when unscheduled events determine the digital output values that must be generated, or in specialized signal generation applications where the delays of hardware data buffers for regularly clocked updates are unacceptable, but the extreme precision of hardware output clocking is not necessary.

The <*port_number*> parameter specifies a hardware address for the digital output port. If you use the onboard digital port directly, specify port address 0. If you use the digital port for digital output expansion, specify port address 0, 1, 2, etc. according to the address range configuration of your digital expansion boards.

The behaviors are different for the two kinds of data sources, so there are two command forms.

For the command form with a *shared variable data source*, the value of the variable is used to initialize the output port state when the command starts. Other tasks can change the variable value at any time, and the only way the `DIGITALOUT` command can know whether this has happened is to examine the variable value each time there is an opportunity to execute. If `DIGITALOUT` is running as a high-priority task, this has a danger of soaking up all available CPU capacity, so that other tasks cannot run. To avoid this hazard, you can specify a positive <*interval*> parameter value, in microseconds, setting a time interval between updates. If you do not specify the interval, the DAPL system will select a default.

For the command form with a *pipe data source*, the behaviors of `DIGITALOUT` are a little more complicated. A value might not arrive promptly to set the initial digital output port state, so an optional <*init_value*> parameter can specify the initial output state to apply when the command starts, prior to the first update. After that, new values from the pipe can represent continuous streams or occasional bursts.

The <*slew_option*> and <*interval*> parameters work in combination to tell the `DIGITALOUT` command the strategy for obtaining update data and controlling when they are sent to the digital output port.

If no <*interval*> value is specified, the `DIGITALOUT` command will send the data out as quickly as possible, any time the task scheduler allows the `DIGITALOUT` command to execute, for minimum real-time delays. If more than one value is received, all values but the most recent one are considered "old history." Use one of the following <*slew_option*> choices.

- `skip` - If more than one update value is available, ignore old values and post the most current one to minimize real-time delays. Use this option if output levels are what matters and transitions are noise.
- `slew` - Almost the same as the `skip` option, but use all data, and spin through all of the updates quickly. Most digital devices can track these transitions without difficulty. This is the default behavior with no <*slew_option*> specified. Use this option if observing transitions is important to track state changes.

If you specify an <*interval*> value, the `DIGITALOUT` command will try to send updates at the specified interval in a regular sequence. Timing starts with the arrival of the first update sample. The interval is determined by a clock that is independent of the input and output configurations. Regular updating intervals are possible, but not with the speed and timing accuracy of hardware control. There is some uncertainty about the exact time of an update when other tasks coexist with the `DIGITALOUT` task at equal priority

There are two possible assumptions about the way that data will arrive at the `DIGITALOUT` task when an <*interval*> value is specified. Under the first assumption, the data will arrive in real time, one value for each update interval. If multiple values are received, this indicates that a delay occurred somewhere. Only the most recent value means anything, and the rest are "old history." Use one of the following three <*slew_option*> modes.

- `noskip` - When a timing interval arrives, if an update value is available, send the first one to the digital output port. Never discard any update data. Use this option when a precomputed data block is provided and it is important to reach and hold each level without missing anything.
- `skip` - Discard old data and send only the most recent value to the digital output port. This is the same as `skip` mode without the timing interval, except that updates occur at timed locations instead of any time the task happens to run.
- `slew` - The same as the `slew` mode without a timing interval, except that the sequence of rapid updates begins on a timing interval. The same as the `skip` mode with a timing interval, except instead of discarding the old data, spin through them and send them to the digital output port in quick sequence. This is the default when an <*interval*> is specified but a <*slew_option*> mode is not.

Under the second assumption, updates are sent to the `DIGITALOUT` command in precomputed blocks, and the values must be collated one-for-one with timing intervals of length <*interval*>. If for any reason updates are missed, the corresponding values are discarded so that the remaining updates remain at the intended locations. These options require a timing <*interval*> and one of the following two <*slew_option*> modes.

- `skipalign` - Skipped updates are detected and counted. When each timing interval arrives, one value is discarded for each missed update time, and then the bit pattern is sent out to the digital port.
- `slewalign` - This is almost the same as the `skipalign` option, except that instead of merely discarding extra update values, they are sent in rapid sequence to the digital port.

In the two aligned modes, a consistent strategy of discarding a value for each missed clock interval only works when the supplied data stream is continuous. If not, the strategy fails because many update cycles are missed and counted, potentially causing a large number of values to be discarded. To avoid this problem, specify a <*reset_time*> parameter value that is larger than <*interval*>. If this <*reset_time*> interval passes without receiving any new data for updates, the missed sample count is reset to zero and no data are discarded to start the next burst of activity. Make <*reset_time*> sufficiently large that activity bursts are clearly distinguished from ordinary scheduling delays.

Here are some application examples for using `DIGITALOUT` command modes.

1. **The application requires the most recent bit values as soon as possible.** Use no <*interval*>*,* so that there is no added delay. Select the <*slew_option*> `skip`, so that any extra values are considered "old history" and deleted. The most recent update value is sent to the digital output port.
2. **Updates consist of a few short pulses, but a minimum pulse width must be guaranteed for each.** Specify an updating time interval with the <*interval*> parameter, and specify the <*slew_option*> `noskip`. Send the bit patterns one at a time or in a block of update values.
3. **Send a block of pre-computed bit patterns, and release them on clocked update intervals.** Specify an updating time interval with the <*interval*> parameter, and specify the <*slew_option*> `skipalign`.
4. **Deliver intermittent bursts of bit pattern changes.** Values are computed and provided one update at a time during a period of activity, but there are no updates during periods of inactivity. Specify the <*interval*> for the updates, with the <*slew_option*> `skipalign` and a <*reset_time*> equal to 10 times <*interval*>. If 10 or more updates are missed, that indicates the end of an activity burst and the count of late samples is cleared to zero.

Note: The output levels on the digital output ports remain unchanged after `STOP` and `RESET` commands.

Examples

```
DIGITALOUT (P1, 0)
```

Reads the most recent values from pipe `P1`, and writes them to digital port 0, using the default strategy `skip` of sending the most current value with minimum delay.

```
DIGITALOUT (V1, 0, 1000)
```

Reads the current bit pattern of variable `V1` at time intervals of approximately 1000 microseconds (one millisecond) and updates the value on digital port 0.

```
DIGITALOUT (P1, 5,slew)
```

Reads update values from pipe `P1`, and writes them to expansion digital port 5, using the strategy `slew` of rapidly posting all backlogged old values.

```
  DIGITALOUT ($FF00, P1, 0)
```

First it initializes the output state of digital port 0 to a value of $FF00, so that the 8 high-order bit positions are active-high and the 8 low-order bit positions are inactive-low. After that, it reads the available values from pipe `P1`, and writes the most current one to digital port 0, using the default strategy `skip` of sending only the most recent update value at each opportunity to execute.

```
  DIGITALOUT (P1, 1, noskip, 1000)
```

At intervals of 1000 microseconds (1 millisecond) attempts to read a value from pipe `P1`, and, if it finds one, it updates expansion digital port 1 using the `noskip` strategy of sending the next available value at each opportunity to execute.

```
  DIGITALOUT ($004C, P1, 2,slewalign, 1000, 4500)
```

Initializes expansion digital port 2 to the bit pattern "0000 0000 0100 1100". Tries to read a value from pipe `P1` at intervals of 1000 microseconds (one millisecond) for updating the value at digital port 2. If scheduling is slightly delayed, so that an update is missed and multiple values arrive later, the `slewalign` strategy posts all old data in rapid succession up to the best-aligned value for the current time interval. If data are not available for more than 4 update intervals of 1000 microseconds, presumes that the activity providing data has ended, and resets the interval counting so that no data are discarded at the start of the next data burst.

See Also

`DACOUT`, `ODEFINE`

DISPLAY

Display selected system information.

DISPLAY *<item>*

DISP *<item>*

D *<item>*

Parameters

<item>

One of the following keywords:

ALLSYMBOLS	COMMANDS	CMDSPEC	CONSTANTS	CPIPES
DAPLNAME	DHARDWARE	DMODEL	DREVISION	DVARIANT
DVERSION	EMSG	ERRORS	ENUM	EVMSG
EVNUM	HMEMORY	ICOUNTS	INPUTS	KNLLOG
MEMORY	MODULES	OCOUNTS	OEMID	OPTIONS
OUTPORT	OUTPUTS	OVERFLOWQ	PIPES	PROCEDURES
SERIAL	STARTS	STRINGS	SYMBOLS	SYSLOG
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. Acceptable abbreviation: ALL.

CMDSPEC <cmdname> [option]

DISPLAY CMDSPEC shows the parameter list specification associated with the cmdname processing command. To see the same information in a restructured internal form, specify the type option. To see the same information in a more structured layout, specify the typetree option.

COMMANDS

DISPLAY COMMANDS lists information about the custom commands which have been loaded. Acceptable abbreviations: COMMAND, COM.

CONSTANTS

DISPLAY CONSTANTS lists the names, data types, and values of all shared constants. Acceptable abbreviations: CONSTANT, CONST.

`CPIPES`

`DISPLAY CPIPES` lists information about user-defined and pre-defined communication channel pipes. Acceptable abbreviations: `CPIPE`, `CP`.

`DAPLNAME`

`DISPLAY DAPLNAME` displays the DAPL operating system name such as "DAPL3000".

`DHARDWARE`

`DISPLAY DHARDWARE` displays DAP hardware information (not available on all models).

`DMODEL`

`DISPLAY DMODEL` displays the DAP model name, such as DAP5400a/627.

`DREVISION`

`DISPLAY DREVISION` displays the DAP hardware revision number such as "1".

`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`.

`DVERSION`

`DISPLAY DVERSION` displays DAPL operating system version such as "1.00".

`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 that can be found in the Error Messages supplement to this manual. 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.

`ERRORS <number>`

`DISPLAY ERRORS` prints a description associated with the error message `number`, as provided by the DAPL system in the `$SysOut` text pipe or the `ERRORQ` buffer after error detection. If `number` is omitted, all error message texts are listed. Acceptable abbreviation: `ERROR`.

`EVMSG`

`DISPLAY EVMSG` displays all error messages, one or more, that are associated with the event ID of the most recent queued error. The messages are replaced when the next error is queued, or cleared by a `RESET` command.

`EVNUM`

`DISPLAY EVNUM` displays the event ID associated with the most recent queued error. The event ID is a number in a hexadecimal notation. The event ID is overwritten when the next error is queued, or cleared by a `RESET` command.

`HMEMORY`

DISPLAY HMEMORY displays the number of bytes of shared heap memory and total available data memory. Acceptable abbreviations: HMEM, HM.

ICOUNTS
DISPLAY ICOUNTS prints the current input configuration sample count. Acceptable abbreviations: ICOUNT, IC.

INPUTS
DISPLAY INPUTS lists the input procedures. Acceptable abbreviations: INPUT, IN.

KNLLOG
DISPLAY KNLLOG prints the messages currently stored in the system eventlog buffer.

MEMORY
DISPLAY MEMORY prints the number of bytes currently in use and the total memory available. Acceptable abbreviations: MEM, M.

MODULES
DISPLAY MODULES lists information for all installed modules. Acceptable abbreviations: MODULE, MOD.

OCOUNTS
DISPLAY OCOUNTS prints the current output configuration update count. Acceptable abbreviations: OCOUNT, OC.

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. Acceptable abbreviations: OPTIONS, OPT.

OUTPORT
DISPLAY OUTPORT prints the current configuration of the output ports. Acceptable abbreviation: OUTPORTS.

OUTPUTS
DISPLAY OUTPUTS lists the output procedures. Acceptable abbreviations: OUTPUT, OUT.

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 **14**. Acceptable abbreviations: OVERFLOW, OVER.

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 which have not yet been processed by all tasks reading data from the pipe. Acceptable abbreviations: PIPE, P.

`PROCEDURES`

`DISPLAY PROCEDURES` lists the names, type, and activity of all input, output, and processing configurations which are defined. Acceptable abbreviations: `PROCEDURE`, `PROC`.

`SERIAL`

`DISPLAY SERIAL` displays the DAP serial number.

`STARTS`

`DISPLAY START` lists the currently active start groups, and the names of the input, output and processing procedures that are included in each group. Acceptable abbreviation: `START`.

`STRINGS`

`DISPLAY STRINGS` lists the names, data type, and values of all shared strings. Acceptable abbreviation: `STRING`, `STR`.

`SYMBOLS`

`DISPLAY SYMBOLS` prints all user-defined names known to the system. Acceptable abbreviations: `SYMBOL`, `SYM`.

`SYSLOG   [<n> [, <m>]]   [ID=<hex> [count]]`

`DISPLAY SYSLOG` and its variations display error and warning messages stored in the system event log. Locations in the log are indexed starting at 0 for the newest message, and each successive index 1,2, etc. will access an older message. To specify an index or a range of index values, use the index parameters `n` and `m`, where `m` is greater than `n`. If you know a message's hexadecimal ID code, instead of specifying an index relative to the head of the list, you can specify the ID of the first message and select the number of messages to display. They will be presented in the order oldest to newest.

`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. Acceptable abbreviations: `TMEM`, `TM`.

`TRIGGERS`

`DISPLAY TRIGGERS` displays the names, operating modes, and properties of all defined software triggers. Acceptable abbreviations: `TRIGGER`, `TRIG`.

`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 **14**. Acceptable abbreviations: `UNDERFLOW`, `UNDER`.

`VARIABLES`

`DISPLAY VARIABLES` displays the names, data types and current values of all shared variables. Acceptable abbreviations: `VARIABLE`, `VAR`.

`VECTORS`

`DISPLAY VECTORS` displays the names, data types and lengths of user-defined vectors. Acceptable abbreviations: `VECTOR`, `VEC`.

`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 that can be found in the Error Messages supplement to this manual. 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 | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<region1>
A specification for the triggering condition.
`REGION`

<trigger>
The trigger asserted whenever a difference satisfying *<region1>* is found.
`TRIGGER`

<region2>
An optional specification of holdoff (hysteresis) to avoid spurious re-triggering.
`REGION`

Description

`DLIMIT` computes differences between consecutive data values from pipe *<in_pipe>*. When a difference satisfies the *<region1>* condition, an event is asserted in *<trigger>*. For example,

```
OUTSIDE, -255.0, 255.0
```

This specifies reporting an event by asserting *<trigger>* when the amount of change from one sample to the next is greater than 255 in magnitude. The data types of region specifications and the data stream must match.

<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>* holdoff condition. Specifying *<region2>* can prevent multiple triggering when a signal change covers more than one sample. *<region2>* can be the same as region *<region1>*, which means that the rate of change must decrease to less than the original triggering condition, but spurious triggering events can still occur if the data set is noisy. See Chapter **16** 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, depending on how much unprocessed data are in each task's data buffers.

Examples

```
  DLIMIT (IP5, OUTSIDE, 0, 0, T2)
```

Scan input channel pipe `IP5` for any changes between consecutive sample values. Record each change event in trigger `T2`.

```
  DLIMIT (P1, OUTSIDE -250.0, 250.0, T1, OUTSIDE -60.0, 60.0)
```

Scan `P1` for changes less than –250.0 or greater than 250.0. After an event is asserted in trigger `T1`, the trigger cannot be re-asserted until the rate of change first reduces to less than 60.0 in magnitude.

See Also

`LIMIT`, `LOGIC`

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.

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`.

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
`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. The communications pipes `$SysIn` , `$SysOut`, 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 which 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`

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. In addition to the basic transform, the most commonly required pre-processing and post-processing operations also are provided.

All variants of the `FFT` command use the same form for the three initial parameters.

```
FFT( <mode>, <size>, <window>, … )
```

The final two to four parameters specify the input and output data streams. The number of pipes required depends on the value of the mode parameter.

Mode 0 real ---> complex	`FFT( 0, … , Re_in, Re_out, Im_out)`
Mode 1 complex ---> complex	`FFT( 1, … , Re_in, Im_in, Re_out, Im_out)`
Mode 2 reverse complex ---> real	`FFT( 2, … , Re_in, Im_in, Re_out)`

Mode 3 reverse complex ---> complex	`FFT( 3, … , Re_in, Im_in, Re_out, Im_out)`
Mode 4 real ---> power	`FFT( 4, … , Re_in, Pow_out)`
Mode 5 real ---> magnitude	`FFT( 5, … , Re_in, Mag_out)`
Mode 6 real ---> polar	`FFT( 6, … , Re_in, Mag_out, Phs_out)`

The data types of all input and output streams must match, with the one exception of the output power mode 4, which allows 32-bit `LONG` output to be used with 16-bit `WORD` input because of a large output range. The calculations are specialized for the type of processed data:

• *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. 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.

Mathematically, forward and reverse transforms are inverse operations, provided that an additional factor of `1/N` is applied somewhere, where `N` is the number of terms in an input data block. The `FFT` command applies this scaling factor to all forward transforms (in all modes except 2 and 3).

Operating Modes Without Post-Processing

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 basic 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>* for modes that do not use it.

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, by using using *<mode>=2*. Omit parameter *<pipeoutI>* from the parameter list when it is not needed.

Operating Modes With Post-Processing

The other three operating modes perform transforms in the forward direction (from time to frequency) and then apply post-processing operations to make spectrum analysis easier. The post-processing operations are meaningful only for the case of forward transforms of real-valued inputs, so the parameter *<pipeinI>* is not used. Processed results are returned, rather than the raw transform data.

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:

- **Mode 4 - Power.** Signal power for each frequency is calculated by squaring the real and imaginary terms and combining the two squared values. The part of the signal power artificially reflected into image frequencies above the Nyquist frequency of the sampling is recombined. Because of squaring, the range of output values can be large, so be careful with data scaling.

- **Mode 5 - Magnitude.** Basically, this is the same thing as mode 4 power, but with an additional square root operation. This tends to eliminate scaling problems, but requires extra computing time.

- **Mode 6 – Magnitude and phase angle.** The magnitude term is the same as for mode 6. In addition, the ratio of the real and imaginary terms from the original transform is analyzed to determine the phase angle associated with the frequency. The phase angle for the artifact frequency above the Nyquist frequency is equal in magnitude and opposite in sign, and is ignored. The angle is expressed in radians. For angle terms in fixed-point data types, the range of the integer values corresponds to the angle range -pi to +pi.

FFT Modes with Data Post-Processing

Description	*<pipeinR>* (signal)	*<pipeinI>*	*<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.

Data Block Sizes

For all operating modes, the second `FFT` command parameter *<m>* indirectly specifies the size of the input data blocks. The lengths of the data blocks are a power of 2, so the actual size is 2 to the power of *<m>*. The value of *<m>* is specified in the task parameter list, as summarized in the following list.

`<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

Windowing

The `<window>` parameter is a constant or a vector specifying the window operation used by the transform. Windowing provides a term-by-term scaling of the original input data before applying the transform. This sometimes has the advantage of reducing spectrum distortions that result from chopping the data into separate blocks for analysis, but also causes some spreading and reduction of peak values. If these distortions are important, they must be taken into consideration when analyzing the results.

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:

`Window types`

`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.

Aliasing

To avoid aliasing, the sampling rate must be chosen so that frequencies of all relevant phenomena are below the Nyquist frequency appearing at term `N/2`, where `N` is the number of terms in a complete spectrum block. 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, which is why the post-processing modes return only half-length blocks.

The symmetry phenomenon works both ways, however, and can lead to some surprises for the unwary. The transform formulas have no way to know from the sampled data whether the original signal was above or below the Nyquist frequency, so both frequencies appear in the spectrum. For example, suppose the real-valued input data block for a 256-term FFT has a cosine wave with peak value 10000 and frequency 6*(2 * π / 256) radians per second, sampled at 1/256 second intervals. Examining a mode 0 FFT, one would expect to see a frequency peak of magnitude 10000 in location 6 of the raw transform data, right? Well, maybe not. Half of the wave appears at location 6, but the other half appears in the symmetric image at location 251. The chapter on the Fast Fourier Transform in this manual discusses the aliasing phenomena in more detail.

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 6, 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 **18** “Fast Fourier Transform” for more information about frequency spectra, sampling, aliasing effects, 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. You cannot use the `FILL` command to insert data into the reserved `$SysIn` or `$SysOut` communication pipes.

Values are converted to the appropriate data type for placing into the pipe. The initializer values must be compatible with the data type of the pipe.

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 an interval.

FINDMAX *(<in_pipe>, <n>, <region>, <out_pipe> [, <ix_pipe>] [, <ix_trig>])*

Parameters

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

<n>
The number of values in each block from *<in_pipe>*.
`WORD CONSTANT`

<region>
The region that determines which data positions are scanned.
`REGION` specifier

<out_pipe>
Output data pipe for the maximum value observed.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<ix_pipe>
Optional output pipe for the block index of the maximum value.
`WORD PIPE | LONG PIPE`

<ix_trig>
Optional software trigger for the location of the maximum value.
`TRIGGER`

Description

`FINDMAX` reads blocks of *<n>* values from the *<in_pipe>*. The maximum value observed in the selected terms is copied to *<out_pipe>*, so there is one output value per input block. The data types of *<in_pipe>* and *<out_pipe>* must match. Terms whose indices are in *<region>* are scanned. Terms are indexed from *0* to *<n>-1*. If relevant peaks can occur at any location, specify the full range 0 to *<n>-1*.

Note: While a `REGION` specification is applied to data values in most other DAPL commands, the `FINDMAX` command applies it to the index of the data values.

One or two optional additional parameters can be specified to record the position of the maximum value observed within the data block. If *<ix_pipe>* is specified, the index for the maximum value is placed into this pipe. The indexing is the same scheme used by the `REGION` specification to select data for scanning. If *<ix_trig>* is specified, an event will be recorded in this trigger indicating the location in the original data stream where the maximum value was detected. You can use the trigger in a command such as `WAIT` to recover data at this same relative position in any data stream. If neither optional parameter is specified, information about the location of the peak is discarded.

This command can be used after an FFT in mode 4 or mode 5 to determine the location of a peak in the frequency spectrum. If the particular peak of interest is not the highest one, the peak of interest can be isolated using *<region>*.

Examples

```
   FINDMAX (P1, 512, INSIDE, 50, 100, P2)
```

Read blocks of 512 values from pipe P1. Place the largest value observed from indexed locations 50 to 100 into pipe P2.

```
   FINDMAX (P1, 128, INSIDE, 0, 127, IXTRIG, P3)
```

Read blocks of 128 values from pipe P1. Place the largest value anywhere in the block into pipe P2. Place the index of the largest value into P3. Also assert an event at the corresponding location using trigger IXTRIG, so that tasks can locate corresponding data in other signals.

See Also

FFT, WAIT

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   PD1 FLOAT, PD2 FLOAT
FIRFILTER (PD1, FVEC2, 0, .7071, 4, 0, PD2, 1024, 4096)
```

Apply a filter of arbitrary length to data from float pipe `PD1`, 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 `PD2`, 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.

Decimation	**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

Decimation	Passband Limit (% Nyquist)	Stopband Limit (% Nyquist)	Data Type	Total taps	Delay samples (after decimation)
8	6.25	9.38	`Word`	149	9.25
			`Long` `Float` `Double`	207	12.87
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

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 | LONG CONSTANT`

<out_pipe>
The pipe to which the number of assertions is written.
`WORD PIPE | LONG PIPE`

Description

`FREQUENCY` counts how many trigger assertions occurred in each interval of *<length>* samples. The number of assertions is sent to *<out_pipe>*, one number for each interval.

Example

```
FREQUENCY (T1, 100, P1)
```

Send the number of trigger assertions occurring in the intervals 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.

Example

```
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.

Example

```
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:

```
*** DAPL3000 Interpreter [3.0 R1/626] Serial# 76000 ***
```

`3.0` is the DAPL version number. `R` is the Data Acquisition Processor product code. `1` is the hardware version code. `626` is the model number. `76000` 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 | FLOAT PIPE | DOUBLE PIPE`

<count>
: A value that specifies the size of blocks to be scanned.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe1>
: Output data pipe for the maximum value from each block.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe2>
: If specified, the pipe into which the index of the maximum value is placed.
`WORD PIPE | LONG PIPE`

Description

`HIGH` scans blocks of size *<count>* samples for maximum values. The maximum of each block is placed in the pipe*<out_pipe1>*. If pipe*<out_pipe2>* is specified, indices describing the location of each maximum value are placed in *<out_pipe2>*. The index numbers range from 0 to *<count>*-1.

When there are multiple values equal to the same maximum, the location of the first maximum is reported.

Note: The `FINDMAX` command is a specialized variant of `HIGH` that can restrict the search to specified index positions.

Examples

```
HIGH (IPIPE3, 100, PHIGH)
```

Read blocks of 100 values from input channel pipe 3 and send the maximum data value in each block to pipe `PHIGH`.

```
HIGH (P2, 1000, PMAX, PIX)
```

Read blocks of 1000 values from pipe `P2`. Send the maximum data value in each block to pipe `PMAX`, and send the index of the maximum for each block to pipe `PIX`.

See Also

`FINDMAX`, `LOW`, `RANGE`, `VARIANCE`, `STDDEV`

HTRIGGER (for input sampling)

Specify the operating mode of the hardware input trigger signal.

HTRIGGER *<type>*

Parameters

<type>

A keyword.

`ONESHOT | GATED | OFF`

Description

Including `HTRIGGER` in an input sampling configuration determines the manner that input sampling responds to the external hardware input triggering signal. The *<type>* keyword specifies the operating mode. It must be one of the following:

- `OFF` -- The default when the `HTRIGGER` command is omitted from the input configuration. The DAP does not respond to the external input triggering hardware signal. Sampling can start immediately.
- `ONESHOT` – Sampling can begin when the hardware signal for triggering input sampling goes active. Once the trigger level is detected and sampling begins, the DAP no longer responds to the external hardware signal.
- `GATED` – Sample clocking proceeds while the hardware trigger signal is at the active level, and is suspended while the hardware trigger signal is at the inactive level.

The hardware trigger signal is described in the Data Acquisition Processor hardware documentation.

Example

```
HTRIGGER  ONESHOT
```

The Data Acquisition Processor ignores the sampling clock until the external hardware signal for input sampling goes active. After sampling starts, the level of the triggering signal does not matter.

See Also

`CLCLOCKING`, `CLOCK`, `SAMPLE`, `HTRIGGER for output`

HTRIGGER (for output updating)

Specify the operating mode of the output hardware trigger signal.

HTRIGGER *<type>*

Parameters

<type>

A keyword.

`ONESHOT | GATED | OFF`

Description

Including the `HTRIGGER` in an output updating configuration determines the manner that output updating responds to the external hardware output triggering signal. The *<type>* keyword specifies the operating mode and must be one of the following:

- `OFF` -- The default when the `HTRIGGER` command is omitted from the output configuration. The DAP does not respond to the external hardware signal. Updating can begin when sufficient data are available in the output buffers.
- `ONESHOT -` Updating can begin when the hardware output triggering signal goes active. Once the trigger level is detected and updating is underway, the DAP no longer responds to the external hardware signal.
- `GATED` – Update clocking proceeds while the hardware trigger signal is at the active level, and is suspended while the hardware trigger signal is at the inactive level.

The hardware trigger signal is described in the Data Acquisition Processor hardware documentation. It is not recognized until the `OUTPUTWAIT` condition is satisfied.

Example

```
HTRIGGER   GATED
```

Specify that the hardware trigger allows output updates to be delivered when the external trigger level is active, and suspends updates when the external trigger level is inactive.

See Also

`CLCLOCKING`, `CLOCK`, `UPDATE`, `OUTPUTWAIT`, `HTRIGGER for input`

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`

INTERP

Define a task that computes a function value using a lookup table.

INTERP *(<in_pipe>, <x_vector>, <y_vector>, <out_pipe>)*

Parameters

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

<x_vector>
: A list of monotonically increasing abscissa (input) values.
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<y_vector>
: A list of corresponding ordinate (output) values.
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

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

Description

`INTERP` evaluates a function by linear interpolation on a lookup table. This command is especially useful for sensor linearization and calibration.

<x_vector> and *<y_vector>* represent a set of ordered pairs of numbers *(x, y)* that describe a function. The numbers of elements in *<x_vector>* and *<y_vector>* must be the same, the values in the *<x_vector>* must be in a strictly ascending sequence, and the input and output data must be the same data type matching the input and output data pipes.

For each value in *<in_pipe>*, an `INTERP` task searches for the nearest bounding 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, PINTRP)
```

Read data from pipe `P1`, interpolate an output value based on the two-piece function in the table defined by vectors `X` and `Y`, and send the results to pipe `PINTRP`.

See Also

THERMO

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 detects values satisfying a specified level condition.

LIMIT *(<in_pipe>, <region1>, <trigger> [,<region2>])*

Parameters

<in_pipe>

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<region1>

A specification of the triggering level condition.
`REGION`

<trigger>

The trigger asserted when values in the specified range are found.
`TRIGGER`

<region2>

An optional specification for holdoff (hysteresis) to avoid spurious re-triggering.
`REGION`

Description

A `LIMIT` task scans input data for values that satisfy the condition specified by *<region1>*. When such a value is found, *<trigger>* is asserted. For example,

```
INSIDE, 5000.0, 7500.0
```

This condition specifies reporting an event by asserting *<trigger>* when the level of the signal enters the region 5000 to 7500.

An optional second region, *<region2>*, can specify 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 when a signal changes slowly or when events occur in bursts that only need to be recognized once. *<region2>* often set the same as *<region1>*, although this is not required. The data types of the region specifications and the input data stream must match.

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, depending on how much unprocessed data are in each task's data buffers.

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; asserting trigger `T1` whenever one is found. After an assertion, do not trigger again until after a value not in the range 5500 to 5600 is detected.

```
  LIMIT (P1, OUTSIDE,-20000,20000, T2, OUTSIDE,-10000,10000)
```

Scan data in pipe `P1` for levels 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 while data from `P1` remain outside the range -10000 to 10000.

See Also

`DLIMIT`, `LOGIC`

LOADING

Define a task to simulate data flow and CPU loading for latency studies.

LOADING *(<in_pipe>, <blockin>, <avgCPU>, <devCPU>, <out_pipe>, <blockout>)*

Parameters

<in_pipe>
Input data stream to process.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<blockin>
Size of input data blocks to process.
`WORD CONSTANT`

<avgCPU>
The average CPU consumption in microseconds for each input block received.
`WORD CONSTANT | LONG CONSTANT`

<devCPU>
The range of deviation, plus or minus microseconds, in CPU consumption.
`WORD CONSTANT`

<out_pipe>
Output data stream.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<blockout>
Size of output data blocks to produce.
`WORD CONSTANT`

Description

The `LOADING` command provides a means of simulating CPU consumption and data flow in a DAPL configuration before the actual processing has been developed. Values are taken from the *<in_pipe>* in multiples of *<blockin>* specified. For many systems the *<blockin>* size is 1 because values arrive singly, but some systems will read a bank of sensor or feedback signals to process together in one operation. After processing, results are placed into the *<out_pipe>* in groups of size *<blockout>*.

The contents of the data streams are numerically valid but meaningless to the simulation. The purpose of this command is not to produce useful calculated results; rather it is to provide a means for determining how the DAPL system acts under load given the amounts of data, block sizes and sampling rates specified. Control systems using sampled data expect delays of one sampling interval, and ordinarily complete all computations in time for the next sample to arrive. Latency longer than one sampling interval can mean data backlogs. Missed deadlines can be a serious difficulty for some time-critical systems. With some pessimistic assumptions about process timing, a simulation can determine whether response latency is likely to be a problem, unlikely to be a problem, or completely guaranteed to never be a problem.

The computation is not relevant, and only the time delays are important to the simulation. Time delays and variations are specified in units of microseconds, with the *<avgCPU>* parameter specifying the average computing time and the *<devCPU>* parameter specifying the largest random deviations. The *<avgCPU>* value must be positive, and the *<devCPU>* value must be less than *<avgCPU>*. The distribution of computing time is triangular, so there will be many values clustered close to the *<avgCPU>* level but very few that are near the limits at *<devCPU>* microseconds above or below the average level.

See the STATISTICS command for information about analyzing the delays in the processing network.

Example

```
LOADING (P1,10, 1000,30,  P2,1)
LOADING (P2,1, 50,20,  P3,1)
```

Simulate the timing of a cascade of two processing commands. The first command takes data blocks of 10 samples from source pipe P1, uses 970 microseconds to 1030 microseconds of processing time for each block, and places single output results into destination pipe P2. The second command receives each value from the first command, takes from 30 to 70 microseconds to compute a new value, and places it into destination pipe P3. Run a processing configuration with this sequence to see if the propagation delays can satisfy timing requirements.

See Also

STATISTICS

LOGIC

Define a task that asserts a trigger when data bits match a specified pattern.

LOGIC *(<in_pipe>, <xor>, <and>, <trigger>)*

Parameters

<in_pipe>
The pipe from which bit patterns are taken and tested.
`WORD PIPE | LONG PIPE`

<xor>
A bit pattern number identifying bit positions that are active-low.
`WORD CONSTANT | WORD VARIABLE | LONG CONSTANT | LONG VARIABLE`

<and>
A bit pattern number selecting bit positions to test.
`WORD CONSTANT | WORD VARIABLE | LONG CONSTANT | LONG VARIABLE`

<trigger>
The trigger that is asserted when bits match the specified conditions.
`TRIGGER`

Description

`LOGIC` asserts a trigger when selected bits from *<in_pipe>* match conditions specified by the *<xor>* and the *<and>* parameters. The values in *<in_pipe>* can be obtained by sampling the digital input port, and can be used to detect external alarm events. There is a built-in hysteresis behavior, and `LOGIC` asserts *<trigger>* only once each time the conditions are satisfied.

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.

Usually, it is easiest to specify the bit tests using a hexadecimal notation. If more than one bit is tested, `LOGIC` requires all the selected bits to satisfy the triggering condition. If the Boolean result is 1, *<trigger>* is asserted. After that, the `LOGIC` task must receive an input value for which the computed Boolean is 0 before it will trigger again.

If you need more complicated bit detection logic, use a DAPL expression before applying this command.

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 `TRIGGER` operating mode having a nonzero `STARTUP` property.

Examples

```
   LOGIC (IPIPE3, $0004, $0004, 0, T1)
```

Assert trigger T1 when bit 2 of a bit pattern from input channel pipe 3 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

DLIMIT, LIMIT, 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 | FLOAT PIPE | DOUBLE PIPE`

<count>

A value that specifies the size of blocks to be scanned.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe1>

Output data pipe for the maximum value from each block.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe2>

If specified, the pipe into which the index of the maximum value is placed.
`WORD PIPE | LONG 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.

`LOW` scans blocks of size *<count>* samples for minimum values. The minimum of each block is placed in the pipe*<out_pipe1>*. If pipe*<out_pipe2>* is specified, indices describing the location of each minimum value are placed in *<out_pipe2>*. The index numbers range from 0 to *<count>*-1.

When there are multiple values equal to the same minimum, the location of the first maximum is reported.

Note: The `LOW` command is the same as `HIGH` except for seeking the lowest (most negative) instead of the highest (most postive) value.

Examples

```
   LOW (IPIPE3, 100, PLOW)
```

Read blocks of 100 values from input channel pipe 3 and send the minimum value of each block to pipe `PLOW`.

```
   LOW (P2, 1000, P3, PIX)
```

Read blocks of 1000 values from pipe `P2`. Send the minimum value from each block to pipe `P3`, and send the position index of the minimum in each block to pipe `PIX`.

See Also

`FINDMAX`, `HIGH`, `RANGE`

MASTER

Configure an input or output configuration's sampling clock as a master clock.

```
MASTER    [ DIVIDE = <n> ]
```

Parameters

<n>

Integer divisor for clock rate

Description

The `MASTER` command is used in systems with multiple Data Acquisition Processors. It configures the designated Data Acquisition Processor to serve as a master that generates clocking or updating timing signals to synchronize all slave Data Acquisition Processors.

To make the DAP act as a master for sampling activity on all slave Data Acquisition Processors, include the `MASTER` command in the input sampling configuration. To make the DAP act as a master for output updating activity on all slave Data Acquisition Processors, include the `MASTER` command in the output updating configuration.

When using a very fast Data Acquisition Processor as the `MASTER` for input sampling, the rates may be faster than some slower DAP models can operate. If you specify the optional *DIVIDE=<n>* parameter, a clock divider reduction will be applied to the sampling or updating signal to reduce the rate. The value of *<n>* must be 1, 2, 3 or 4. The default value is 1. The default will work with all DAP models, but other values are not supported by all models. Check your DAP hardware manual before using this feature.

See Also

`CLOCK for input`, `CLOCK for output`, `SLAVE`, `IDEFINE`, `ODEFINE`

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 reintereprets 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 as valid elements of the output pipe data type, 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.

Conversion Table

<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 of various data types, and place the data consecutively into pipe P4 .

```
MERGE (P5,P6,P7,$BinOut)
```

Transfer data from pipes P5, P6, and P7 to the binary output communication pipe; this puts data data streams into a multiplexed sequence and transfers them to the host computer.

See Also

BMERGE, BMERGEF, MERGEF, SEPARATE, SEPARATEF

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>, ... , <in_pipe_n-1>

Data pipes providing input data.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>

Output pipe for merged data.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`MERGEF` merges data from several pipes `<in_pipe_0>` through `<in_pipe_n-1>` into a single pipe `<out_pipe>`. An identifying flag is included with each value, specifying which pipe provided the value. `MERGEF` differs 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. Application must sort through the data tags to reconstruct the original signals.

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, in the natural data type of *<out_pipe>*. The flag value indicates the source. The flag is followed by the value.

If the data types of an input pipe and and output pipe do not match, the value is subject to the same conversion rules described for the `MERGE` command. For most efficient operation, avoid mixing data types.

`MERGEF` is particularly useful for transmitting binary data to the PC when data values arrive sporadically and at different rates from different data sources.

`MERGEF` is the inverse of `SEPARATEF`.

See Also

`BMERGE`, `BMERGEF`, `MERGE`, `SEPARATE`, `SEPARATEF`

NMERGE

Define a task that merges data in groups.

NMERGE *(<n_0>, <in_pipe0>, [<n_1>, <in_pipe1>, ...] <out_pipe>)*

Parameters

<n_0>, …, <n_n-1>

Number of data values to transfer from input pipes.
`WORD CONSTANT | LONG CONSTANT`

<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

`NMERGE` reads data from one or more input pipes and places the data into an output pipe. The data are transferred in groups, but unlike the `BMERGE`, the source data pipes *<in_pipe_0>* through *<in_pipe_n-1>* are not restricted to being the same block size or the same data type. `NMERGE` transfers *<n_0>* data values from *<in_pipe_0>* to *<out_pipe>*, then transfers *<n_1>* data values from *<in_pipe_1>* to *<out_pipe>*, and so forth, in the manner of the `MERGE` command. The difference is that it takes data from each source in groups rather than one value at a time.

When the input and output pipes do not match for making a transfer, the data conversion rules of the `MERGE` command are applied. Conversions have a high computational cost, so for best efficiency avoid mixing data types.

`NMERGE` is useful for merging binary data from several pipes when the data rates in pipes are consistent but not identical, for transfer through a single communication pipe to the host computer. For instance, a single trigger time stamp value (`LONG` data type) can be merged with the values that were selected from a data stream by the `WAIT` command.

Example

```
NMERGE (1, P1, 1000, P2, $BinOut)
```

Transfer one data value from pipe `P1` followed by 1000 data values from pipe `P2` to the host computer through `$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 | LONG 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>*, discarding the first *<n-1>* triggers. *<n>* must be a positive, nonzero integer.

Example

```
NTH (T1, 100, T2)
```

Pass every hundredth trigger event from trigger `T1` to `T2`, discarding the rest.

See Also

`WAIT`

ODEFINE

Begin an output updating configuration.

ODEFINE *<name>*

ODEF *<name>*

Parameters

<name>

Assigned output configuration name.
Alphanumeric string limited to 23 characters.

Description

`ODEFINE` begins a group of commands that define an output updating configuration.

```
ODEFINE <name>
  [output configuration command] *
END
```

The complete list of output configuration commands is given in Chapter **5**. The command lines represented by the bracket notation 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.

Example

```
ODEFINE OUTPR
  CHANNELS  2
  SET OP0 A0
  SET OP1 A1
  TIME 1000
END
```

Begin the definition of an output configuration named `OUTPR` with 2 analog output channels in the output channel pipe.

See Also

`END`, `IDEFINE`, `PDEFINE`

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.
EVENTLOGSIZE	Set the number of available entries in the event log.
FLOATERROR	Set floating point error mode.
OVERFLOWQ	Hold overflow messages.
PROMPT	Control the system prompt character.
QUANTUM	Set global time slice quantum.
RESCUE	Activate a priority override “rescue” process
RESET	Reset options to power-up default.
RESTORE	Restore the options from the options stack.
RINTERVAL	Time interval separating rescue task quanta
RQUANTUM	Time interval in which rescue task can operate
SAVE	Push the current options onto a stack.
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. This option is advisory only– processing command implementations might not respond to it.

`DECIMAL = ON|OFF`

When `DECIMAL` is `OFF`, some system commands will display numbers using a hexadecimal format. When `DECIMAL` is `ON`, all number displays use a decimal number format. The default is `ON`. The command interpreter and the formatting of data for run-time input and output are not affected by this option.

`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 which 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.

`EVENTLOGSIZE = <n>`

The `EVENTLOGSIZE` option specifies the number of entries to reserve in the DAPL system event log for error and warning message. If the number of event messages exceeds this number, old values are removed to make room for new messages.

`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 **14** 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 **15** for more information.

RESCUE = ON|OFF

Turning on this option establishes a high-priority system task with no processing obligations of its own. This task allows ordinary processing tasks to make gradual progress, despite priority problems that would ordinarily prevent any activity. This slow progress is usually enough to identify the tasks that can run correctly and the tasks that are producing the deadlock problem. The costs of using the rescue task are so small that most configurations can leave the default ON without difficulty. Highly optimized configurations with rigid scheduling requirements can turn this option OFF.

RESET

Resets the options and options stack back to power-up defaults, as they would be immediately after a power-up start of the system. Specialized software applications can use this to good effect to establish a known state in one operation. Otherwise, use this with extreme caution, particularly in an interactive software environment such as DAPstudio, because it can produce major unexpected side effects such as silencing interactive command displays.

RESTORE

Restores the most recent options stored on the options stack. See the description of SAVE below. RESTORE does not take a *<value>*.

`RINTERVAL = <n>`
Specifies the time interval in microseconds between scheduled activations of rescue task processing.

`RQUANTUM = <n>`
Specifies the time interval duration in microseconds during which processing under control of the rescue task takes place.

`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>`*.

`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 the Interpreter display window of DAPstudio. 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 **14** 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	
Number Range	`ON`	`OFF`
-32768 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 sufficient data are present in output channel pipes.

OUTPUTWAIT *<preload>*

Parameters

<preload>

Number of samples that must be available before updating starts.

`WORD CONSTANT | LONG CONSTANT`

Description

The `OUTPUTWAIT` command causes an output-updating configuration to wait until *<preload>* samples have arrived from a data supplier before beginning to deliver output updates. The scheduling of tasks is unpredictable, and particularly so when a system configuration first starts to run. A delay in starting the output updates allows collecting enough data in memory to sustain operation despite temporary delays. Running out of data would cause the hardware processes to terminate prematurely with an *underflow condition.*

The value of *<preload>* must not be zero and must be a multiple of the number of output channel pipes. After this number of samples (or more) is received into the output channel pipe, the `OUTPUTWAIT` condition is satisfied, and the output updating task is scheduled for execution at high priority.

When the `HTRIGGER` command specifies an external hardware triggering mode other than `OFF`, the `OUTPUTWAIT` condition must be satisfied first before the output updating will respond to the external trigger signal level.

If a `CYCLE` command is present in the output-updating configuration, it overrides the `OUTPUTWAIT` specification. Satisfying the `CYCLE` command is equivalent to having an unlimited number of values available in memory.

For multiple-DAP systems generating output signals in a master-slave configuration, the `OUTPUTWAIT` value for each slave board must be satisfied before it can respond to the output clocking signals from the master board. Conflicts would occur at initial startup or at the beginning of output bursts. In some rare cases, you might need to increase the `OUTPUTWAIT` value on the master board, to give the slave boards sufficient time to generate enough data.

If you do not specify an `OUTPUTWAIT` value, the DAPL system computes a default number equivalent to 100 milliseconds (1/10 second) of output updates. Most applications should use the default `OUTPUTWAIT` value.

Warning: Setting *<preload>* less than its default value may result in output underflow and premature shutdown.

Example

```
OUTPUTWAIT 1000
```

Wait for 1000 values to arrive in the output channel pipe before starting to update the outputs.

See Also

`COUNT for output`, `CYCLE`, `UPDATE`, `HTRIGGER for output`

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 before accepting new commands.

PCASSERT

Define a task that asserts a trigger based on asynchronous input.

PCASSERT *(<signal>, <trigger> [,<ref_rate>])*

Parameters

<signal>

A source of data indicating a triggering event.
`WORD VARIABLE | LONG VARIABLE | WORD PIPE | LONG PIPE`

<trigger>

The trigger to assert when an event is requested.
`TRIGGER`

<ref_rate>

An optional data rate control.
`WORD CONSTANT |`
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`PCASSERT` asserts a trigger based on asynchronous input from *<signal>*. *<signal>* may be a fixed point variable or 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 *<signal>* is ordinarily used by the host computer to post its commands, but is not otherwise affected by 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. For example, 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.

By default, `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 T at the current input sample number whenever variable V is set to a nonzero value by the PC host or another task.

```
PCASSERT ($Cp2In, T)
```

Assert trigger T at the current input sample number whenever a value is received through communication pipe $Cp2In.

```
PCASSERT ($BinIn, T)
WAIT(IP(0..7), T, 4000, 4000, $BinOut)
```

Assert trigger T at the current input sample whenever a value is transferred from the PC host through the pre-defined communication pipe $BinIn. The event timestamps are relative to data in a single channel, and the WAIT command accepts any number of channels from an input channel pipe relative to a single channel, so no rate adjustment is needed. 4000 samples are taken before the event time, and 4000 samples after the event time, for a total of 8000 samples, 1000 from each channel. The results are transferred back to the PC host through the pre-defined communication pipe $BinOut.

```
PCASSERT (PREQUEST, T, 100)
AVERAGE (IP0,100,PAVG)
WAIT (PAVG, T, 0, 1, PCURRENT)
```

Assert a trigger event for a WAIT command that reads data reduced by an AVERAGE command.

```
BAVERAGE (IP(0..7), 8, 500, PBAVG)
PCASSERT ($BinIn, T)
TRIGSCALE (T, 0, 8, 500, TADJ)
WAIT (PBAVG, TADJ, 0, 8, $BinOut)
```

A BAVERAGE command working on 8 channels takes data at 8 times the rate for a single channel, but then reduces the data by a factor of 500. A request for data arriving through pre-defined communication pipe $BinIn produces an event in trigger T. A TRIGSCALE command adjusts the timestamp to account for the multiple channels and the averaging. The most recent set of 8 averaged results is returned to the PC host through the pre-defined $BinOut channel.

```
WAIT(IP(5),TFAULT,127,128,PFAULT)
PCASSERT ($BinIn, T, PFAULT)
WAIT(PFAULT,T,255,1,$BinOut)
```

Faults occur in the measured system at unpredictable times. At each of these fault events, a block of 256 samples is copied from sample channel IP(5) to the PFAULT pipe. For each request from the PC host through the pre-defined $BinIn communication pipe, an event location is asserted in trigger T relative to the data count in the PFAULT pipe. The second WAIT command then reports the most recent 256 captured values to the PC host through the pre-defined $BinOut communication pipe.

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`

PID

Define a task for PID feedback control.

PID *(<set_pipe>, <fb_pipe>, <gain_pipe>, <Tsamp>, <out_pipe>, [<low_clamp>, <high_clamp>])*

Parameters

<set_pipe>
Input data pipe for setpoint command levels or trajectory sequences.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<fb_pipe>
Input data pipe monitoring feedback from plant output.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<gain_pipe>
Pipe for adjusting control gain settings.
`FLOAT PIPE`

<Tsamp>
Sampling time interval for normalizing gains.
`FLOAT CONSTANT`

<out_pipe>
Control output signal to drive plant.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<low_clamp>
An optional parameter that limits negative control output swings.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<high_clamp>
An optional parameter that limits positive control output swings.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

Description

`PID` implements an “ISO standard form” PID controller, using control gains in natural units. The `PID` command reads feedback samples of the controlled system’s output from *<fb_pipe>,* and compares them to the desired system output level specified by *<set_pipe>*. The difference is the control error. The `PID` command applies the control gains from *<gain_pipe>* to compute control output values, which are sent to the *<out_pipe>.* The control output values drive the plant. The goal of the feedback control is to make the controlled system output match the specified setpoint level, reducing the control error to 0.

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, in an effort to damp out oscillations.

This basic strategy is more formally defined by the following equation:

```
output = K ( e  + (<Tsamp>/Ti) * intgrl(e) + (Td/<Tsamp>) * deriv(e) )
```

where

```
e  =  control error
intgrl(e) = estimated integral of e
deriv(e)  = estimated derivative of e
K, Ti, Td   are the control gains
```

Gains are specified as a set of four floating point parameters, placed into the *<gain_pipe>* in the following order.

K	feedback gain, dimensionless
Ti	integral time constant in seconds
Td	derivative time constant in seconds
Z	zero-shaping gain, dimensionless

Some notes regarding the gains:

1. The integral term and derivative term gains are scaled by *<Tsamp>*, accounting for natural time units. This makes it possible to adjust the sampling rate without retuning PID gains.
2. The zero shaping "Z gain" is an independent gain multiplier from 0.0 to 1.0 affecting the setpoint command signal path. There is no effect on the feedback or stability, so it is always safe. In some applications, a lower value can reduce undesirable transients in response to setpoint changes. A good tuning strategy is to set Z to 1.0, and tune the basic PID gains for best stabilization. Then, if accurate tracking of a changing setpoint signal is important, try reducing Z and watch for improved transient performance.
3. Derivative gain response increases with frequency, and this can lead to excessive amplification of frequencies close to the Nyquist limit. To avoid noise amplification or destabilization associated with this, the derivative estimator includes a built-in lowpass filter with rolloff at approximately 1/10 of the Nyquist limit.
4. For comparison, some other PID commands use the parallel PID form, obtained by combining multipliers so that each of the PID terms is scaled independently. That form is easier to compute internally, but harder to analyze and tune.

Place the four gains into the *<gain_pipe>* as a complete set. To adjust a single gain, specify all gains but repeat the previous values for the gains that are not changed. Initial gain settings can be loaded using the `FILL` command.

Setpoint commands can be specified as discrete levels, continuous trajectory streams, or a mix.

- If you provide single isolated setpoint values, the last value received will be treated as a constant level signal. That value will be used until a new setpoint level is provided.
- If you provide a continuous trajectory stream, the PID command will take one value for each update. It is important that trajectory samples are queued and ready. If the `PID` command temporarily runs out of

trajectory data and shifts to the isolated level strategy, this can induce a time skew between the desired trajectory stream and the PID action.

- If you want isolated setpoint values, but with specified transition paths to reach them, place bursts of data into the *<set_pipe>*, with the final value of the burst being the new settling level.

The `PID` command will operate upon any basic data type, but the data types of the command level, feedback, and output drive signals must match. Considerations when selecting a data type:

• `WORD`	natural fit for sampled data, efficient
• `LONG`	supports extended precision processing well
• `FLOAT`	a little quieter, scales well over small and large signals
• `DOUBLE`	provides the best precision

The *<low_clamp>* and *<high_clamp>* parameters limit the output range of the control drive signal. If you specify one limit, you must specify both. The limits can protect output devices or limit the drive power to large transient swings. When not specified, the implied limits for fixed point configurations are the range limits of the number representation; and the implied limits for floating point configurations are minus and plus 32767.

The *<low_clamp>* and *<high_clamp>* parameters have another effect. When there are large transients such as at initial start up, integral control action can have the negative side effect of building up a large accumulation, yielding an unnecessarily long settling time. The nonlinear *anti-windup strategy* of the PID command reduces the transient swings by reducing the integral accumulation rate when a *<low_clamp>* or *<high_clamp>* limit is reached.

Examples

```
PIPES PLEVEL WORD, PGAIN WORD, PFB WORD, PDRIVE WORD
FILL  PLEVEL  18000
FILL  PGAIN  8.2   0.08  0.125  1.00
…
PID (PLEVEL,PFB,PGAIN,0.001,PDRIVE)
DACOUT(PDRIVE,1)
```

Configure a PID control loop to operate upon `WORD` signals with classic PID action. The initial setpoint level placed into pipe `PLEVEL` is `18000`. The initial feedback gains are `K=8.2, Ti=0.08, Td=0.125, Z=1.00`. The updating time interval is 0.001 seconds. The output limits default to the balanced maximum range –32767 to +32767. Receive feedback information about system output level from pipe `PFB`. For each feedback value received, compute the PID control output and send it to pipe `PDRIVE`, to be delivered directly to output converter 1.

```
PIPES PLEVEL FLOAT, PGAIN FLOAT, PFB FLOAT, PDRIVE FLOAT
PID (PLEVEL,PFB,PGAIN,0.001,PDRIVE,0,32767)
```

Operate the same as in the previous example, but using `FLOAT` signals. Limit the PID output range so that the control level never swings negative.

See Also

`FILL`, `DACOUT`, `PIDLATCH`, `PIDRAMP`, `PIDSCRAM`

PIDLATCH

Define a task that switches smoothly between direct and feedback control.

PIDLATCH *(<set_pipe>, <fb_pipe>, <ctrl_pipe>, <drive_pipe>, <out_pipe>, <switch_pipe>)*

Parameters

<set_pipe>
The pipe driving the PID task setpoint level.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<fb_pipe>
The controlled feedback signal delivered to the PID controller.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<ctrl_pipe>
The drive signal calculated by the PID task..
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<drive_pipe>
The controlled drive signal delivered to the plant.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>
The feedback signal of the plant response.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<switch_pipe>
Operating mode, 0 or 1.
`WORD PIPE`

Description

The `PIDLATCH` command acts as an intermediary between a PID control loop and a controlled plant, so that the two can operate either in the usual coordinated fashion, or independently, with the plant driven by the setpoint command directly.

This command is intended for naturally stable systems that have special startup and shutdown requirements. For example, some systems must be “brought up to speed” in manual mode during a warm-up interval. After that, the equipment can be switched into active production with feedback control.

The `PIDLATCH` command monitors the level signal that goes to the PID control.

- the setpoint level <*set_pipe*>

It captures two signals that ordinarily transfer information between controller and plant.

- the output of the plant <*out_pipe*>
- the output of the controller<*ctrl_pipe*>

It delivers two signals to replace the signals that are captured.

- *<drive_pipe>* to drive the plant
- *<fb_pipe>* to drive the controller feedback

By default, the initial operating mode is mode 1. When the operating mode value from *<switch_pipe>* is 1, the plant and PID controller operate in ordinary closed loop fashion. The values from the *<out_pipe>* are relayed immediately to the *<fb_pipe>,* to drive PID task operation, and the values from the *<ctrl_pipe>* are relayed immediately to the *<drive_pipe>*, to drive the plant. When the operating mode value from *<switch_pipe>* equals 0, the PID controller and plant operate independently. The disconnected controller passively tracks the plant output level, while the plant is driven by a smoothed version of the setpoint level from *<set_pipe>*. The state of the PID controller remains close to the state of the plant, so there is minimal transient disturbance when the operating mode is restored to 1 to resume closed loop operation.

All basic data types are supported, but the data types of all signal pipes must match.

Example

```
PIPE  PSWITCH WORD
FILL  PSWITCH  0
…
PID(PSET,PIDFB,PGAINS,0.01,PIDOUT)
PIDLATCH (PSET,PIDFB,PIDOUT,PDRIVE,POUT,PSWITCH)
```

Configure a `PIDLATCH` task to monitor the setpoint `PSET` of a PID processing command. Capture its output signal `PIDOUT`. Substitute drive signal `PDRIVE` to drive the plant, while the plant output is captured by pipe `POUT`. The substitute feedback signal sent to the PID controller is `PIDFB`. The mode switch `PSWITCH` is pre-loaded with a value 0, which means that the controller and plant operate independently, with the plant following the `PSET` signal without PID control.

See Also

`PID`, `PIDRAMP`

PIDRAMP

Define a task that generates smooth transition ramps between control levels.

PIDRAMP *(<level_pipe>, <ramp_pipe>, <duration>, <shape>)*

Parameters

<level_pipe>
Pipe receiving isolated setpoint command levels.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<ramp_pipe>
Output pipe receiving data for smooth level-to-level transitions.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<duration>
Transition width.
`WORD CONSTANT | WORD VARIABLE | WORD PIPE`

<shape>
Identifier string for transition shape.
`STRING`

Description

The `PIDRAMP` command converts isolated control-level values into smooth transitions from level to level. `PIDRAMP` is used in conjunction with a `PID` command or other control loop command that accepts a steady level or a trajectory sequence. Smooth ramping helps to avoid oscillations in systems with poor stability. Complex ramping sequences can be pre-configured and run as a "batch" without further supervisory intervention.

Isolated input level values are received from *<level_pipe>*. As each is received, the `PIDRAMP` command generates the number of output values specified by the pipe or scalar *<duration>*, placing the data for the smoothed trajectory into *<ramp_pipe>.* At the final end of the smooth transition, the last value is the desired new level. The number of points in each transition cannot exceed 4096. The transition style is specified by parameter *<shape>*, which must be one of the following strings enclosed in double-quotes.

- "LINEAR"
  The best choice to limit the maximum rate of change during the transition, but there is a small discontinuity in the first derivative at each end.
- "EXPONENTIAL"
  The best choice to depart the old level quickly and devote most of the transition time to accurate final settling.
- "HYPERBOLIC"
  A symmetrical "S"-shaped transition that avoids discontinuity in values and derivatives at both ends, with fastest change in the middle.

The `PIDRAMP` command will generate any of the basic data types. The data types of the *<level_pipe>* and the *<ramp_pipe>* must match.

Examples

```
PIDRAMP (PLEVEL, PSMOOTH, 40, "EXPONENTIAL")
...
FILL  PLEVEL 10000
```

Receive a command from pipe PLEVEL to make a transition from the current operating level to a new operating setpoint 10000. The output command sequence is sent to a PID loop via output pipe PSMOOTH. The transition takes 40 samples and approaches the new target level exponentially.

```
PIPE  PLEVELS FLOAT
PIPE  PDURATIONS WORD
FILL  PLEVELS     0.00     20000.00  20000.00  0.00
FILL  PDURATION  200     500       10000        500
...
PIDRAMP (PLEVELS, PSMOOTH, PDURATIONS, "HYPERBOLIC")
```

Execute a 4-move sequence in which hyperbolic transition shapes are smooth at both ends of each transition. The sequence is preloaded into pipes PLEVELS and PDURATIONS. For the first move, use 200 samples to reach starting output level 0.00. For the second move, shift to level 20000.00 in 500 samples. For the third move, hold the 20000.00 level for 10000 samples. For the last move, take 500 samples to return to the 0.00 level. The complete set of 11200 trajectory points will be computed and delivered to the control loop command input via the PSMOOTH pipe.

See Also

FILL, DACOUT, PIDLATCH, PIDSCRAM

PIDSCRAM

Define a task that overrides a feedback loop for emergency shutdown.

```
PIDSCRAM (<drive_pipe>, <override_pipe>,
  <safelevel>, <duration>, <shape>, <scram_trig> )
```

Parameters

<drive_pipe>
Pipe providing the original control loop drive signal.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<override_pipe>
Pipe receiving pass-through drive signal or override shutdown sequence.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<safelevel>
Safe level for end of shutdown transition.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<duration>
Transition width.
`WORD CONSTANT`

<shape>
Identifier string for transition shape.
`STRING`

<scram_trig>
Software trigger reporting emergency shutdown event.
`TRIGGER`

Description

The `PIDSCRAM` command responds to emergency shutdown signals received by the *<scram_trig>* software trigger. When a shutdown event is detected, it replaces the normal control drive signal from *<drive_pipe>* with a controlled shutdown sequence in *<override_pipe>*. Under ordinary operation, the original drive signal in *<drive_pipe>* passes through directly to *<override_pipe>* with no change.

`PIDSCRAM` allows dedicated safety hardware or other supervisory fault detection systems to coordinate system shutdown sequencing when the control loop is part of a larger interconnected system. `PIDSCRAM` is appropriate when there is risk of system damage if the system fault condition is allowed to persist, but the supervisory system is not able to respond fast enough. The `PIDSCRAM` command ensures that the control loop shuts down autonomously, so that the only follow-up action required from supervisory control is the final power-down.

There is a cost in latency and response jitter by having the additional task receiving and retransmitting the control drive signal, so this disadvantage must be weighted against advantages of system safety.

The safe shutdown level is specified by the *<safelevel>* parameter. This value is the final value of the shutdown transition. The data type of the *<safelevel>* parameter must match the *<drive_pipe>* and *<override_pipe>*.

The number of samples spanned by the shutdown transition is specified by the *<duration>* parameter.

The transition style is specified by parameter *<shape>*, which must be one of the following strings enclosed in double-quotes.

- "LINEAR"
  The best choice to limit the maximum rate of change during the transition, but there is a small discontinuity in the first derivative at each end.
- "EXPONENTIAL"
  The best choice to depart the old level quickly and devote most of the transition time to accurate final settling.
- "HYPERBOLIC"
  A symmetrical "S"-shaped transition that avoids discontinuity in values and derivatives at both ends.

Example

```
PIDSCRAM (PCTRL, PSCRAM, 0, 60, "EXPONENTIAL", SCRAMTRIG)
```

Pass the control drive signal from pipe `PCTRL` to pipe `PSCRAM` without change, until trigger `SCRAMTRIG` detects the signal of an emergency shutdown event. When that occurs, make a smooth transition from the current output level to safe output level 0 in 60 steps, using an exponential transition shape.

See Also

`PID`, `PIDRAMP`

PIPES

Create new pipes for transferring data between tasks.

```
PIPES <pipe_def> [, <pipe_def>]*

PIPE <pipe_def> [, <pipe_def>]*

P <pipe_def> [, <pipe_def>]*

<pipe_def> = <pipe_name> [MAXSIZE=<max_size>] [STATIC]
   [BYTE | WORD | LONG | FLOAT | DOUBLE]
```

Parameters

<pipe_name>
  Assigned pipe name.

<max_size>
  The maximum number of values that can be held in the pipe.
  `WORD CONSTANT | LONG CONSTANT`

Description

The `PIPES` command creates pipes for streaming data between processing tasks. The data-type keyword `BYTE`, `WORD`, `LONG`, `FLOAT`, or `DOUBLE` specifies the type of the pipe contents. `BYTE` is not a computational type, but it indicates that the pipe contents are 8-bit text.

Pipes contain storage buffer areas, which in turn contain data. Pipes might have storage areas set up at the time they are defined, but the storage areas will initially contain no data.

<max_size> sets the maximum number of values that can be held in the pipe buffer storage. 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 DAPL system will assign to the pipe only as much memory as needed, up to the `MAXSIZE` limit. In most cases, it is best to omit the `MAXSIZE` parameter and let DAPL take care of providing the right amount of buffer memory for best performance. There are some situations, however, where it is useful to adjust the storage size. For pipes that tend to accumulate an excessive data backlog if allowed to do so, the backlog size can be bounded by limiting the pipe capacity. Also, some applications process huge data blocks as a single unit, and for these `MAXSIZE` can be made larger.

For some real-time applications, the small amounts of time to dynamically allocate and release blocks of buffer storage for pipes can mean unacceptable delays in delivering data. These applications can specify the `STATIC` option. This option tells the pipe to use only pre-allocated buffer memory from a storage area of `MAXSIZE` (typically, much smaller than the default value). This avoids the overhead of allocating and releasing storage from a free-memory pool. The `STATIC` option is not always a good idea, however, because it forces all data to be copied through the memory in the pipe's reserved buffers, disabling data management optimizations that might be available otherwise.

If the number of values held by the pipe reaches `MAXSIZE`, and more data arrive to go into that pipe, the pipe cannot accept the additional data. The task attempting to make this transfer is temporarily suspended until some data are removed from the pipe to make more capacity available. When a reader task has copied all available new data from a pipe, and the pipe has no new data to send when the reader task attempts to read more, the reader task is temporarily suspended until some other task provides the new data.

Examples

```
  PIPE P1
```

Define the pipe `P1`, with the default data type of `WORD` and the default maximum capacity of 32768 values.

```
  PIPES P2 MAXSIZE=1024, P3 LONG
```

Define a `WORD` pipe `P2` with maximum capacity 1024 values, and a `LONG` pipe `P3` with default maximum capacity.

```
  PIPE A256  MAXSIZE=256 STATIC
```

Define a pipe `A256` that uses pre-allocated buffer storage with a 256-item capacity limit.

See Also

`EMPTY`, `FILL`

POLAR

Define a task that converts complex data pairs from Cartesian to polar coordinates.

POLAR *(<p1>, <p2>, <p3>, <p4>)*

Parameters

<p1>
The pipe that provides the real parts of input complex numbers.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p2>
The pipe that provides the imaginary parts of input complex numbers.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p3>
The pipe that receives the amplitude of the complex result.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p4>
The pipe that receives the phase of the complex result.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

The `PIPES` command converts input data into polar coordinates. *<p1>* and *<p2>* are pipes that provide real and imaginary parts of complex number pairs. *<p3>* and *<p4>* are pipes that receive the amplitude and phase parts of the complex numbers in polar form. The data types for all input and output pipes must be the same.

Floating point phase results are in radians and require no conversions, but you can convert to degrees using a DAPL expression to multiply by 180/π. For fixed point data, the phase is expressed in a normalized form, where the largest positive value represents π, and the largest negative value represents –π.To convert `WORD` phase data to radians, multiply times π and divide by32768. To convert `LONG` phase data to radians, multiply times π and divide by 2147483648. For example, the following DAPL expression performs a conversion to degrees for `WORD` data type.

```
PDEG = PRAD * 180 / 32768
```

Quantization and noise can make phase angle results erratic when the values of the real and imaginary parts are both very small. This is an inherent hazard of the polar representation, and there is no consistent action that the `PIPES` command can take to correct for this.

If you need only the magnitude part or the angle part, consider using the `CMAG` or the `CANGLE` command for better efficiency.

Example

```
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`, `CMAG`, `CANGLE`

PRIORITY

Establish the execution priority of tasks defined in a processing procedure.

PRIORITY *<priority_number>*

Parameters

<priority_number>

Priority level in the range 1 to 4.
`WORD CONSTANT`

Description

The `PRIORITY` command assigns an execution priority to all of the tasks in a processing procedure. If specified, a `PRIORITY` command must appear after the `PRIORITY` command but before the processing tasks are defined. The priority ranges from 1 (lowest) to 4 (highest). The default priority level of 2 is assumed when the `PRIORITY` command is omitted. Most processing should run at the default priority level.

Within one priority level, every task gets an equal opportunity to execute. None of the tasks in a processing configuration with lower priority will be scheduled to execute until processing is suspended in all higher priority tasks. However, if tasks at lower priority constrain the execution of a higher priority task, the lower priority task may temporarily run in place of the constrained higher-level task until the conflict is resolved. For example, a lower-priority task might remove data from a pipe so that a higher-priority task can write new data there.

Real-time response latency is sometimes improved dramatically by shifting essential time-critical parts of the processing to separate tasks at a higher priority level, with routine data movement and bulk processing performed at a lower level.

Example

```
PDEFINE  HiPriority
  PRIORITY   3
  // task definitions here
END
```

Declare that all tasks in the processing procedure `HiPriority` will execute at higher than normal priority, for critical real-time response.

See Also

`PDEFINE`, and the chapter Prioritized Multitasking

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

Encode a signal level using pulse-width modulation.

PWM *(<in_pipe>, <minpulse>, <modbit>, <source>, <encoded>)*

Parameters

<in_pipe>
Input signal level data.
`WORD PIPE`

<minpulse>
Nominal lower bound on the width of a pulse.
`WORD CONSTANT`

<modbit>
Bit position, 0 to 15, for the bit to be modulated.
`WORD CONSTANT`

<source>
An initial digital stream prior to modulating.
`WORD PIPE`

<encoded>
Output data stream for digital port after modulation.
`WORD PIPE`

Description

`PWM` is an abbreviation for Pulse Width Modulation. This command allows encoding a continuous positive signal from *<in_pipe>* as digital pulses of varying width, where the average output level corresponds to the level of the input signal. The encoded output sequence replaces a selected bit position *<modbit>* from the original digital data stream *<source>*, producing a modified output stream *<encoded>*. If you do not have any prior content for the digital stream, a DAPL expression can initialize a stream for you, for example:

```
PSTREAM = 0
```

The level from *<in_pipe>* specifies a duty cycle between 0 and 1. If *<in_pipe>* has a `FLOAT` data type, the values are in the natural range from 0.0 to 1.0. If the *<in_pipe>* has a `WORD` data type, the full numeric range from 0 to 32767 represents duty cycles from 0.0 to 1.0 proportionally.

`PWM` toggles the value of the encoded bit so that the average of the pulsed signal levels equals the level of the input signal. `PWM` watches the running average of the output levels and switches when this value is too low or too high compared to the reference level from *<in_pipe>*. For an input modulation level of 0, the modulated signal bit will be continuously 0. For an input modulation level of 32767, the modulated signal bit will be continuously 1. For an input modulation level of 16384, the modulated signal bit will be 0 for 50% of the time and 1 for 50% of the time. Avoid noisy modulation level signals and extremes of range for best results.

To avoid chatter or overheating from too rapid switching, PWM enforces a minimum pulse width *<minpulse>*. After a switching transition, PWM holds the new 0 or 1 level for the number of samples indicated by *<minpulse>*. This can result in slightly larger excursions from a perfect balance over the short term, but over the long run the tracking of the average level remains accurate.

Example

```
PIPES PSTREAM WORD
…
PSTREAM = 0
PWM(PLEVEL, 25, 15, PSTREAM, OPIPE1)
```

Encode the high order bit 15 of an initially empty data stream PSTREAM with a pulse width modulated sequence. The modulation level is specified by values from the pipe PLEVEL. The minimum high or low output pulse duration is 25 samples. The modulated sequence is sent directly to an output channel pipe OPIPE1 for driving clocked output to a digital port.

See Also

ODEFINE, PID

QDCOUNT

Convert 16-bit Quadrature Decoder Board counts into a running sum.

QDCOUNT *(<inpipe>, [<initial>,] [<mode>,] <outpipe>)*

Parameters

<inpipe>

Input data pipe of 16-bit counts from a Quadrature Decoder Board.
`WORD PIPE`

<initial>

Initializer value for the running count accumulator.
`WORD CONSTANT | LONG CONSTANT`

<mode>

Keyword string specifying the operating mode.
`STRING`

<outpipe>

Output data pipe.
`WORD PIPE | LONG PIPE`

Description

The `QDCOUNT` command converts a sequence of readings of the 16-bit running counter on a Quadrature Decoder Board to produce a running, signed 32-bit count. Many applications accumulate counts that exceed the range representable by the 16-bit hardware counter on a Quadrature Decoder Board. The `QDCOUNT` command detects the overflow events and corrects for them, yielding an accurate 32-bit count. There is one important restriction, however. *The application sampling rate must be designed to assure that no more than 32767 counts up or down can accumulate between readings from the Quadrature Decoder Board*. Otherwise, it is not possible to determine whether a new reading from the 16-bit hardware counter indicates an increase or a decrease, thus resulting in an incorrect running sum.

The *<inpipe>* provides a sequence of sample values, obtained by reading the running 16-bit counter register on the Quadrature Decoder Board.

The optional *<initial>* parameter specifies an initial value for the 32-bit count accumulator. If this parameter is omitted, the starting value of the 32-bit count will be zero.

The optional *<mode>* parameter is a quoted string specifying an operating mode that affects the way processing is started. The `QDCOUNT` command has two operating modes. If the keyword "`ABSOLUTE`" or "`ABS`" is specified as the mode string, all counts present in the Quadrature Decoder Board register are considered meaningful. For this mode, the first output produced by the command will equal this first value plus the specified initial counter state. On the other hand, if the keyword "`RELATIVE`" or "`REL`" is specified as the mode string, `QDCOUNT` presumes that an indeterminate number of counts could exist in the Quadrature Decoder Board count register at the time when processing starts, so only changes subsequent to the initial count matter. For this mode, the first output produced by the command equals the *<initial>* parameter, and subsequent outputs reflect the cumulative change in counts relative to the first observed count. If the *<mode>* option is omitted, the default value is "`RELATIVE`".

The *<outpipe>* parameter specifies an output pipe. For each value received from *<inpipe>,* one value is written to *<outpipe>*. Each output value reports the current updated value of the 32-bit internal accumulator. If *<outpipe>* is a WORD valued pipe, the higher 16 bits are truncated, which could result in an overflow condition if the 32-bit accumulated count is too large. There is no protection from overflow conditions for either a WORD or LONG valued output pipe, but overflow is more likely to be a problem for the WORD case because of the relatively limited count range.

The QDCOUNT command responds to the value of OPTIONS BUFFERING at the time that the QDCOUNT starts. If used in control applications requiring minimal response latency (at the expenses of more processing overhead), specify the BUFFERING=OFF option.

The initialization sequence in the QDCOUNT command is different than it was in earlier 16-bit versions of this command. This can affect the number and value of samples at the beginning of a measurement sequence. Be sure to retest when substituting the newer DAPL system version into an existing application that previously used the 16-bit custom command version. If an application downloads the 16-bit QDCOUNT custom command version, this substitutes it for the 32-bit DAPL version until the next time the DAPL system is loaded.

Examples

```
QDCOUNT(PQD, PRUN32)
```

Accumulate a running 32-bit count, receiving 16-bit Quadrature Decoder Board readings from the WORD pipe PQD, and placing the updated values of the 32-bit running sum in the 32-bit pipe PRUN32. The default initial count of 0 and the default "RELATIVE" processing mode are used.

```
QDCOUNT(PQD2, 1024, "ABSOLUTE", PRUN16)
```

Accumulate a running 32-bit count, but report only the low order 16 bits, receiving 16-bit Quadrature Decoder Board readings from the WORD pipe PQD2, and placing the low 16 bits of the 32-bit running sum into 16-bit pipe PRUN16. An initial count offset of 1024 is applied to the running sum, and the "ABSOLUTE" processing mode is used.

See Also

CTCOUNT, CTRATE

QDECODE

Define a task that counts up-down pulses from a quadrature encoder.

QDECODE *(<pport>, <bits>, <out> [, <decim>])*

Parameters

<pport>
Data sequence obtained by sampling a digital port.
`WORD PIPE`

<bits>
Address of encoded signal pair in the digital port.
`WORD CONSTANT`

<out>
Destination of decoded output count.
`LONG PIPE | LONG VARIABLE`

<decim>
Optional decimation factor to use with output pipe option.
`WORD CONSTANT`

Description

The `QDECODE` command monitors the angular position of a rotating mechanism having an attached quadrature encoder. Samples are captured from the digital port and read from pipe *<pport>*. The bits to analyze are indicated by the *<bits>* parameter. Based on an analysis of the sequence of digital transitions, positive and negative counts of rotation are accumulated. The current sum value is posted to the output *<out>*.

One digital signal alone would be enough to indicate that the device is rotating, but not the direction. Using the two signals in combination, it is possible to identify both the movement and its direction. The `QDECODE` command reviews all of the input data and maintains the running sum of the position steps, up or down, and the result corresponds to the current angular position. The first sample determines the starting state and initializes the counter to zero. Updating of the internal accumulator starts with the next sample.

`QDECODE` is intended for applications in which rotating speeds and ranges of motion are relatively small. An example would be monitoring the position of a manually-operated control wheel. Encoder applications with high-speed rotating equipment or multiple channels should use a hardware-based implementation such as the Microstar Laboratories quadrature decoder board, MSXB 050.

The output behavior of `QDECODE` is a little different depending on whether the results go to a pipe or a variable. The *<out>* parameter can be a shared variable to allow other tasks to read the data at whatever times they wish. For this case, the `QDECODE` task wastes no time bringing *<out>* up to the most current value, grabbing a block with as much new data as possible and updating the value of *<out>* after the block is analyzed. The *<out>* parameter can be a pipe to observe the count at regular intervals for logging or feedback control. If there is no optional *<decim>* specification, the *<out>* pipe receives an update after processing each input value. Ordinarily, the encoder is sampled at a high rate, faster than other data streams. To reduce the data rate so that the `QDECODE` output rate matches the other data streams, let *<decim>* specify a positive integer value. Then, the *<out>* pipe receives one value for each *<decim>* values processed from the input stream.

The signals from the quadrature encoder are a pair of TTL digital signals connected to adjacent pins on the Data Acquisition Processor board's digital port. At a constant rotation speed, the encoder device produces square wave outputs on the two digital lines, with the transitions of one signal in the center of the squared pulses from the other signal. Or, in other words, the two signals are "in quadrature" at 90 degrees phase shift. Transitions never occur simultaneously on both lines. The two digital signals must be adjacent bits on the digital port, starting with an even-numbered bit: bit positions 0-1, or bit positions 2-3, etc. Specify the location of the bit pair using the *<bits>* parameter.

To detect every pulse accurately, the sampling rate for the digital port must be at least double the rate at which the encoder produces transitions at maximum rotating speed. The following example configuration samples the quadrature decoder signals at four times the rate of data capture in two analog channels.

```
Idefine  An2QD
  channels 8
  set IP0  b0      // digital encoder bits
  set IP1  g
  set IP2  b0      // digital encoder bits
  set IP3  d0             // ANALOG 0
  set IP4  b0      // digital encoder bits
  set IP5  g
  set IP6  b0      // digital encoder bits
  set IP7  d1             // ANALOG 1
  time  2.4
end
```

Examples

```
QDECODE( IP(0,2,4,6), 2, PACCUM, 4 )
```

Using the input sampling configuration shown in the example above, obtain an input stream of samples from the digital port, with the data arriving at a high rate. Observe the bit transitions at digital port positions 2 and 3. Depending on the observed changes in the bits, appropriately update the accumulated count. Every four input samples, report one current count value to pipe `PACCUM`, producing an update rate that matches the sample rate for analog samples in `IP3` and `IP7`.

```
QDECODE( PDIG, 10, VQ )
```

Obtain digital port samples through pipe `PDIG`. Observe the bit transitions at digital port positions 10 and 11 and from these determine the current angular position count. Place this into shared variable `VQ`.

See Also

`QDCOUNT`

RANDOM

Define a task that generates positive pseudorandom numbers.

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 | FLOAT PIPE | DOUBLE PIPE`

Description

`RANDOM` generates pseudorandom numbers appropriate for the specified *<out_pipe>*. Random numbers can be sent to a digital-to-analog converter to generate white noise.

<type> is reserved for a distribution selector code and must be 0. Type 0 uses a composite 32-bit generator to produce uniformly distributed 16 or 32 bit values. The `FLOAT` and `DOUBLE` output values are the same results as the fixed point numbers but scaled to range 0.0 to 1.0, so the full precision of `DOUBLE` data type is not used. While statistical properties are quite good, this is not a cryptographically strong generator, and in particular, the behaviors of individual bits must not be presumed random.

<seed> determines the sequence of random numbers. Any nonzero value generates a predetermined, reproducible sequence of numbers. A value of zero creates an unpredictable seed derived from the Data Acquisition Processor real-time clock.

Examples

```
PIPE P1
RANDOM (0, 0, P1)
```

Send pseudorandom numbers between 0 and 32767 to `WORD` pipe `P1`.

```
PIPE P1L LONG, P2W
…
RANDOM (0, 1774985, P1L)
P2W = P1L/32768 - 32768
```

Send predetermined pseudorandom numbers uniformly distributed between –32767 and 32767 to `WORD` pipe `P2W`.

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 | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<region>
A region selecting the data values to transfer.
`REGION`

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

Description

`RANGE` transfers data values that satisfy the condition specified by *<region>*, copying them from *<in_pipe>* to *<out_pipe>*. Values that do not satisfy *<region>* are ignored. The data types of the input and output pipes must match.

Use caution when using `RANGE` in combination with other processing. The amount of data written to *<out_pipe>* varies with the content of the data stream, making data transfer rates unpredictable.

Example

```
RANGE (P1, INSIDE,-1000.0,1000.0, P2)
```

Read data from pipe `P1`. For each value received, if the value is greater than or equal to -1000.0, and less than or equal to 1000.0, copy the value to pipe `P2`. Otherwise, discard the value and do not place it into pipe `P2`.

See Also

`BOUND`, `HIGH`, `LIMIT`, `LOW`

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 | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<count>
: A positive integer that specifies the number of terms averaged.
`WORD CONSTANT | LONG CONSTANT`

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

<phase_correction>
: An optional parameter requesting 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. The data types of the input and output pipes must match.

`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 processed data contains a different sample count than the sample count of a trigger assertion from unprocessed data. If the optional parameter *<phase_correction>* is present, some artificial initial output values are generated so that the original input samples and the samples produced by `RAVERAGE` correspond one-for-one. The value of this parameter is ignored, but using –1 is suggested to make operation similar to the `FIRFILTER` command.

Example

```
RAVERAGE (P1, 10, P2, -1)
```

Reads data from pipe `P1`, compute the averages of moving windows of 10 values, and put the average values in pipe `P2`. Use the phase correction option, so that the input and output sequences will contain the same number of samples and align when displayed on the same plot.

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 | FLOAT PIPE | DOUBLE PIPE`

<cnt>

A value that specifies the number of copies of each data value to transfer.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>

Output data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`REPLICATE` reads data values from *<in_pipe>* and places *<cnt>* copies of each data value into *<out_pipe>*. The data types of the input and output pipes must match.

Example

```
REPLICATE (P1, 3, P2)
```

Transfer three copies of each data value from `P1` into pipe `P2`.

See Also

`COPY`, `MERGE`

RESET

RESET *[<option>]*

Clear configurations, logs and system options.

Parameters

<option>
Option name.

Description

RESET clears user-defined configurations and stored results, as specified by the *<option>* keyword. The supported options are:

- no option-	Stop any running application configurations, and clear all application configurations from memory.
SYSLOG	Clear the contents of the system error message and event log.
ALL	Stop any running application configurations, and clear them all from configuration memory. Clear the contents of the system error message and event log. Reset system configuration options.

The main purpose of the RESET command is to clear user applications. This stops any running configurations, clears all user-defined configurations, clears any unused data, releases allocated memory, and erase all user-defined elements except communication pipes and downloaded command modules. Placing a RESET command at the beginning of each configuration ensures that no stray elements from a previous run interfere with the current configuration.

To clear all previous message from the system log, use the RESET command with the SYSLOG keyword. A RESET command without an *<option>* keyword does not affect the error message history stored in the system log.

To completely clear the system state back to a power-up initial state, use the RESET command with the ALL keyword. In addition to stopping and clearing all user configurations, and clearing the system log history, this resets all configurable system options to defaults in the manner of the OPTIONS RESET command. Use this with extreme caution, particularly in an interactive software environment such as DAPstudio, because it can produce major unexpected side effects such as silencing interactive command displays.

Example

```
RESET
```

Before stopping a measurement application, stop all user-configured tasks on the Data Acquisition Processor, remove all buffered data from memory, and clear all configuration definitions.

See Also

`EMPTY`, `OPTIONS`

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 | LONG PIPE | FLOAT PIPE | DOUBLE PIPE

<n>
The number of terms in each data block.
WORD CONSTANT | LONG CONSTANT

<out_pipe>
Output data pipe for the RMS values.
WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE 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, STDDEV

RSUM

Define a task that computes the running sum (integral) of a data stream.

```
RSUM (<in_pipe>, <count>, <divisor>, [<window>, ] <out_pipe> [, <reset>] > )
```

Parameters

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

<count>
Decimation interval, the number of input terms for each output.
`WORD CONSTANT | LONG CONSTANT`

<divisor>
Scaling divisor applied to each output.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`
`WORD VARIABLE | LONG VARIABLE | FLOAT VARIABLE | DOUBLE VARIABLE`

<window>
Optional window interval length.
`WORD CONSTANT | LONG CONSTANT`

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

<reset>
Optional restart interval length.
`WORD CONSTANT | LONG CONSTANT`

Description

The `RSUM` command is a general-purpose accumulator command for integration and running sums. It can operate value-by-value in the manner of `RAVERAGE`, or block-by-block in the manner of `AVERAGE.` It can sum continuously, or it can restart automatically after a block is completed. It can scale the results by an arbitrary scaling factor you specify.

Values are received from *<in_pipe>*. The values are summed into an accumulator. There are three ways to configure the duration of this summation processing.

1. If both the *<window>* parameter and the *<reset>* parameter are omitted, the summation continues indefinitely. Though the range of the accumulator is large, it is not infinite, and you will need to avoid an unbounded summation.
2. If the *<window>* length parameter is specified, input values eventually go out-of-scope and are dropped from the sum automatically in the manner of the `RAVERAGE` command. If the number of terms is large, this can use a lot of buffer storage.

3. If the *<reset>* parameter is non-zero, the accumulator value will be reset to zero each time this number of terms is processed, in the manner of `AVERAGE`. The *<reset>* parameter cannot be used when a *<window>* interval is specified.

Output values are the values of the running accumulator, divided by the current value of the *<divisor>*. The divisor value is arbitrary, but two common choices are:

1. Setting the *<divisor>* value to 1 leaves the reported values unscaled.
2. Setting the *<divisor>* value equal to the *<reset>* parameter value gives "divide by number of samples" in the manner of the `AVERAGE` command.

You must specify an output interval in the *<count>* parameter. The choices are:

1. Equal to 1: report the accumulation after every update.
2. Equal to *<window>*: report the accumulation once for the span of the summation window with no overlap.
3. Equal to *<reset>*: report the accumulation once for each input block.
4. Other value N: apply decimation, skipping N-1 values for each value reported.

The result of applying `RSUM` to a sample sequence without scaling is equivalent to applying a multiple-step rectangular rule integral approximation with time step normalized to one unit. Higher order integral approximations differ slightly at the beginning and end of an integration interval, but otherwise, the accumulator updates are exactly the same as the rectangular rule. Over a large number of terms, usually the difference does not matter.

Examples

```
RSUM (P1, 1, 100.0, P2, 1000)
```

Read data from pipe `P1`, compute the sum over 1000 points, reporting the cumulative value divided by 100.0 after processing each input value. Put the average values into pipe `P2`.

```
RSUM (P1, 10, 1.0, 101, P2)
```

Read data from pipe `P1`, compute the sum over a 101 term sliding window. There is no scaling of the result. The output stream is decimated by a factor of 10, yielding one output value for each 10 input values processed.

```
RSUM (P1, 100, 1.0,  P2)
```

Compute a continuous running accumulation of data from pipe `P1`, without scaling. Do not reset the accumulator, or cancel old data using a sliding window. Place results in pipe `P2`, reporting one value for every 100 processed. Make sure that the values from pipe `P1` do not have a constant bias or the accumulation will diverge toward infinity.

See Also

`AVERAGE`, `RAVERAGE`, `FIRFILTER`

SAMPLE

Specify the operating mode for an input sampling configuration.

SAMPLE *<option>*

Parameters

<option>

A keyword.

BURST | CONTINUOUS

Description

The SAMPLE command selects the operating mode for an input sampling configuration. The modes differ primarily in how sampling activity is restarted after it has been stopped. The available operating modes are:

- CONTINUOUS mode. This is the default mode. When used in combination with a hardware trigger, sampling starts under control of the trigger; otherwise, sampling starts immediately. Once started, sampling activity continues indefinitely until the COUNT command is satisfied, all available buffer memory is full, or the sampling configuration is stopped. After stopping, sampling remains stopped regardless of available memory or hardware signals.
- BURST mode. The BURST mode is used in combination with the COUNT command and an external hardware trigger. Sampling activity starts and stops similarly to the CONTINUOUS mode, but after stopping, it resets. When the triggering occurs again, the sampling activity can restart.

Both sampling modes can work with hardware triggering, as configured by the HTRIGGER command. When used, the hardware trigger starts the sampling the first time, but only the BURST mode responds to the trigger more than once. The GATED triggering mode is useful with the CONTINUOUS sampling, allowing sampling activity to be suspended and resumed without coordinating this with triggered data bursts.

In both sampling modes, the COUNT command can request sampling to stop after collecting the specified number of samples. Using COUNT is a requirement for the BURST mode, but an option for the CONTINUOUS mode. To get a burst that is *as long as possible* before memory capacity is exhausted, allowing the burst to be terminated by a memory overflow condition, specify an extremely large number on the COUNT command.

Do not configure a SLAVE board for BURST mode input sampling. The BURST mode expects the DAP hardware to provide a start signal, but that hardware signal is disabled when a master DAP provides the timing. A DAP configured as MASTER can operate in BURST mode, and the clocking signals it generates will work with the slave boards configured for CONTINUOUS mode operation. This has the same effect as if all boards were operating in a common BURST mode.

In previous versions of the DAPL system, this command was called UPDATE, because of similarities to the UPDATE configuration command for output updating. That name can be used as an alias. The keyword options on the two commands are similar, except that BURSTCYCLE mode for UPDATE does not have any meaning within sampling configurations.

Examples

```
SAMPLE   CONTINUOUS
```

Specify that input sampling operates in the continuous mode. Once stopped, the sampling will not respond again, even if data storage becomes available or the hardware triggering signal is active.

```
COUNT   25000
HTRIGGER   ONESHOT
SAMPLE   BURST
```

Specify that input sampling operates in the burst mode. The triggering goes active briefly until sampling starts. Sampling continues until 25000 samples are collected and then stops. This sequence is repeated the next time the hardware-triggering signal goes active.

See Also

`COUNT for input`, `HTRIGGER for input`, `UPDATE`, `SLAVE`

SAMPLEHOLD

Pause DAPL command processing until all sampled data are processed.

SAMPLEHOLD

Description

SAMPLEHOLD pauses DAPL until the currently active input configuration finishes sampling and all input channel pipes are empty. A sequence of commands

```
START <input_configuration>
SAMPLEHOLD
PAUSE 500
STOP <input_configuration>
```

is useful for extracting blocks of data relative to an external hardware trigger. 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.

See Also

COUNT, START, STOP, UPDATE

SAWTOOTH

Define a task that generates sawtooth wave data.

SAWTOOTH *(<amplitude>, <period>, <out_pipe>*
  [, <mod_type>, <mod_pipe] [, <mod_type>, <mod_pipe])

Parameters

<amplitude>
  The absolute magnitude of the waveform peak.
  WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT

<period>
  The number of sample values in each wave cycle.
  WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT

<out_pipe>
  Output pipe for waveform data.
  WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE

<mod_type>
  An optional modulation selector keyword.
  STRING

<mod_pipe>
  Pipe for a modulation signal, when a selector keyword is specified.
  WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE

Description

SAWTOOTH generates sawtooth wave data and places the data in *<out_pipe>*. The *<period>* is the number of sample values in each wave cycle. The *<amplitude>* is the absolute magnitude (positive one half the peak to peak range) of the output wave.

The *<amplitude>* does not have to match the output data type exactly, but it is restricted to representable values of the output data type. For example, the integer 100000 would be valid as the amplitude for a float output signal, but 61472133 would be invalid because it is not representable without rounding.

The cycle length specified by *<period>* is the number of samples for one complete waveform cycle. For fixed point output data types, the cycle length must be a fixed point number. For floating point data types, the cycle length does not need to be an exact integer, which means that it can produce frequencies that are not harmonically related to the sampling frequency. Aribtrarily long values of *<period>* are allowed. The most efficient operation is achieved using fixed point data and a fixed waveform length of 2048 terms or less.

There are two optional modulations. No modulation, one modulation option, or both modulation options can be specified. Each modulation specification consists of a selector string, followed by a pipe name identifying the modulation stream. The two modulation options are:

1. *Amplitude modulation. Modulation type "AMPLITUDE".*
   Values of the modulation signal multiply the corresponding values of the original waveform. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower

values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary.

2. *Frequency modulation. Modulation type "`FREQUENCY`".*
Values of the modulation signal act as a multiplier on the waveform frequency. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary but typically will be in the range 0.0 to 1.0. For example, if the *`<period>`* is 500 and the values from the frequency modulation pipe are all 0.5, the frequency is cut in half and the effect is the same as setting *`<period>`* equal to 1000.

When applying modulation, one modulation value is used for each waveform value generated.

Note: It is not possible to generate both a high-valued and a low-valued sample at the waveform high-to-low transitions; consequently, sawtooth waveform samples do not sum exactly to zero. If this small bias matters, you can use a DAPL expression to correct it.

Examples

```
SAWTOOTH (1000, 100, P2)
```

Generate a sawtooth wave with values ranging from -1000 to 1000, with a period of 100 samples. Place the waveform data into pipe P2.

```
SAWTOOTH (32767, 400, PMOD, "AMPLITUDE", PAMPL)
```

Generate a sawtooth wave with output values ranging from –32767 to 32767, and with a cycle length 400. Apply amplitude modulation, taking the multipliers from pipe `PAMPL`, and placing modulated waveform data into pipe `PMOD`.

See Also

`COSINEWAVE`, `SINEWAVE`, `TRIANGLE`, `SQUAREWAVE`

SDISPLAY

Print information about the specified configuration element. Symbols are the names of elements defined in a DAPL configuration.

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>, ... , <out_pipe_n-1>

Sequence of output data pipes for separated data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`SEPARATE` reads data from *<in_pipe>* and places the data consecutively into one or more output pipes. The first data value is sent to *<out_pipe_0>*, the second data value is sent to *<out_pipe_1>*, etc. . `SEPARATE` is the inverse of `MERGE`.

Unlike other DAPL commands that require input and output data types to match, the `SEPARATE` can distribute the output values to data pipes of different types, but the data types must match the types of the pipes that were merged originally to produce the multiplexed stream. Any conversion operations that were applied by the `MERGE` operation are reversed by the `SEPARATE` command to regenerate the original data streams. For most efficient processing, use data streams of matching type to avoid the conversions.

`SEPARATE` can be used for reading binary data from a host computer and splitting the binary data stream into several pipes for processing.

Examples

```
SEPARATE (P1, P2OUT, P3OUT, P4OUT)
```

Read data from pipe `P1` and place data consecutively into pipes `P2OUT`, `P3OUT`, and `P4OUT`.

```
SEPARATE ($BinIn, P5WORD, P6FLOAT)
```

Transfer and demultipliex the data from the binary input com pipe `$BinIn` to pipes `P5WORD and P6FLOAT`.

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 | FLOAT PIPE | DOUBLE PIPE`

<out_pipe_0>, ... , <out_pipe_n-1>

Sequence of output data pipes for separated data.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE 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>* according to the index specified by the flag. Each value transferred is preceeded by a flag from 0 to n-1, in the natural data type of the input pipe, that identifies the destination pipe. SEPARATEF is the inverse of MERGEF. Invalid flags cause the SEPARATEF task to terminate with an error.

Unlike most other DAPL processing commands, the input and output pipe data types do not have to match. When the data type for the input pipe and an output pipe do not match, a conversion operation is applied in a manner that reverses the conversion applied when the data stream is encoded by a MERGEF task. For most efficient operation, use input and output data streams of the same type to avoid the conversions.

SEPARATEF can be used for reading binary data that are sent at irregular intervals from a host computer, splitting the binary data stream into several pipes for processing.

Examples

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

Read flagged data from pipe P1 and place data 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 depend on the data acquisition processor model — see the hardware comparison listing or your Data Acquisition Processor hardware manual for information about allowable gains and sampling rate limitations at each gain. The following are examples of analog channel specifiers and gains:

```
S2 40
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.

```
CTL0
CT0
CT1    ICLOCK
```

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`'. If a `CTy` notation has the additional keyword `ICLOCK`, this indicates that the source of the counted events should be the timer board oscillator rather than an external signal line. 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 40
```

Input channel pipe 2 contains samples of a ground reference measured with gain of 40.

```
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 physical pins covered by each channel group notation depend on the group size as specified by the GROUPSIZE command, the Data Acquisition Processor model, and attached expansion cards. For details about connections to signal pins on the DAP analog signal connector and on expansion boards, check the hardware manual for your Data Acquisition Processor model, or check the “Hardware Differences” summary page that is included with each DAPL system release.

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))
```

Example

```
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>, <mod_pipe] [, <mod_type>, <mod_pipe])

Parameters

<amplitude>
The absolute magnitude of the waveform peak.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

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

<mod_type>
An optional modulation selector keyword.
`STRING`

<mod_pipe>
Pipe for a modulation signal, when a selector keyword is specified.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`SINEWAVE` generates sine wave data and places the data in *<out_pipe>*. The *<period>* is the number of sample values in each wave cycle. The *<amplitude>* is the absolute magnitude (positive one half the peak to peak range) of the output wave.

The *<amplitude>* does not have to match the output data type exactly, but it is restricted to representable values of the output data type. For example, the integer 100000 would be valid as the amplitude for a `float` output signal, but 61472133 would be invalid because it is not representable without rounding.

The cycle length specified by *<period>* is the number of samples for one complete waveform cycle. For fixed point output data types, the cycle length must be a fixed point number. For floating point data types, the cycle length does not need to be an exact integer, which means that it can produce frequencies that are not harmonically related to the sampling frequency. Aribtrarily long values of *<period>* are allowed. The most efficient operation is achieved using fixed point data and a fixed waveform length of 2048 terms or less.

Note: The `SINEWAVE` is identical to `COSINEWAVE` except for the phase of the signal.

There are two optional modulations. No modulation, one modulation option, or both modulation options can be specified. Each modulation specification consists of a selector string, followed by a pipe name identifying the modulation stream. The two modulation options are:

1. *Amplitude modulation. Modulation type "`AMPLITUDE`".*
   Values of the modulation signal multiply the corresponding values of the original waveform. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary.
2. *Frequency modulation. Modulation type "`FREQUENCY`".*
   Values of the modulation signal act as a multiplier on the waveform frequency. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary but typically will be in the range 0.0 to 1.0. For example, if the *`<period>`* is 500 and the values from the frequency modulation pipe are all 0.5, the frequency is cut in half and the effect is the same as setting *`<period>`* equal to 1000.

When applying modulation, one modulation value is used for each waveform value generated.

Examples

```
SINEWAVE (1000, 100, P2)
```

Generate a sine wave with values ranging from -1000 to 1000, with a period of 100 samples. Place the waveform data into pipe P2.

```
SINEWAVE (32767, 400, PMOD, "AMPLITUDE", PAMPL)
```

Generate a sine wave with output values ranging from –32767 to 32767, and with a cycle length 400. Apply amplitude modulation, taking the multipliers from pipe `PAMPL`, and placing modulated waveform data into pipe `PMOD`.

See Also

`COSINEWAVE`, `SAWTOOTH`, `TRIANGLE`, `SQUAREWAVE`

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` moves selected data from *<in_pipe>* to *<out_pipe>* and provides flexible options for skipping blocks of data. After initially doing a one-time skip of *<initial_skip>* values, `SKIP` alternately moves *<take_cnt>* values to *<out_pipe>* then ignores *<skip_cnt>* values. The data types of the input and output pipes must match.

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 for input, CLOCK for output, HTRIGGER for input, HTRIGGER for output, 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 | FLOAT PIPE | DOUBLE PIPE`

<out_pipe>

Output pipe for square root data.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE 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. For fixed point data types, the returned value is the greatest integer lower bound on the square root value. The data types of the input and output pipes must match.

Example

```
SQRT (P1, P2)
```

Read data from pipe `P1`, compute square roots, and place the results in pipe `P2`.

See Also

`CMAG`, `RMS`

SQUAREWAVE

Define a task that generates square wave data.

SQUAREWAVE *(<amplitude>, <period>, <out_pipe>*
[, <mod_type>, <mod_pipe] [, <mod_type>, <mod_pipe])

Parameters

<amplitude>
The absolute magnitude of the waveform peak.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

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

<mod_type>
An optional modulation selector keyword.
`STRING`

<mod_pipe>
Pipe for a modulation signal, when a selector keyword is specified.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`SQUAREWAVE` generates square wave data and places the data in *<out_pipe>*. The *<period>* is the number of sample values in each wave cycle. The *<amplitude>* is the absolute magnitude (positive one half the peak to peak range) of the output wave.

The *<amplitude>* does not have to match the output data type exactly, but it is restricted to representable values of the output data type. For example, the integer 100000 would be valid as the amplitude for a `float` output signal, but 61472133 would be invalid because it is not representable without rounding.

The cycle length specified by *<period>* is the number of samples for one complete waveform cycle. For fixed point output data types, the cycle length must be a fixed point number. For floating point data types, the cycle length does not need to be an exact integer, which means that it can produce frequencies that are not harmonically related to the sampling frequency. Aribtrarily long values of *<period>* are allowed. The most efficient operation is achieved using fixed point data and a fixed waveform length of 2048 terms or less.

There are two optional modulations. No modulation, one modulation option, or both modulation options can be specified. Each modulation specification consists of a selector string, followed by a pipe name identifying the modulation stream. The two modulation options are:

1. *Amplitude modulation. Modulation type "AMPLITUDE".*
   Values of the modulation signal multiply the corresponding values of the original waveform. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower

values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary.

2. *Frequency modulation. Modulation type "`FREQUENCY`".*
Values of the modulation signal act as a multiplier on the waveform frequency. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary but typically will be in the range 0.0 to 1.0. For example, if the *`<period>`* is 500 and the values from the frequency modulation pipe are all 0.5, the frequency is cut in half and the effect is the same as setting *`<period>`* equal to 1000.

When applying modulation, one modulation value is used for each waveform value generated.

Note: If you need exact waveform symmetry, you must specify a cycle length that is an even number, so that the number of samples on the positive half cycle can equal the number of samples on the negative half-cycle.

Examples

```
SQUAREWAVE (1000, 100, P2)
```

Generate a square wave with values ranging from -1000 to 1000, with a period of 100 samples. Place the waveform data into pipe P2.

```
SQUAREWAVE (32767, 400, PMOD, "AMPLITUDE", PAMPL)
```

Generate a square wave with output values ranging from –32767 to 32767, and with a cycle length 400. Apply amplitude modulation, taking the multipliers from pipe `PAMPL`, and placing modulated waveform data into pipe `PMOD`.

See Also

`COSINEWAVE`, `SAWTOOTH`, `SINEWAVE`, `TRIANGLE`

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. The processes started by one START command are sometimes called a *start group*. Tasks within a start group can read copies of data produced by tasks within that group.

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 which 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] [TASKING] [PIPES] [DISPLAYALL] | OFF*

STAT *ON | [DISPLAY] [TASKING] [PIPES] [DISPLAYALL] | OFF*

Description

The STATISTICS command displays information about processor and capacity utilization of the system and processing tasks. The report displayed by STATISTICS might look something like the following:

```
    Task            Time Used
                     (in mS)
  ----------------------------
  RESCUER   ( 5)     10.57
  DAPL_TSK  ( 2)      0.36
  HOST_TSK  ( 2)      0.08
  INF_TSK   ( 2)      0.00
  CFG_TSK   ( 2)      0.00
  RANDOM    ( 2)   1002.24
  RANDOM    ( 2)    967.57
  $DAPLEXPR ( 2)   2285.14
  DACOUT    ( 2)    153.27
  IDL_TSK   (-1)      0.00
  ----------------------------
  total time elapsed:                    5000.72 mS
  system idle/overhead:         0.00/581.49 mS (0.00%/11.63%)
```

The report lists the tasks, shows their running priority, and the amount of time that the tasks consumed while running within the monitored time interval. Summary CPU loading statistics are also shown.

To initiate collection of the measurements, issue the STATISTICS command with the ON command line option. After STATISTICS collection begins, pause for a few seconds, then use a STATISTICS DISPLAY command, or simply STATISTICS 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 should no longer be collected, issue the STATISTICS command with the OFF command line option.

It is common to send the STATISTICS command to the DAPL system interactively. To save typing, the STATISTICS command can be abbreviated to STAT.

The STATISTICS command has some additional command line options that can sometimes be useful.

- `STAT CLEAR`
  Use this to reset the accumulated statistics to zero at any time.
- `STAT PIPES`
  Use this instead of DISPLAY to get a report with extra information about the maximum amount of

buffered data accumulated in pipes. This is particularly useful for studying real-time applications, because backlogs of data in pipes indicate delays in delivering real-time response.

- `STAT  TASKING`
  Use this at any time to add an extended listing of tasking information, showing information about delays between opportunities for tasks to execute. This can be important for estimating bounds on guaranteed real-time response to events.
- `STAT  DISPLAYALL    or    STAT  ALL`
  Adds both the `PIPES` and `TASKING` reports as described above.

Example

```
STATISTICS ON
STATISTICS CLEAR
PAUSE 10000
STATISTICS DISPLAY
STATISTICS OFF
```

Report how the processing tasks in the current configuration used CPU resources during a 10-second execution interval.

STATUS

Display information about the current status of the system.

```
STATUS
```

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.

STDDEV

Define a task that computes the sample standard deviation for a group of samples.

STDDEV *(<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 the processed block.
`WORD CONSTANT | LONG CONSTANT`

<out_pipe>
Output pipe for the standard deviation data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`STDDEV` computes one value of standard deviation for each block of *<count>* samples received from *<in_pipe>*. Results are sent to *<out_pipe>*. The data types of *<in_pipe>* and *<out_pipe>* must match.

The standard deviation of a sample block is computed as: sum the squared deviations from expected value, divide by *<count>*, and take the square root. If you are using `STDDEV` to estimate parameters of a normal distribution, the result will be slightly biased. You can correct the bias by pre-computing the correction factor `sqrt(`*<count>*`/`*<count>*`-1)` and applying this using a DAPL expression.

Note: `STDDEV` is equivalent to `VARIANCE` combined with a square root operation.

Examples

```
STDDEV (IPIPE0, 1000, PSTD)
```

Analyze groups of 1000 values from input channel pipe `IPIPE0` and send the results to pipe `PSTD`.

```
STDDEV (P2, 40, P3)
P3ADJUST = P3 * 1.0127
```

Analyze groups of 40 values from pipe `P2` to estimate the variance of the distribution from which the sample values were obtained. Apply a bias correction factor for a block size 40, adjusting the original results in pipe `P3` to obtain corrected parameter estimate results in pipe `P3ADJUST`.

See Also

`AVERAGE`, `RMS`, `VARIANCE`

STOP

Stop input sampling, processing, and output updating configurations.

STOP `[<name> [,<name>]*]`

STO `[<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>,*
<pmag>, <pang> [, <limit1>, <limit2>])

Parameters

<p1>

A pipe which contains the real part of the Fourier transform of the input to a system under test.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p2>

A pipe which contains the imaginary part of the Fourier transform of the input to a system under test.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p3>

A pipe which contains the real part of the Fourier transform of the output of the system under test.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p4>

A pipe which contains the imaginary part of the Fourier transform of the output of the system under test.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<pmag>

Output pipe for amplitude data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<pang>

Output pipe for phase data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<scale>

A scale factor for decibel output.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<limit1>

An optional word constant for suppressing computations with very small inputs.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<limit2>

An optional word constant for suppressing computations with very small outputs.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

Description

`TFUNCTION1` calculates frequency response 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 and is more efficient when data sets are clean and accurate. If signals are noisy, it is better to compute the crosspower and autopower spectra, which can be averaged to reduce noise, and then apply `TFUNCTION2` to calculate the transfer function.

<p1> and *<p2>* are pipes that contain the real and imaginary components of the Fourier transform of the input signal for 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. The amplitude is converted to decibels and written to pipe *<pmag>*. *<scale>* is a scale factor for the decibel output that can be useful for preserving precision when using fixed point data types. For more details about representing decibel values in the various data types, see the description of the `DECIBEL` command. The phase is represented in radians and written to pipe *<pang>*. For more details about representing phase angles in the various data types, see the description of the `POLAR` command. All pipes and constants must be of the same data type.

The transfer function output is meaningful only where the inputs and outputs are not too small; otherwise, the results are dominated by noise and very erratic. *<limit1>* and *<limit2>* are optional constraints to fix the transfer function result for frequencies with insufficient signal energy. If the amplitude of the input level is less than *<limit1>*, or the output level is less than *<limit2>*, *<pmag>*, and*<pang>* are 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>, <pmag>, <pang>
  [, <limit1>, <limit2>])
```

Parameters

<p1>
A pipe which contains the real parts of the autopower spectrum of the input to a system under test.
`LONG PIPE`

<p2>
A pipe which contains the real parts of the crosspower spectrum of the output of the system under test.
`LONG PIPE`

<p3>
A pipe which 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`

<pmag>
Output pipe for amplitude data.
`WORD PIPE`

<pang>
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 and is more efficient when data sets are clean and accurate. If signals are noisy, it is better to compute the crosspower and autopower spectra, which can be averaged to reduce noise, and then apply `TFUNCTION2` to calculate the transfer gains.

The `CROSSPOWER` command produces useful results in `LONG`, `FLOAT`, and `DOUBLE` data types, so the `WORD` data type is not supported. All pipes and constants must be of the same data type.

<p1> and *<p2>* are pipes that contain the real and imaginary components of the Fourier transform of the input signal for 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. The amplitude is converted to decibels and written to pipe *<pmag>*. *<scale>* is a scale factor for the decibel output that can be useful for preserving precision when using fixed point data types. For more details about representing decibel values in the various data types, see the description of the `DECIBEL` command. The phase is represented in radians and written to pipe *<pang>*. For more details about representing phase angles in the various data types, see the description of the `POLAR` command. All pipes and constants must be of the same data type.

The transfer function output is meaningful only where the inputs and outputs are not too small; otherwise, the results are dominated by noise and very erratic. *<limit1>* and *<limit2>* are optional constraints to fix the transfer function result for frequencies with insufficient signal energy. If the amplitude of the input level is less than *<limit1>*, or the output level is less than *<limit2>*, *<pmag>*, and*<pang>* are 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 in *<trigger>*, first skipping *<n>-1* counts, and then asserting an event at the next location. This cycle is repeated for each subsequent trigger assertion, so the count for each posted event increases by *<n>*. Suppose for example that *<n>* equals 1000. At initial start-up, the 999 sample counts numbered 0 through 998 will be skipped, and the first event will be posted at sample number 999. The next event will be posted at sample 1999, the next event at sample 2999, and so forth.

`TGEN` is useful during test and development stages of a project, when a system is not fully functional. The artificial events allow other parts of the processing to be tested, before a signal is available that can provide actual triggering events.

`TGEN` is unusual because it does not analyze a data stream to select triggering locations. This means that the events it generates could be independent of the data rate in any particular stream, but does not mean independence from processing rate deadlocks when used with multiple streams. For example, a task using the events to process data at a very slow rate can force the `TGEN` task to add new events slowly, so that it fails to keep pace with a task processing data at fast data rates. To avoid problems, apply the events to data streams operating at a consistent rate.

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 code or name string 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 which 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 respectively. Alternatively, you can specify the thermocouple type as a one-letter string enclosed in double-quotation marks. 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 terms should be adjusted according to the gain on the input signal.

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. Separate measurements can be used to establish the value of this variable. 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 “E”	-270 to 1000	0.4
ANSI type J (IEC 584)	1 “J”	-200 to 1200	0.3
ANSI type K (IEC 584)	2 “K”	-270 to 1370	0.4
ANSI type R (IEC 584)	3 “R”	-50 to 1760	0.4
ANSI type S (IEC 584)	4 “S”	-50 to 1750	0.3
ANSI type T (IEC 584)	5 “T”	-270 to 400	0.3
ANSI proposed type N	6 “N”	-260 to 1300	0.3
Type U (DIN 43710)	7 “U”	-200 to 600	0.2
Type L (DIN 43710)	8 “L”	-200 to 900	0.3
Type B (IEC 584)	9 “B”	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, “K”, 12500, 32767, P2)
```

Thermocouple potential is measured at a gain of 40 on a –5 to +5 volt range, so the maximum positive digitizer range 32767 represents +1/8 of a volt, which is 12500 steps in units of 10 microvolts. Read input data from pipe P1, multiply the data by 12500/32767, 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.

Several hardware-related restrictions limit the range of input and output rates the the `TIME` command can allow.

- analog conversion rate limits for converter devices
- digital propagation and settling time limits for digital devices
- timing resolution supportable by timing oscillators
- timing durations supportable by digital interval counters
- CPU processing capacity limits for managing hardware devices

These restrictions, dependent on the hardware features of your specific Data Acquisition Processor, will determine maximum sampling and updating rates that can be supported. For specific information about your particular Data Acquisition Processor model, go to the HWDIF.HTM file. From your software install CD, look for the "Documentation | PDF" link in the introductory screen, and select the "DAPL 3000 Hardware Dependencies" item to browse the file.

Even though the hardware dependencies document states that conversions can be performed at certain rates, full accuracy of the conversion is not always guaranteed at these maximum rates. Some of the conditions that affect the maximum sampling and updating rates:

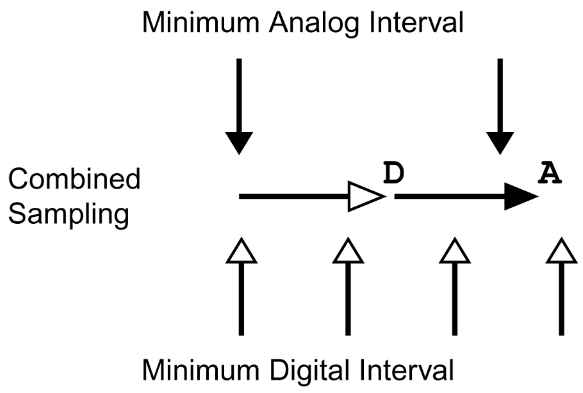
- *Quality of signal cables.* Cable type, length and termination are critical to the success of high speed sampling and updating. Cabling should be tested to verify proper operation at high speeds.
- *Slew rate limiting.* If an input or output amplifier switches between widely differing input or output 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.

If your configuration uses expansion boards, these boards can have their own isolation amplifiers, conversion clocking, latching logic, etc., with these devices having their own timing restrictions. The DAPL system might allow timing intervals for which the expansion boards cannot produce accurate results. Be careful to observe the timing restrictions described in the hardware manuals for each board.

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.

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.

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 which 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.

Examples

```
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 which 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 also can 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>, <mod_pipe] [, <mod_type>, <mod_pipe] )
```

Parameters

<amplitude>
The absolute magnitude of the waveform peak.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<period>
The number of sample values in each wave cycle.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

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

<mod_type>
An optional modulation selector keyword.
`STRING`

<mod_pipe>
Pipe for a modulation signal, when a selector keyword is specified.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`TRIANGLE` generates triangle wave data and places the data in *<out_pipe>*. The *<period>* is the number of sample values in each wave cycle. The *<amplitude>* is the absolute magnitude (positive one half the peak to peak range) of the output wave.

The *<amplitude>* does not have to match the output data type exactly, but it is restricted to representable values of the output data type. For example, the integer 100000 would be valid as the amplitude for a `float` output signal, but 61472133 would be invalid because it is not representable without rounding.

The cycle length specified by *<period>* is the number of samples for one complete waveform cycle. For fixed point output data types, the cycle length must be a fixed point number. For floating point data types, the cycle length does not need to be an exact integer, which means that it can produce frequencies that are not harmonically related to the sampling frequency. Aribtrarily long values of *<period>* are allowed. The most efficient operation is achieved using fixed point data and a fixed waveform length of 2048 terms or less.

There are two optional modulations. No modulation, one modulation option, or both modulation options can be specified. Each modulation specification consists of a selector string, followed by a pipe name identifying the modulation stream. The two modulation options are:

1. *Amplitude modulation. Modulation type "AMPLITUDE".*
   Values of the modulation signal multiply the corresponding values of the original waveform. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower

values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary.

2. *Frequency modulation. Modulation type "`FREQUENCY`".*
   Values of the modulation signal act as a multiplier on the waveform frequency. For fixed-point data types, the maximum representable value corresponds to a scaling factor of 1.0, and lower values correspond to proportional fractions. For floating point data types, the modulation values are arbitrary but typically will be in the range 0.0 to 1.0. For example, if the *`<period>`* is 500 and the values from the frequency modulation pipe are all 0.5, the frequency is cut in half and the effect is the same as setting *`<period>`* equal to 1000.

When applying modulation, one modulation value is used for each waveform value generated.

Note: If you need exact waveform symmetry, you must specify a cycle length that is an even number, so that the waveform peaks all occur at exact sample locations.

Example

```
TRIANGLE (1000, 100, P2)
```

Generate a triangle wave with values ranging from -1000 to 1000, with a period of 100 samples. Place the waveform data into pipe P2.

```
TRIANGLE (32767, 400, PMOD, "AMPLITUDE", PAMPL)
```

Generate a triangle wave with output values ranging from –32767 to 32767, and with a cycle length 400. Apply amplitude modulation, taking the multipliers from pipe `PAMPL`, and placing modulated waveform data into pipe `PMOD`.

See Also

`COSINEWAVE`, `SAWTOOTH`, `SINEWAVE`, `SQAREWAVE`

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_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:
`ARMED | 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. All other parameters are optional, and most applications can use the defaults. If none of the optional parameters are specified, the trigger will operate in the `NATIVE` mode, as defined below.

The trigger begins processing its incoming stream of event notifications when all reader and writer tasks for the trigger are started. Operating modes modify the way that incoming trigger events are processed, accepting some requests to assert the trigger and suppressing others.

- `NATIVE`. The `NATIVE` mode applies no filtering actions, and the trigger does not use any of the trigger properties. This deprecated notation means exactly the same thing as the `NORMAL` mode using default property values.

- `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. For data streams processed in blocks, this avoids redundant events applied to one data block.

- `DEFERRED`. This mode is the same as `NORMAL` mode, except that events occurring inside the `HOLDOFF` interval are delayed until after the `HOLDOFF` interval, guaranteeing that you always capture enough data following an event.

- `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 optionally can 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`. After an event is posted, the trigger will not accept any new assertion events until *`<hold>`* samples later, ignoring any events that occurs in the *`<hold>-1`* samples before that time. The default is no holdoff interval, so an event can be posted at the next sample.

- `STARTUP`. This property specifies an interval similar to `HOLDOFF`, except that an event cannot occur until *`<start>`* number of samples. For example, if the `STARTUP` interval is 100, the 99 samples numbered 0 through 98 are skipped, and the trigger can start responding to events at the 100th sample, number 99.

- `CYCLE`. This property is used by the `AUTO` mode to determine the number of samples between artificially-generated events. For example, if the `CYCLE` length is 100, an artificial event can occur once per 100 samples when there are no other events posted.

Note: A trigger uses 32-bit sample counts as its "timestamp" for identifying the location of assertions. If an application generates more than 2^32-1 data values (4294967296 samples), trigger assertion counts wrap around to zero.

Examples

```
TRIGGERS T2, T3
```

Define two triggers, `T2` and `T3`, using default mode and properties.

```
TRIGGERS TA MODE=AUTO CYCLE=5000 HOLDOFF=100
```

Trigger `TA` operates in the `AUTO` mode, and artificial events are inserted once per 2000 samples when there are no other events. There is a holdoff interval of 100 samples, so after any actual or artificial event, no new events are accepted until the 100th sample following.

```
  TRIGGERS TM MODE=MANUAL STARTUP=2000 GATE=ARMED
```

Trigger `TM` operates in the `MANUAL` mode. At startup time, no event can be recognized until the 2000th sample. At the end of the startup interval, the trigger can respond to the first event it receives. Upon receiving that event, it changes its `GATE` property from `ARMED` to `DISARMED`.

```
  TRIGGERS TN MODE=NORMAL STARTUP=5000 HOLDOFF=100 \
    GATE=DISARMED
```

Trigger `TN` starts `DISARMED`. Two things must occur before the trigger `TN` responds to incoming events: 1) the `GATE` property must be changed to `ARMED` and 2) the startup interval of 5000 samples must be completed. After both of these conditions are satisfied, the trigger continues operating in `NORMAL` mode.

See Also

`LIMIT`, `WAIT`, `TRIGARM`, `HTRIGGER for input`, `HTRIGGER for output`

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 **16** 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>, <div>, <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`

<div>
: 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 *<div>* 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 *<div>* 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 `<div>` parameter. The value of `<div>` 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 `<div>` 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.

Examples

```
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 **16** 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 observe the time stamps recorded by trigger events. TSTAMP waits for *<trigger>* assertions. Each time *<trigger>* is asserted, TSTAMP puts the sample number of the trigger event into *<out_pipe>*. The output value can then be treated as an ordinary fixed point number.

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 the operating mode for an output updating configuration.

UPDATE *<option>*

Parameters

<option>

A keyword, one of the following.

`BURST | BURSTCYCLE | CONTINUOUS`

Description

The `UPDATE` command selects the operating mode for an output updating configuration. The modes differ primarily in how they restart after updating has been temporarily stopped because of data underflow or satisfying the condition specified by a `COUNT` command. The available operating modes are:

- `CONTINUOUS` mode. This is the default mode. Once started, updating activity continues indefinitely until the supply of data is exhausted or the updating is stopped. Once stopped, updating remains stopped regardless of additional data or hardware signals.
- `BURST` mode. Updating activity starts and stops similarly to the `CONTINUOUS` mode, but after stopping, it resets. When new data are available and starting conditions are again satisfied, the updating activity can restart.
- `BURSTCYCLE` mode. This mode is similar to the `BURST` mode, but the data supplied for the first burst are retained in memory and used for all bursts. Only a hardware triggering signal is needed to resume updating for subsequent bursts.

There are two conditions necessary to begin updating in all modes.

- Sufficient data to cover the requirements of the `OUTPUTWAIT` command or `CYCLE` command must be available.
- If hardware triggering is configured, the hardware triggering level must then be active.

Updating activity stops under one of the following conditions.

- The number of updates specified by the `COUNT` command has been completed.
- No data were available when hardware updating needed them. This is an unintended *underflow condition.*
- A specific number of samples delivered to output processing have all been used. This causes a deliberate and controlled *underflow condition.*

In the `CONTINUOUS` updating mode, or in the `BURST` updating mode prior to satisfying the `COUNT` specification, an underflow is considered an error condition, and results in a warning message. In the `BURST` mode, an underflow that occurs at the time that a `COUNT` command specification is satisfied is not considered an error condition, and it is not diagnosed.

When using the CYCLE command in the BURST updating mode, the COUNT command must also be used, otherwise there is no way to stop the bursts. Typically, the CYCLE length covers one period of a repeating waveform, and the COUNT command determines how many copies are generated by a burst.

The BURST mode expects new data to reinitialize it after stopping. The BURSTCYCLE mode retains all of its previous data in memory and uses that data again the next time the updating is started. If you need to reconfigure the output signal between bursts, you will want to use the BURST mode. If the output waveforms are always the same, you will want to use the BURSTCYCLE mode.

Do not specify BURST mode or BURSTCYCLE mode output updating for a DAP configured as a SLAVE board. These modes expect the DAP hardware to provide a start signal, but that hardware signal is disabled when a master DAP provides the timing. A DAP configured as MASTER can operate in BURST mode, and the clocking signals it generates will work with the slave boards in CONTINUOUS mode operation. This has the same effect as if all boards were operating in a common BURST mode.

Example

```
OUTPUTWAIT  21
COUNT  21
UPDATE  BURST
```

Data are transferred to an external system in packets of 20 digital words, followed by a separator zero word. Specify that output updating operates in burst mode, in groups of 21 words. Updating automatically stops after each packet is delivered. The burst mode allows updating to resume as soon as the next packet of data arrives.

See Also

CYCLE, COUNT, HTRIGGER, OUTPUTWAIT, SAMPLE

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.

If the data type specification is omitted, the data type of the variable defaults to `WORD`. If there is an initializer expression in this case, the value must be compatible with `WORD` data type.

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 | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<count>

The number of data values per block.

`WORD CONSTANT`

<out_pipe>

Output pipe for variance data.

`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

`VARIANCE` computes the variance of blocks of *<count>* data values from *<in_pipe>*. Each variance result is sent to *<out_pipe>*.

In general, the input and output data types must match, but there is a special case for `WORD` input data because it is common for the results to be too large to represent in a `WORD` output data type. The `VARIANCE` command will allow a `LONG` output data type from processing `WORD` input data. `LONG` input data can produce the same sort of range problems, but there is no higher precision fixed point type for safely returning its results, so use `LONG` input data with caution.

The standard deviation of a sample block is computed as: sum the squared deviations from expected value, divide by *<count>*. If you are using `VARIANCE` to estimate parameters of a normal distribution, the result will be slightly biased. You can correct the bias by pre-computing the correction factor *<count>*`/`*<count>*`-1` and applying this using a DAPL expression.

Note: `VARIANCE` is equivalent to `STDDEV` but without a square root operation.

Example

```
PIPES  P1W WORD, P2L LONG
VARIANCE (P1W, 10000, P2L)
```

Compute the variance of blocks of 10000 observations from `WORD` pipe `P1W` and place the variance result into `LONG` pipe `P2L`.

See Also

`AVERAGE`, `RMS`, `STDDEV`

VECTOR

Define a vector data element.

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 sometimes acceptable to place a backslash "\" continuation character at the end of continued lines, but doing so is not recommended. This combines all of the lines into a single long line for processing, and could overflow the maximum command line length.

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.

Example

```
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 | FLOAT PIPE | DOUBLE 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 | FLOAT PIPE | DOUBLE 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>*. The sample of the event is considered to be the one occurring exactly at or immediately after the external 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 types of the input and output pipes must match.

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

`DLIMIT`, `LIMIT`, `LOGIC`, `TRIGSCALE`

20. Previous Versions of DAPL

This section summarizes obsolete DAPL features that either cannot be supported now or will necessarily become unavailable in future releases.

The DAPL operating system has changed substantially in response to users' requirements. In the course of this development, 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 here, even if they seem to work. Where 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, equivalent or superior functionality is available using newer commands.

The DAPL 3000 system represents a major new development, and several commands that were in previous versions of the DAPL system and announced for retirement are unavailable in this DAPL 3000 system release.

System Commands Now Obsolete

BDOWNDLOAD	Download 16-bit commands using the DAPIO32 programming interface or a software utility such as CDLOAD32.EXE.
UPDATE (for input sampling)	This command was incorrectly named. Input procedures have nothing to do with output updating. The command is now called SAMPLE.

Processing Commands Now Obsolete

OFFSET	Use a DAPL expression.
FFT32	Use the FFT command. If you specify FFT32, what you will actually get is FFT except restricted to 16-bit data.
LCOPY	The LCOPY command can no longer provide latency control features. Processing is routed to the COPY command, which has no latency control. Alternative configuration options will eventually replace the intended functionality of LCOPY.
PID1	This command form is accepted, but it does not respond to the supervisory ramping and latching features. For those features, you must upgrade to the PID command and use the new PIDRAMP and PIDLATCH commands.
SCALE	Use a DAPL expression.
CHANGE	Use the DLIMIT command.

Processing Commands Now Retired

PRINT	Use a FORMAT command.
WAVEFORM	You can still generate all of the same waveforms. Just specify the type of waveform you want directly, e.g., SINEWAVE.
PWM	This was replaced by a new command PWM with the same name but different configuration parameters.
INTEGRATE, BINTEGRATE	These are replaced by the new RSUM command. All of the old capabilities except a reset using a software trigger are supported. The RSUM command supports all data types.
PEAK	This command could not be extended to accept all data types. It also had some uncorrectable problems that could in some circumstances cause data loss.

`EXTRACT`	Use a DAPL expression to mask the bits you want to keep, adjust their polarity, and shift them to any position you want. `bit =  Port&Mask >> Shift`
`SCAN` `LSCAN`	Not needed in the DAPL 3000 system.
`CABS`	Use a DAPL expression of the form `mag_sq = a*a + b*b`. Or, for a true complex value magnitude, use the `CMAG` command.

Channel List Lengths

DAPL version 2000 accepted an old notation on the `IDEFINE` command and the `ODEFINE` command specifying the number of channels. This notation is no longer supported. The `CHANNELS` or `GROUPS` command must be used instead.

Hardware Pin Notations

When using a counter-timer board and assigning a `CT` pin type to an input channel using a `SET` command, the keyword notation `ICLOCK` is required. The older notation with an arbitrary number code is no long accepted.

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.

Latency Control Options

The DAPL system command `OPTIONS BUFFERING=OFF` is treated as advisory by all tasks. It has no direct meaning to the DAPL system. 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.

`OPTIONS SCHEDULING` is no longer configurable, and this option is ignored. The entire DAPL 3000 system operates under a single unified scheduling scheme that is better than any of the previously available options.

Variables in Parameter Lists

In DAPL 2000, certain commands that require constant-valued parameters would accept variables. This was for compatibility with earlier versions of DAPL, but it was not strictly correct. Variables can change value at any time, but the DAPL 2000 tasks did not always detect and respond appropriately to the changes. In DAPL 3000, task parameters that are supposed to be constants must be constants, either explicit numerical values or named constant values. Variable names cannot be substituted.

Supporting Older Software

For supporting legacy applications that used DAPL 2000 or earlier system versions, contact Microstar Laboratories to obtain a copy of the software document pertinent to that generation of hardware and software.

21. 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. 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 Commands`
: 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 nothing 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 receiving simultaneously-captured data 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 and DAP hardware documentation..

`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.

`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, triggers, and downloaded modules.

`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 in groups of 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 output 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 input pins 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. An active input procedure really has only one input channel pipe, 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 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. Pipes are sometimes informally called *data streams*. 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.

`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 the DAPL system interpreter 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 scheduling strategy attempts to guarantee that the most urgent processing is completed quickly, and otherwise, all processing gets an opportunity to 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 so that other tasks at equal priority can run. 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 multiple tasks at the same priority do not excessively 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.

`Start Group`

All processes started by the list of items on a `START` command. Tasks within a start group are able to read copies of the data streams produced by other tasks within the start group.

`String`

In a DAPL configuration, a string is a sequence of characters enclosed in quotation marks `(" ").` DAPL enforces length constraints on the strings, strips out any unprintable character codes, and converts alphabetic characters in these strings to upper case. Strings used locally by commands in custom command module development follow the programming language conventions and can contain a mix of upper and lower case characters.

`Synchronize`

In the context of sampling and updating, synchronization means using a common time reference, usually a precision oscillator, to establish the instants at which data capture or output updating events occur. 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.

`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 timestamp 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 location of the trigger event in its data stream. 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. Built-in processing commands for the DAPL system apply saturation, as needed, because this correctly indicates the sign of the result, and does not produce large artificial jumps between very large positive and negative values.

`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