DAPL 3000 Extensions for xDAP Family

Supplement to DAPL 3000 Manual Version 1

Version 1.04

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

Contents

Table of Contents

1. Introduction

The DAPL 3000 system uses a configuration notation that is highly back-compatible with the DAPL 2000 system. With the introduction of the xDAP family of Data Acquisition Processors, however, the traditional configuration commands are no longer sufficient. They cannot completely control configurations of hardware supporting multiple channel selectors that operate in parallel. In addition, there are new operating modes for asynchronous input activity not covered by the usual clocked sampling configurations.

This manual covers the command variants added to the DAPL 3000 system to support these new features. No additional software systems are required to use these commands. They are already included in the DAPL 3000 system software that is installed with xDAP products.

The xDAP family devices also include embedded processors with a considerably larger reserve of processing capacity. This capacity can be exploited using new processing commands and variants that are covered in this manual.

2. New Notations

This discussion expands on the section "*General Rules for Command Syntax*" in the DAPL 3000 Manual, Chapter 2, "*Introduction to DAPL*."

Number Notations

DAPL 3000 configurations use fixed point and floating point number notations. These notations are very similar to the notations used in the C++ language, except that precision of the hardware I/O devices rather than the CPU architecture motivates the data type defaults.

Fixed-point numbers are represented by sequences of numeric characters, optionally preceded by a minus sign. A hexadecimal notation is also supported. Some examples:

```
16    -65535    301991898    0x0000FF00
```

Floating point numbers are represented by a sequence of numeric characters, optionally preceded by a minus sign, followed by a fraction or exponent notation. The fraction notation consists of a decimal point optionally followed by numeric character digits. The exponent notation consists of an optional fraction notation, the letter e, optional minus sign, and up to three numeric characters indicating a power of 10. Some examples:

```
16.0    -65535e0    526.11e22    -4.29144e-102
```

Since the DAPL 3000 system supports signed and unsigned numbers of various precision, it often must make assumptions about the appropriate data type when it encounters a number notation in a configuration script.

- Integer data types can have precision 8, 16, 32, or 64 bits. Data produced by input sampling or transferred through user-configured data pipes default to signed 16-bit integers. Integers used to define constant values or task parameters default to the unsigned data type of minimum precision, unless the constant value has an explicit minus sign, in which case it defaults to the signed integer type of minimum precision.

- Floating point numbers can have IEEE 754 single precision of 32 total bits, or double precision of 64 total bits. The default data type used to represent floating point numbers is 32-bit IEEE 754 floating point.

- The default data type used internally for *DAPL Expression* tasks is 64-bit IEEE 754 double-precision floating point.

Some contexts specify prior information about data types, and in these situations, more flexibility is allowed in the number notations. In the following examples, the DAPL system knows the data types specified for the defined `constant` elements `mylong`, `mydouble`, and `myfloat`, so it can represent the number value in the specified data type even if the notations are not completely consistent.

```
constant  mylong    long    = 100
constant  mydouble  double  = mylong
constant  myfloat   float   = 1
```

There are other contexts in which the data type cannot be determined in advance. The most common of these is explicit constant values in a task definition parameter list. In the following example, the **SKIP** command assumes initially that the three numeric parameters are unsigned 8-bit integers, since all three values can be properly represented by this data type.

```
SKIP ( inpipe,  99,1,99,  outpipe )
```

Later, when the **SKIP** command parameter requirements are compared to the parameters actually provided, it can be determined that the command needs 16-bit integer values, and these conversions are applied automatically.

If the default notations would result in an incorrect data type for a number, you can override the default data type assumptions using a suffix notation. The following are examples of type-coercion postfix notations:

```
99L, 99l     --  Represent the value as 32-bit integer
99f, 99.0F   --  Represent the value as 32-bit floating point
99.l, 99.0L  --  Represent the values as 64-bit floating point
```

Signal Routing Notations

An extended notation for DAPL 3000 input sampling configurations is available for xDAP models that have I/O modules mounted in one or more panel slots.

An xDAP can establish connections to signals received by a compatible a-Series analog expansion module mounted in a designated a-Series analog backplane slot. The addressing of analog expansion modules can vary with the xDAP model, but for any given model, the hardware manual for the xDAP will define certain *routing codes* telling the xDAP how to reach the signals on the expansion module. Check the hardware manual for your xDAP model to determine which expansion modules are present, where the connectors are located, and the routing codes to access the physical connectors.

For each expansion module, 8 differential input signals typically will be received, though this can depend on the modules. The analog signals will be identified by differential pin identifiers `d0, d1, etc`. That is, local pin numbering starts at zero on each module.

The syntax of the **SET** command for an xDAP input channel definition is extended to include the routing information prefix codes along with the pin identifier.

SET <channel> [<route_code>:] <input_pin> [<gain>]

For example, suppose that you want to address all 8 differential input pins on an expansion module that your xDAP manual says is addressed using routing code `c1`. Then you could encode your input definition something like the following.

```
IDEFINE   sampling
  CHANNELS  8
  SET  ipipe0  c1:d0
  SET  ipipe1  c1:d1
  SET  ipipe2  c1:d2
  SET  ipipe3  c1:d3
  SET  ipipe4  c1:d4
  SET  ipipe5  c1:d5
  SET  ipipe6  c1:d6
  SET  ipipe7  c1:d7
  SCAN   2.50
END
```

If you omit the *[<route_code>]* notation from a channel declaration, the signal routing defaults to the standard input connector on the xDAP front panel.

Multiple-Sampling Notations

The DAPL 3000 system supports multiple concurrent sampling configurations. This feature might not be available for all xDAP models, but higher-end models will support it.

Through multiple generations, the DAPL system has used a *channel pipe* notation such as `IPIPE1` to indicate that a data stream is a channel from the input channel pipe, delivering data from sampling. The data flow begins when the configuration starts. You could define more than one input sampling configuration, but at most one of them could run at any given time.

The restriction on running only one input sampling configuration is relaxed in DAPL 3000, which now allows multiple samplers to run concurrently — provided that they conform to certain restrictions to be discussed shortly. Concurrent sampling processes start their respective converter hardware devices simultaneously, within the limitations of very small digital propagation delays. The concurrent sampling processes continue using a common internal time reference, so they never diverge from each other, but they can operate on different time-scales or use different operating modes.

For example, an application might require intensive high-rate sampling on a few channels, while a lower rate would be sufficient for monitoring a relatively large number of channels for status information. If limited to a single sampling configuration, the scan interval would need to be very small to capture the *high-rate* data. But the same rate would then apply to the *slow* channels as well, wasting a lot of resources. If two sampling configurations are used, one could support the high-rate channels, while the other supports the low-rate samples independently, making much better use of resources.

To configure multiple input sampling, start by defining the input sampling procedures in an ordinary manner. For example, define one fast procedure and one slow one as follows:

```
// Fast simultaneous sampling
idefine  faster
  channels 2
  set  ipipe0  d0
  set  ipipe1  d2
  scan  4.0
end

// Slow sampling, simultaneous does not matter
idefine  slower
  channels  2
  set  ipipe0  d12
  set  ipipe1  d13
  scan  1000
end
```

For processing, this makes the usual `IPIPE` notations in the processing section ambiguous. Should these refer to the `faster` configuration, or to the `slower` one? To specify which, you need to use *qualified channel pipe references* such as the following.

```
// Mixed rate processing
  pdefine   fast_and_slow
  // High rate data
  copy( faster.ipipes(0,1), $BinOut )
  // Low rate data
  baverage( slower.ipipes(0,1),2, 20, Cp2Out )
end
```

The qualifier notation used to designate input channel pipes is similar to notations used for object composition in C++, Visual Basic, extended Pascal, Java, and many other programming languages.

Now, the restrictions.

1. With multiple independent input processes, if they happen to address the same input selector device, this would produce a conflict when they tried to connect to two different analog input signals simultaneously. To avoid hardware conflicts of this sort, at most one input sampling procedure can reference channels that go to any given input channel selector. For example, on xDAP 7420 there are 8 converter channels, and channel selectors for these converters route input signals from pins `d0` and `d1`, from pins `d2` and `d3`, from pins `d4` and `d5`, etc. Thus, if you reference pin `d0` in one input sampling procedure, you are forbidden from referencing either `d0` or `d1` in another procedure. This is the *hardware consistency condition*. Each input procedure must satisfy the hardware consistency condition before it can be started.

2. The notion of *start groups* is extended to include processing procedures that reference the input sampling configurations. A *start group* consists of the input sampling and processing configurations listed on one **`START`** command. Input procedures are called *designated input procedures* if referenced by name, explicitly, within one or more of the processing configurations listed on the **`START`** command, or if they were started by a separate **`START`** command and are already running.

3. If the start group lists an input procedure configuration that is not explicitly referenced by *qualified channel pipe references* anywhere in the list of new processing procedures, or in any currently running processing procedures, the DAPL system will attempt to start that input processing procedure as a *default input configuration*. A default input procedure will provide data for any input channel pipe references that do not use *qualified channel pipe* notations to explicitly designate an input procedure.

4. At most one input configuration can be started as a default input configuration. After a default input procedure is started, any attempt to start another one as default will be diagnosed as an error. It is not possible to start more processing procedures to "upgrade" an input procedure currently running as a default to make it into a designated procedure, so that some other configuration could serve as the default.

5. It is tricky but possible to start procedures using separate **`START`** commands. The input procedures must satisfy the *hardware consistency condition*, and no input procedure should be started prematurely as a *default input configuration* if that procedure is also needed as an explicitly designated procedure for processing to be started later.

6. Most applications are relatively simple, and they can use a **`START`** command that specifies no input or processing procedures. When this is done, the system will attempt to start all of the currently-defined procedures as one *start group*. All input sampling configurations referenced by *qualified channel pipe* notations, in any of the defined processing procedures, will be started as *designated input procedures*. If any unstarted input procedure remains, it will be started as a *default input configuration*.

Avoid defining sampling configurations that are never used. Suppose, for example, that a processing procedure P and input procedures A and B are defined. Processing procedure P uses only explicitly designated input channel pipes from A. A **START** command listing no procedure names will start the processing procedure P and the two input sampling procedures without error. Input procedure B, which is not explicitly referenced, is started as a default configuration. But, since the processing procedure P provides no tasks to read data from procedure B, the samples from procedure B accumulate in buffer memory, and can lead to a buffer memory overflow condition.

When using software triggering to detect events in a sample stream from one input procedure, and then selecting data from a sampling stream produced by another input procedure, you must use **TRIGSCALE** processing of the events to account for the differences between the data rates.

Be careful when transferring binary data from channels at different rates. This applies in general, but it is particularly easy to encounter rate problems when using multiple sampling processes. For the example of the faster and slower processing configurations, as shown previously, the following would lead to disaster:

```
// Take data from fast and slow sources at equal rates
BMERGE  ( faster:IPIPES(0,1), slower.IPIPES(0,1), 100, $BinOut )
```

The fast stream produces 250 samples for each sample produced by the slow stream. Thus, after the first transfer of 100 samples from each stream, 24900 samples from the fast stream remain backed up in memory, because there are no corresponding samples from the slow stream to pair with them for the **BMERGE** processing. Buffer memory will quickly overflow. One way to account for the rate differences is to force the transfer of data in the correct ratios.

```
// Transfer data blocks using a 250-to-1 ratio
NMERGE  ( 500, faster.IPIPES(0,1),  2, slower.IPIPES(0,1),  $BinOut )
```

Another way is to configure a supplemental communication pipe using DAPcell services, so that data produced at different rates use independent transfer paths.

```
// Transfer fast and slow data independently
COPY  ( faster.IPIPES(0,1),  $BinOut )   // fast data channels
COPY  ( slower.IPIPES(0,1), Cp2Out )     // slower supervisory channels
```

High-Rate Sampling Configurations

When configuring xDAP models that support extended sampling rate modes, there are some additional constraints for using channel notations. At the extended rates, the number of channels you can use is limited by the transfer capacity, not the number of hardware connections available. For example, if your xDAP model has 8 input channel selectors (simultaneous sampling in groups of 8) and supports an extended sampling rate of 2.0 million samples per second, but with an 8 million samples per second transfer rate limit, you can use no more than 4 channels at one time.

To achieve maximum rates, each signal channel must connect to a different channel selector device and separate converter unit. For example, on an xDAP 7420 model, channels d0, d2, d4, and d6 connect to separate channel selectors, hence they can operate in parallel and achieve the full rate with a SCAN interval of 0.5 microseconds. However, channels d0, d1, d2, and d3 connect to only two of the available channel selectors. If these channels operate at their maximum rate, each of the channel selectors must be used twice to capture the connected signals, and the maximum rate you can get for each channel is 1 million samples per second, limiting the SCAN interval to 1.0 microseconds.

If you don't need the maximum number of channels at the maximum rate, you can take advantage of *multi-sampling notations* described in the previous section to measure other channels at lower rates, using the available data transfer capacity to advantage. For example, an xDAP 7420 unit has an 8 million samples per second transfer capacity, but if only 2 channels are needed at their 2 million samples per second maximum rate, you have 4 million remaining samples per second capacity. That leaves 6 channel selectors free, each with two available input channels. You could measure 12 input channels at a 0.25 million samples per second rate, as discussed in the previous section on multi-rate sampling, which would use an additional 3 million samples per second of the available transfer capacity. This is a valid configuration because the total utilization of 7 million samples per second does not exceed the 8 million samples per second limit.

3. DAPL Expression Tasks

DAPL expression tasks provide highly configurable calculation options, applied on a value by value basis. This style of computing is less efficient than performing calculations on massive blocks of data. But in return, DAPL expression tasks make many things very easy to do.

The DAPL 3000 system supports much more versatile DAPL expression processing than previous versions of the DAPL system.

- more data types
- higher precision
- configurable data type conversions
- more kinds of operations
- *if-then-else* style conditional calculations

Form of a DAPL expression task definition

A DAPL expression task looks like an assignment statement from a typical programming language. Rather than a single action, however, it is an ongoing process that operates on streams of data. You can think of it as having a built-in `receive – process – send` loop that runs continuously.

The task definition consists of the following:

- a specification for the target destination to receive the results of calculations
- an equal sign operator indicating the transfer of data into the target destination
- an expression made from a combination of data sources and operators, defining the calculations to be performed to generate the data stream

Example:

```
// Calculate the average values of data taken from two signal
// channels from input sampling.
pTarget  =  (IPipe0 + Ipipe1)*0.5
```

Data sources for DAPL expression tasks

DAPL expression tasks accept data in the following forms:

- pipes (data streams)
- explicit literal constants
- symbolic shared constants
- symbolic shared variables

Scalar values (the constants and variables) are implicitly expanded into streams of data that can be combined with other data streams as calculations proceed.

When the DAPL expression makes multiple references to the same named element, the values for each reference are the same.

Example:

```
// An inefficient but valid way to calculate 3*P1
ThreeP1  =  P1 + P1 + P1
```

Data Elements and Types

Data types

All data values used in DAPL expressions, whether they are from shared scalar sources, data transfer pipes, or previous intermediate calculations, have data types. These data types are available for various kinds of elements within the DAPL system, but they can also appear within the context of DAPL expression tasks.

The supported data types are very similar to *representation specific* data types available in the C and C++ languages.

`int8`	signed two's complement 8-bit fixed point, the normal representation of sampled data
`uint8`	unsigned 8-bit fixed point
`int16`	signed two's complement 16-bit fixed point (like most sampled data)
`uint16`	unsigned 16-bit fixed point
`int32`	signed two's complement 32-bit fixed point
`uint32`	unsigned 32-bit fixed point
`int64`	signed two's complement 64-bit fixed point
`uint64`	unsigned 64-bit fixed point
`float`	32-bit IEEE floating point
`double`	64-bit IEEE floating point
`bool`	logic data type, 'false' or 'true'

Data destinations

A DAPL expression task delivers the stream of calculated values to the destination location indicated. A destination location can be one of the following.

- pipe (data stream, all values)
- symbolic shared scalar variable (last value placed into variable storage)

If the data type of the destination is not the same as the data type of the calculated result, an implicit cast converts the value to the data type of the destination. If the data type of the destination is not capable of representing the provided value, this is not an error, but it can mean a loss of information.

Example:

```
// Float value becomes double by transfer into a pipe of type double
pDoubPipe  = pFloatPipe * 9 / 5 + 32

// Result of of integer calculations reduced to 'true' or 'false'
boolVar  =  (IPipe7 - 10 * IPipe6)
```

Scalar elements

Scalar values can be obtained from a shared variable, a shared constant, or an explicit literal constant. Shared scalar elements must be previously defined by a **CONSTANT** or **VARIABLE** command. The definition assigns a name, data type, and value.

Examples:

```
1000      // explicit constant
OneK      // symbolic shared constant, previously defined
```

Data types from sources

Every data value used in a DAPL expression task has an associated data type. The data type of a previously-defined symbolic name determines the data type of the value obtained from that element. The data type of a pipe data stream determines the data type of the values obtained from that pipe. The data type for number notations is determined by default type-assignment rules, unless there is an explicit notation overriding the default.

Inferred data types of explicit constant notations

The data type of an explicit constant notation, if not specified by a special override notation such as a data type cast or postfix notation, is inferred from the represented value. Special notations are rarely necessary in DAPL expressions, since values without any special cast or postfix notations are automatically assigned an efficient data type.

Examples:

```
1.0         // implicit double, no explicit typing specified
-100        // implicit signed int8, no explicit typing specified
100ul       // explicit typing, data type forced to 32-bit unsigned
double(10)  // cast conversion from implicit integer to double
```

Integer data type postfix notations

You sometimes might want to force a specific data type for a term in a DAPL expression, rather than allowing the DAPL expression parser to select a data type automatically. There are two kinds of *postfix notations* that can be applied to explicit numeric constant notations for this purpose. These notations are placed at the end of an explicit numeric constant expression.

The first kind of postfix notation is very much like a C++ suffix. These notations apply additional properties to force the value to an unsigned form, or to extend the length of the representation.

<none>	allow the DAPL expression to select a minimal data type
`u`	force an unsigned data type (applied to 8-bit, 16-bit, 32-bit, 64-bit integers)
`i`	force a signed data type (applied to 8-bit, 16-bit, 32-bit, 64-bit integers)
`l`	force a longer data type (extend to 32-bit integer)
`ll`	force an extra long data type (extend to 64-bit integer)
`ul`	force both unsigned and a long data type (32-bit unsigned integer)
`ull`	force both unsigned and extra long 64-bit integer

Examples:

```
101         // implied 8-bit unsigned integer, no postfix
101i        // force to 16-bit signed integer, i postfix
101l        // explicit 32-bit signed integer, l postfix
100000ull   // explicit 64-bit unsigned integer, ull postfix
```

The second type of postfix notation restricts the data type to one specific representation. Most of these notations have no correspondence to C++.

`u8`	force an 8-bit unsigned integer data type
`i8`	force an 8-bit signed integer data type
`u16`	force a 16-bit unsigned integer data type
`i16`	force a 16-bit signed integer data type
`u32`	force a 32-bit unsigned integer data type
`i32`	force a 32-bit signed integer data type
`u64`	force a 64-bit unsigned integer data type
`i64`	force a 64-bit signed integer data type

If you provide a value expression using an explicit data type notation, but the value is not representable within the range of that specified data type, the notation is treated as a configuration error and the task will not run.

Floating point data type postfix notations

The precision of explicit floating point constants can be forced using the following postfix notations, following all of the other characters that define a floating point literal value.

<*none*>	64-bit floating point
`f`	32-bit floating point
`l`	64-bit floating point

Examples:

`202.0`	64-bit floating point (double by default)
`201.0f`	32-bit floating point (float forced by `f` postfix)
`203.0l`	64-bit floating point (double forced by `l` postfix)

There is little penalty for selecting double rather than float as the data type for calculations, but there are some processing efficiencies that result from consistently using one floating point data type.

Examples:

```
// Expression with all floating point calculations in float type
floatDegC  =  float(intDegF) * (5.0f/9.0f) -  32

// Expression with multiple conversions between double and float types
floatDegC  =  (intDegF * 5.0f)/9.0 - float(32)
```

Data type cast

A type casting operation has a notation and a meaning similar to a cast notation in the C++ language. Type casts result in a different representation of the specified value. In some cases, when the specified value does not conform to the restrictions of the new data type, value adjustments are also applied. Most applications can avoid explicit type casts and use the implied casts provided automatically by the DAPL expression compiler.

The two main differences between type casts and postfix notations are:

1. Type casts can be applied to symbolic values and intermediate sub-expressions as well as explicit scalar constant terms.

2. Type casts will produce a value result of some kind, even when the data type is unable to accurately represent the value given.

Casts can be applied to

- complete expressions
- sub-expressions
- streams
- boolean values
- symbolic or explicit scalar values

Type casts have the following form:

```
type-of-cast <new-data-type> ( value-to-cast )
```

The `new-data-type` can be any of the supported data types listed at the beginning of this *Data Elements and Types* section. The value to cast can be a single term, or a set of terms combined by operators.

Three types of casts are currently supported. These are:

1. `static_cast< >`

   The results closely conform to the type casts implemented by the C++ language. The values of bits are taken from the value to cast, and inserted into the storage provided for the new data type. If the new data type has more bits of precision, the value is extended or reduced.

   * signed integer types are extended to fill additional storage bits with a sign bit matching the sign bit of the original value

   * unsigned integer types are extended by filling additional storage with 0-bits

   * any integer types are reduced by eliminating any bits for which there is no available storage

   * float types are extended or reduced by finding the closest floating point representation in the specified new type, which sometimes has limiting or round-off error side effects

   Examples:

   ```
   static_cast<int32>( -16384 ) // Representable, value remains -16384
   static_cast<int16>( 36000 )  // Truncated to 16 bits, value -29536, caution
   static_cast<float>( -10.0e105 )  // Result -INF can preserve only sign
   ```

2. `saturate_cast< >`

   The results have a value as close to the specified value as possible. Casts of floating point values are the same as for `static_cast`. Casts of fixed-point numbers are saturated to the nearest high representable value or the nearest low representable value when the value to cast is beyond the representable range.

   This cast is free from some of the artificial and unexpected sign-reversal effects that can result when using `static_cast` operations to reduce the precision of a result. However, reduced efficiency is a side effect, since every calculated value must be compared to the range limits.

3. `bit_cast< >`

   The results are obtained by taking bits from the specified value, much like `static_cast`, but the bits are always treated like the bits of an unsigned integer. If the new data type provides more bits of storage, the extra high-order bit locations are always filled with 0-bits. If the new data type provides an insufficient number of bits, high-order bits are dropped until the remaining bits will fit in the storage available.

   Bit casts cannot be applied to values with a `float` or `double` data type.

When applied to an explicit constant notation, the cast has the same effect as if it were a separate operation applied to a previous valid numeric value. Intermediate results can be "folded together" when the expression is compiled prior to execution.

Simplified cast notations

While the C++ cast style gives full control over type conversions, it is simpler to let the cast style default to the ordinary `static_cast` form, and specify only the new data type you want. You can do this using the *function call* or *constructor* casting notation, similar to the notation supported in C++. To use these simplified casts, specify the name of the data type as if it were a function, and place the value to be converted inside of parentheses.

The following cast notations are supported:

`int8( )`	convert to 8-bit signed integer
`uint8( )`	convert to 8-bit unsigned integer
`int16( )`	convert to 16-bit signed integer
`uint16( )`	convert to 16-bit unsigned integer
`int32( )`	convert to 32-bit signed integer
`uint32( )`	convert to 32-bit unsigned integer
`int64( )`	convert to 64-bit signed integer
`uint64( )`	convert to 64-bit unsigned integer
`float( )`	convert to 32-bit floating point
`double( )`	convert to 64-bit floating point
`bool( )`	interpret as a 'true' or 'false' result

Examples:

```
int16( TWOPI )    // Truncate a pre-defined value to short integer
double( 1000 )    // Equivalent to 1000.0
bool( 10 )        // Equivalent to 'true' for nonzero value
```

Cast operations preserve sign and value when the new data type has compatible sign properties and sufficient precision.

Examples:

```
int32( -16384 )     // Result -16384 in 32-bit form
uint64( 15 )        // Result 15 represented in 64-bit form
double( -10.0e24 ) // Result -10.0e24 represented in double form
```

Sign preservation is not guaranteed if the value to cast is beyond the range of the specified data type.

Examples:

```
int32( -16384 )     // Representable, value remains -16384
int16( 36000 )      // Truncated to 16 bits, value -29536, use caution
float( -10.0e105 ) // Result -INF can preserve only sign
```

If you want to avoid peculiar sign reversal anomalies when you apply casts, you should use the *saturate_cast* notation instead of the simplified cast form.

Examples:

```
saturate_cast<uint32>( -16384 )   // Closest value 0
saturate_cast<int16>( 36000 )     // Closest value 32767
saturate_cast<float>( -10.0e105 ) // Closest value -1.0e37
```

Hexadecimal integer notations

Hexadecimal number notations typically are most useful for representing individual bits of a digital I/O port, but they can also be used to represent unsigned integer values. A hexadecimal number is indicated by the prefix characters `0x`, similar to the C and C++ programming languages. Following this are the digits for the base-16 number representation from `0` (zero) through `F` (fifteen).

A negative value can be indicated by placing a minus sign before the `0x` hexadecimal prefix. The final value is calculated as if a separate inversion operator is applied to the positive number.

Examples:

```
 0xFFFF         Value 65535
-0x0001         Value  -1
 0x0a           Value 10
```

Decimal integer notations

Explicit decimal notations represent unsigned integer values. The digits for the base-10 number representation range from 0 through 9. The base value is the numeric value before a minus sign is taken into account. A minus sign can precede the digits to indicate negative values, but it is treated as if it were as separate inversion operation applied to the base value.

Examples:

```
100         // positive integer
-320000     // large negative integer
-0x00FF     // equivalent to integer -255
```

A postfix notation that follows the base value notation causes the base value to be coerced to the specified type. If the base value is not consistent with the representable range of the specified new data type, the notation is incorrect and produces a configuration error as the script is compiled.

When there is no postfix notation, the base value is presumed to have the first data type in the following sequence able to correctly represent the unsigned value.

```
uint8
uint16
uint32
uint64
```

Examples:

```
 32760        // representable as 16-bit unsigned integer
0xCFF0        // representable as 16-bit unsigned integer
 32800        // representable as 16-bit unsigned integer
100000        // representable as a 32-bit unsigned integer
```

If the base value is prefixed by a minus sign, the resulting value is equivalent to forming the additive inverse of the base value. The assumed data type is then the first implied type in the following list capable of representing the negative value.

```
int8
int16
int32
int64
```

Examples:

```
-32760          // 16 bit signed integer type
-0x0FFF         // 16 bit signed integer type
-32770          // 32 bit signed integer type
-0xFFFF         // 32 bit signed integer type
-3270i64        // 64 bit signed integer type, forced by postfix
-0xFF000000     // 64 bit signed integer type
```

Floating point number notations

Floating point constants consist of an optional sign, a value specifier, an optional exponent scale factor, and an optional precision postfix notation.

For negative numbers, the notation begins with a minus sign.

Examples:

```
-5.0
10e6
```

The value specifier consists of some decimal digits specifying an integer part, a decimal point, and some additional digits for a a decimal fraction part. Decimal digits can appear before or after the decimal point, or both, but at least one digit is required.

Examples:

```
-5.
.001
0.6175
```

If an optional exponent scale factor is specified, the decimal point and fraction digits are not required. Otherwise, a decimal point is required.

Examples:

```
5e6
```

A floating point exponent scale factor notation consists of 'e', an optional sign, and an unsigned explicit decimal number specifying the exponent for a power of 10 scaling factor to multiply the base value. Hexadecimal notations are not allowed. Exponent values are limited by the range that floating point notations can represent.

Examples:

```
1.0e5
5.5e-19
```

Finally, an optional floating point suffix notation is allowed, to force the selection of a 32-bit or 64-bit floating point precision for the data type. The available options are:

<*none*>	64-bit floating point by default
`f`	32-bit floating point
`l`	64-bit floating point

Examples:

```
-6.28e-7l       (force double)
3.14159f        (force float)
```

Here are more examples:

```
 -32.05    // double constant, default type

 .0001     // double constant, fractional-valued

 -1.e-4    // equivalent to double constant -0.0001

  2e5      // decimal point is optional with 'e' notation

  2.e4     // equivalent to double constant 20000.0

  100.0f   // 32-bit 'float' constant without scaling, value 100.0

  2.6e3l   // 64-bit 'double' constant with scale term, value 2600.0
```

Operations and calculations

Operators specify the actions that are to be taken to combine the data elements specified and obtain calculated results. There are four categories of operations in DAPL expressions.

1. Unary prefix operations. These operations are applied to the value on the right side of the operator, before the modified value is used for further calculations. (The minus sign on an explicit constant can be considered a special case of a unary prefix operation.)

   Examples:
   ```
   - TERM
   ! TERM
   - ( group of calculations )
   ```

2. Infix operations. These operations combine two values to produce a new third value.

   Examples:
   ```
   P1 + 10
   BITS << 5
   ```

3. Conditional operations. These operations calculate three values, a predicate (boolean) and two consequents (other data), to select a result value.

   Examples:
   ```
   A>B ?  A : -32678
   ```

4. Grouping. These operations do not perform calculations, but instead they control the sequence in which other calculations are performed. The value and data type of a group is considered to be the final value and data type produced by the terms inside. Terms inside of a cast are treated much the same as terms inside grouping parentheses, except that an additional data type conversion operation is applied.

   Examples:
   ```
   ((P1 + 10)*2) << 4
   int16( DoubPipe + 1.e-13 )
   ```

Some general rules about what to expect when using operators:

1. Arithmetic operators attempt to preserve numerical value, sometimes at the expense of changes in data type.

2. Bitwise operators attempt to produce results with the number of bits matching the operand with the most bits precision.

3. Calculations involving only unsigned data in general produce unsigned results. When applying an operation to a mix of signed and unsigned data types, the result is deemed to have a signed data type, even for the case of bitwise operations where a numerical meaning of the result is unclear.

4. When padding a signed value to achieve a certain number of bits precision, the extension bits used (consistent with the rules of C and C++) will depend on the sign bit of the original value. This can cause unexpected bit values, and yield incongruous results, when there is a mix of signed and unsigned types. For the most consistent results, favor unsigned data types for performing bitwise logic operations. If you don't start with unsigned values, it is recommended to apply static casts or bit casts to obtain an unsigned type.

Implicit conversions for operations

Most operators have restrictions on the kinds of data they can use. The most common restriction is that the data types of the operand elements must be of the same data type. Data of other types will be converted by an implied cast first, to force the data types to be compatible (if this is possible).

The following kinds of implied conversions can be imposed, depending on the kind of operation applied.

1. Higher integer precision for one or both operands.
```
int8   -->  int16
uint8  -->  uint16
int16  -->  int32
uint16 -->  uint32
int32  -->  int64
uint32 -->  uint64
float  -->  double
```

2. Cast to a floating point type.
```
double with 'other type'  -->  cast 'other type' to double
float with 'other type' except double  -->  cast 'other type' to float
```

3. Mixed integer types preserving value.
```
mixed signed and unsigned --> cast to sufficiently large signed type
```

4. Mixed integer types preserving bit alignment.
```
mixed signed and unsigned --> cast to signed type of larger
```

5. Mixed boolean and arithmetic types.
```
arithmetic types  -->  bool type
bool type  -->  numerical type matching other argument
```

These implicit casts do not lose information. Casts from arithmetic types to `bool` types appear to lose information, but this information would have been discarded in any case when logic operators or final assignments are applied.

Unary operations

Unary operations are applied to the term to the right of the operator before any other calculations involving that term can be done. Casts receive the same treatment as unary operators.

1. `-` operator. Additive inversion. Forces negative values to positive, positive values to negative.

2. `~` operator. Bitwise complement. Toggles the value of all bit positions in integer.

3. `!` operator. Logical negation. Converts 'true' to 'false', or 'false' to 'true'. If applied to a numeric value, automatically casts to a `bool` value first.

Arithmetic operations

There are five binary infix operations that perform arithmetic combining the terms on each side of the operator.

1. + operator. Adds values, after promotions to compatible data types.

2. `-` operator. Subtracts values, after promotions to compatible data types. Result are signed.

3. `*` operator. Multiplies values, after promotions to compatible data types.

4. `/` operator. Divides values, after promotions to compatible data types.

5. `%` operator. Divides values after promotions to compatible data types, returning the remainder value that is not an integer multiple of the divisor.

Bitwise operations

There are three bitwise operations that treat the binary digits of two integer values as if they were individual boolean bits. These typically do not have a meaning as integer arithmetic, but they can be very useful for isolating individual bits from digital signals.

If the two integer argument types do not have the same length, one or both arguments are first cast implicitly into a compatible integer type having the necessary length.

1. `&` operator. Performs bitwise AND operations after promoting terms to compatible integer types. The result bits are 1 if and only if corresponding bits of both integers are 1.

2. `|` operator. Performs bitwise OR operations after promoting terms to compatible integer types. The result bits are 0 if and only if corresponding bits of both integers are 0.

3. `^` operator. Performs bitwise EXCLUSIVE OR operations after promoting terms to compatible integer types. The result bits are 1 if and only if corresponding bits of both integers are different.

Boolean logical operations

There are two boolean operations, applying to arguments of `bool` data type. If the arguments are of some other type, they are first implicitly cast into a `bool` type.

1. && operator. Performs a logical combination of the two arguments. The result is 'true' if and only if both of the arguments are 'true'.

2. || operator. Performs a logical combination of the two arguments. The result is 'false' if and only if both of the arguments are 'false'.

Bit shift operations

There are two operations that treat the integer value in the first argument as a sequence of bits, and shift the positions of bits in the first argument by the number of positions specified in the second argument. The second integer argument must have a non-negative integer value; if the data type is signed, an implied cast will force it to be unsigned.

1. << operator. This left-shifts the bits the positive integer number of positions specified by the second operator toward higher-order positions. For a small initial value, this is equivalent to multiplying by 2. Any vacated low-order bit positions are filled with 0 bits.

2. >> operator. This shifts the bits the positive integer number of positions specified by the second operator toward lower-order positions. Bits previously in lower-order positions are discarded. Vacated high order

bits are filled by replicating the sign bit. When the first argument is unsigned, the sign bit position implicitly is 0, but when the first argument is signed, the sign bit position depends on the value prior to applying the shift.

Comparison operations

There are six operations that compare the values on the two sides of the operator. The operations promote arguments to a compatible data type if necessary before testing. The result of the test is always a `bool` value, 'true' or 'false'.

1. < operator. The result is 'true' if and only if the numerical value of the first argument is strictly less than the numerical value of the second argument.
2. > operator. The result is 'true' if and only if the numerical value of the first argument is strictly greater than the numerical value of the second argument.
3. <= operator. The result is 'true' if and only if the numerical value of the first argument is less than or equal to the numerical value of the second argument.
4. >= operator. The result is 'true' if and only if the numerical value of the first argument is greater than or equal to the numerical value of the second argument.
5. == operator. The result is 'true' if and only if the value of the first argument is equal to the value of the second argument. This applies to `bool` or numerical arguments.
6. != operator. The result is 'true' if and only if the values of the two arguments are not equal. This applies to `bool` or numerical arguments.

Conditional operator

This is a compound (tertiary) operator that combines three arguments. The first argument is the predicate, which must be of type `bool`. If not of type `bool`, it is implicitly cast to that type first.

The value selected for the result depends on the value of the predicate. If the predicate has a value 'true', the resulting value is set equal to the first value expression (the second argument), and the value of the third argument is discarded. If the predicate has a value 'false', the resulting value is set equal to the second value expression (the third argument), and the value of the second argument is discarded.

The two value expressions could have any data type. However, to enforce the restriction that every operation produces an unambiguous data type, both value expressions must result in the same data type, which will then be the data type of the combined expression. The expressions are considered to be numerical values, and automatic casts are applied as necessary to put both values into a common data type. For very simple expressions, it can be obvious when the data types match. For complicated expressions, trying to predict the data types from the terms and operators can be extraordinarily difficult. Rather than doing that analysis, you are likely to know what kind of values to expect, and you can cast both value terms to the same appropriate data type.

Example:

```
// Take the larger value from two pipes, both uint16 data
( pipe1 > pipe2 )?  pipe1 : pipe2


// Tricky! The arithmetic can change one of the data types
test ?  pipeA16 :  (pipeB16+const16)

// Casts can select data types for complicated value expressions
test ? static_cast<uint16>(pipeA16) : static_cast<uint16>(pipeB16+const16)
```

Internal data type promotion

As sequences of operations are performed, the possible range of intermediate results from the calculations can grow. The DAPL expression processing is allowed to select a data type for intermediate results that is more likely to cover any possible requirements for expanding precision, if such a data type is available. It is usually not necessary to worry about intermediate types, except when calculations become really complicated. However, promotion of data types can't guarantee complete preservation of information in all possible expressions.

Example:

```
// The DAPL expression can select a 64-bit integer representation
pProduct = pA16 * pB16 * 32768/10000

// There is no guarantee that any data type can represent the result
pRisky = 16000 * pPipe1 * 16000 * pPipe2 * 16000 * pPipe3 * 16000
```

The following kinds of internal type promotions are allowed.

1. Short integer types are promoted to long integer types having the same signed or unsigned properties.

2. Operands converted to the same common data type before applying the operation.
   - double if either operand is double
   - double if one operand is float and the other is a 64-bit integer
   - float if either operand is float
   - a signed integer type of the size of the larger operand if either operand is signed
   - an unsigned integer type of the size of the larger operand if both operands are unsigned

3. For certain operations, the intermediate data type of the result is determined directly from the kind of operation and the common argument types.
   - integer subtraction and negation always results in a signed integer
   - shift operations always produce the type of the left operand
   - modulo operation always produce in the type of the left operand

4. For most arithmetic operations, the compilation of the command tracks whether it would be possible to produce an integer overflow condition as a result of combined operations. Promotion to a higher degree of internal precision is allowed, if needed, and if a suitable data type of higher precision is available. Otherwise, value limits can be imposed.

Casts involving booleans

When `bool` variables are cast to become a numerical data type, by an assignment, explicit cast, or implicit cast, the result of the cast has value 0 for 'false' and 1 for 'true'. This will be represented as values 0.0 or 1.0 for floating point.

Any numerical value can be cast to a `bool`. The boolean value is set to 'false' if the numerical value is zero or undefined, and is set to 'true' otherwise.

Examples:

```
// if numeric pipe contains zero, use -100, otherwise use +100
bool( p1 )?   -100 : 100

// negate the value from pipe p1 when boolean flag is false
flag*p1 - !flag*p1
```

Casts on final transfer

A DAPL expression task finishes its calculations by taking the final result value and placing this into the destination location indicated. If the data type of the destination is not the same as the data type of the calculated result, an implicit cast converts the value. If the data type of the target does not have sufficient precision to contain the result, this can mean a loss of information.

For back-compatibility, any cast automatically applied for this final transfer is a kind of hybrid between `saturate_cast` and `static_cast`. Signed values can be sign extended when transferred to larger data elements, or saturated when transferred to smaller ones. Unsigned values are generally treated as patterns of bits, transferring as many bits as the target can accept starting with low order bits. An exception is when the value to be transferred comes directly from a single term without any calculations; these cases apply a `saturate_cast.` If you do not want this behavior, apply the style of cast that you choose, and explicitly cast the final result to the data type of the target to avoid an automatically-supplied cast.

Incompatible terms

There are some combinations of operators and data types that cannot be made compatible. The following inconsistent combinations will result in configuration errors.

1. Any bitwise operation (shift, multi-bit OR, etc.) with a floating point value. There is no attempt to cast a floating point value to make it become an integer value so that this operation could be performed.

2. Any explicit postfix condition applied to an explicit numerical value that cannot be represented in the forced data type.

Combining multiple operations

Expressions can consist of a number of terms combined by a number of operators. Parenthesis grouping can be specified to bracket a sub-expression – a set of terms to be calculated first – thus controlling the order of evaluation.

Examples:

```
// Sum scaled terms from pipes P1, P2, and P3
pResult =  P1*10 + P2*2 + P3*20

// Multiply the sums and differences of values from two streams.
pResult =  (P1 + P2) * (P1 - P2)
```

Precedence and order of evaluation

Precedence rules for operators determine the order in which operations should be applied. Otherwise, expressions such as the following would be ambiguous.

```
pResult = P1  *  10+P2  *  2+P3  *  20
```

Is the expression above evaluated by performing the two additions, followed by three multiply operations in succession?

It might seem so, but if we rearrange the white space within the expression, we get the following equivalent expression.

```
pResult = P1*10  +  P2*2  +  P3*20
```

Now it looks like each value from a pipe is multiplied by a scalar value, and the calculation finishes with two addition operations. Clearly this is not the same thing. Which one is correct?

According to the precedence rules, only the second interpretation is valid. These rules are very similar to the precedence rules in C and other programming languages. "Highest precedence" is equivalent to "lowest level detail" in the sense that “low level details” must be decided before “higher-level operations” that depend on them can continue.

If there is a sequence of operations at the same level of precedence, those operations are performed in left-to-right order.

Example:

```
// The following two expressions are equivalent
P1 + P2 + 1000 - vShared
((P1 + P2) + 1000) - vShared
```

The complete set of precedence rules follows. The operations appearing sooner in the list must be applied before the operations that appear later. These follow the operator precedence rules of C and C++.

1. Operations involving parenthesized grouping or implied grouping of terms for casts, sub-expressions, or conditional evaluations

```
// the addition is first, then the division, then the cast
double( (p1+p2)/10 )
```

```
// the grouped sub-expressions are independently evaluated first
test ?   (a + b)/2  :  (a - b)/2
```

2. Unary arithmetic, bitwise, and boolean operations

```
       <cast> ( sub-expression )
      + <item>
      - <item>
      ~ <item>
      ! <item>

// Value is -15, not -5 -- left item is negated before subtraction
-10 - 5

// Value is 10, not 1 --  left term 'true' casts to 1 prior to multiply
!0 * 10
```

3. Multiplicative arithmetic operations

```
      <item> * <item>
      <item> / <item>
      <item> % <item>

// Value 101, not 110 -- the multiplication is performed first
10 * 10 + 1
```

4. Additive arithmetic operations

```
      <item> + <item>
      <item> - <item>

// Value is 2 * P1 value, plus 20. The addition is performed first.
P1 + 10<<1
```

5. Shifting bitwise arithmetic operations

```
      <item>  >>  <positions>
      <item>  <<  <positions>
```

Be very careful with this. It is not unusual for a “bit shift” operation to be interpreted as equivalent to a multiplication or a division by a power of 2. Even if the effect is the same as a multiply operation, the operator precedence is very different.

```
// Value 320, not 82. Both arithmetic operations performed before shift.
4 * 10 << 1 + 2

// A clearer equivalent expression using explicit grouping
(4 * 10)<<(1 + 2)
```

6. Inequality comparison operations

```
      <item>  >  <item>
      <item>  <  <item>
      <item>  >=  <item>
      <item>  <=  <item>

// Arithmetic first, then check value range, result 'true' or 'false'
P1 + 5 > 2 + P2
```

7. Equality comparison operations

```
<item>  ==  <item>
<item>  !=  <item>

// Multiplication is first, then inequalities are checked.
// Result is 'true' when both inequalities are satisfied.
P1 > 5  ==  2*P1 < 12
```

8. Bitwise AND operations

```
<item> & <item>

// Tricky, value 6, not 14. Multiplications first, then bit operations.
     2 * 15&7 * 1
// Clearer equivalent expression with explicit grouping.
     (2*15) & (7*1)
```

9. Bitwise XOR operations

```
<item> ^ <item>

// The two bitwise '&' operations are performed first
AVALUE & 0xFF ^ BVALUE & 0x0011
// Clearer equivalent expression with explicit grouping
(AVALUE & 0xFF) ^ (BVALUE & 0x0011)
```

10. Bitwise OR operations

```
<item> | <item>

// Always equals P2, the 2^2 term reduces to zero first
P2 | 2 ^ 2

// The bitwise XOR is calculated, then three terms combined by OR
AVALUE | 0x00C ^ BVALUE | 0x003
```

11. Logical AND operations

```
<item> && <item>

// Arithmetic and comparison tests done before logical combinations
P1/10 > 1  &&  1 > P2
// Expression with equivalent grouping
(P1/10 > 1)  &&  (1 > P2)
```

12. Logical OR operations

```
<item> || <item>

// Arithmetic and comparison tests done before logical combinations
P1/10 > 1  ||  1 > P2
// Tests on the left and right combined before evaluating logical OR
     P1 > 1 && P2 > 2  ||  P1 < -1 && P2 < -2
// Expression with equivalent grouping
     ((P1 > 1) && (P2 > 2))  ||  ((P1 < -1) && (P2 < -2))
// Value is always 'true' since OR operation is done last
     P1 && 0 || 1
```

13. Decision operations

```
      <test> ? <item> : <item>

  // All three sub-expressions fully evaluated before selection
  P1+P2 > 0   ?   P1>>1 ^ 0x0020  :   P2<<1 | 0x0001
  // Sub-expressions with cast for type safety
  P1+P2 > 0   ?  uint16(P1>>1 ^ 0x0020)  :  uint16(P2<<1 | 0x0001)
  // Invalid. Sub-expression can't change execution order within decision
       P1+P2 > 0   ?   ( P4>5 || P5<0  :  P5>10 )
```

Hazards and Side Effects

Visibility of results

When results are transferred to data pipes, every calculated result will be visible to other tasks that read data from the pipe.

When results are transferred to shared variables, the only visible values are the ones that happen to be stored in the variable at the moments when task scheduling allows other tasks to run. When changes occur quickly, there is no guarantee that the other tasks will see all the delivered values before they overwritten by further changes.

A seeming time-shift paradox can occur when time-critical information is exchanged among tasks using shared variables. (In theory, *all* DAPL processing tasks are exposed to these hazards, but the effects tend to be most noticeable with DAPL expression tasks.) Symptoms occur when one task establishes the value of a shared variable using data that by chance other tasks have not yet seen. Later, when another task (such as a DAPL expression) gets its chance to run, it processes the slightly older data and "catches up," but it does this using the variable value that was produced by the first task from newer data. The effects can appear as if "*a variable value change propagated backwards in time.*"

As an optimization, DAPL expression processing is allowed to discard data and calculations that other tasks cannot possibly see.

Saturation

Despite efforts to avoid overflow of numerical ranges, there are three situations that can lead to overflow hazards.

1. An explicit cast is specified, and the value to be type cast is beyond the range that can be represented by the specified data type.
2. The final result of the DAPL expression evaluation task is a value to be stored as a data type with insufficient range to store the calculated value.
3. A calculation runs out of numerical range when using the data type with the highest available precision.

When these conditions occur, a *saturated value* is assigned. The returned value will be one of the following:

- For a very large positive number or a positive infinity, substitute the largest representable positive number for the target data type.

  Example:

  ```
  // 16 bit pipe receives saturated values 32767
  pIntPipe  = 32800
  ```

- For a very large negative number or a negative infinity, substitute the lowest representable negative number available for the target data type.

  Example:

  ```
  // signed 32 bit pipe receives saturated values -2147483648
  pLongPipe  = -2150000000
  ```

- For negative results and an unsigned data type, substitute the value of 0.

  Example:

```
// unsigned 16 bit pipe cannot represent negative, receives 0
pUnsigneWordPipe = -1000
```

- For certain results with an infinite limiting value, substitute a saturated value with appropriate sign.
  Example:

```
// Ratio is undefined. Float pipe receives values 3.4e38.
pFloat  =  pData / 0
```

Loss of precision

Operations such as casts and transfers of a calculated final result can force floating point values to be converted to integer values. Sometimes these operations are harmless, but other times there are fractional portions of the value that cannot be represented in integer form.

These “rounding” operations will proceed without warning, but the potential problem will be diagnosed with a configuration warning when the DAPL expression task is specified.

Example:

```
// Integer pipe cannot represent 2 PI exactly, truncate to 6
pIntPipe  =  2.0 * PI
```

Roundoff effects

Rounding can occur with any floating point calculations, but the effects can be particularly visible when values are reduced to lower precision.

Example:

```
// Roundoff error is likely to cause unexpected 'true' boolean
boolVar  =  0.666666666667f - 2.0/3.0
```

Latency and efficiency

An approximate model for effort required per sample to process data in a DAPL expression task is

```
work per sample  =
    (block processing cost + scheduling cost) / N  +  calculation cost
```

where

`N` is the size of the data blocks processed by the DAPL expression task

block processing cost is large and approximately constant

scheduling cost is small and approximately constant

calculation cost is small and approximately constant

The worst case occurs when `N = 1`, and one output value is produced each time the DAPL expression task is scheduled for execution. This incurs the full block processing cost for each calculated result.

On the other hand, suppose that `N` is something more like 1000. The DAPL expression processing will be almost 1000 times more efficient than the `N=1` case. But this comes at a cost. To process 1000 values as a block, the 1000 values must exist in memory. But if they exist in memory, that means a corresponding time delay to collect together the data needed to perform the processing.

DAPL expressions do not know which is more important to you, to deliver small amounts of results very quickly, or large blocks of results very efficiently. There is no attempt to predict your intent. The expressions will process the data in whatever amounts happen to be available, each time execution is scheduled. If you need quickest real-time response, at the expense of processing efficiency and overall processing rate, you can force this by restricting the buffering size of the data streams passed to the DAPL expression task. (See the **PIPES** command.)

Example applications

* 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 ^ 0x0000000F
  ```

* Produce 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
  ```

* 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. Determine calibration coefficients `ST1`, `ST2`, and `ST3`. Provide them as variables. Then apply them to the measured strain voltage stream `Vmeas`.

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

* Detect large noise spikes in a signal.

  Apply a highpass filter to the signal. The noise will pass through this filter easily, but lower frequencies will not. Test whether filtered signal values are large, and, if so, assign the value 1 to a pipe. Otherwise, assign the value 0 to a pipe.

  ```
  intFlagPipe =  pHighPassed*pHighPassed > NoiseThresh
  ```

* Simulate real-time signal samples by combining multiple waveforms and clock.

  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
  ```

* Preserve a running estimate of a steady value in a shared variable.

  While it is possible to perform digital filtering to smooth data and estimate a steady value, filter calculations that extend into back history for a very long time tend to be either resource hungry or numerically ill-conditioned. You can implement simple exponential filtering (first-order lag filtering) using DAPL expression processing and one `double` shared variable. Select the value `WINDOW` that is roughly the number of past history samples that you want the estimate to span. Because changes are small and accumulate slowly, there is no harm if tasks see changed values at slightly unpredictable times.

  ```
  vSteady = ((WINDOW-1)*vSteady + double(pNewMeasurement) / WINDOW
  ```

* Use a variable value under control of the host application to continuously adjust between highly smoothed and unsmoothed measurement data.

  Configure the DAPL system to process the raw data with an aggressive FIRLOWPASS filter, preserving the original data rate. Provide a shared floating point variable under control of the host application. Set it to 1.0 initially, which means to "send everything" from the raw signal. Setting the variable to 0.0 uses the signal with maximum filtering. Intermediate settings interpolate between the two cases.

  ```
  iAdjFilt = (fLevel)*pUnfiltered + (1.0f-fLevel)*pFiltered
  ```

* 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 `IPipe(5)`. To make the bits from the digital port visible, the host sends the DAPL system a 'LET HOSTMASK = 0x`ffff'` command. This allows the bits received from the digital port to pass through to `maskedpipe` unchanged.

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

* Use a bit on a digital port to determine which of two signals sends data to the host system.

  Use bit masking logic to select the desired bit position from data received from the digital input port via logical channel IPipe7. If the bit is clear, echo the data from input channel IPipe0. If the bit is set, echo the data from input channel IPipe1. Discard the data values not selected.

  ```
  ( Ipipe7 & 0x0004 ) ?  IPipe1  :  IPipe0
  ```

4. New Commands and Variants

This chapter provides command reference pages describing in detail the features of special commands and command variants used by the DAPL 3000 system to support the xDAP family of data acquisition products. The material in this chapter supersedes some of the material presented in the main DAPL 3000 Manual.

BIRAMP

Define a task that generates bidirectional linearly-ramped data sequences.

BIRAMP *(<range_1>, <range_2>, <samps_1>, <samps_2>, [<init_phase>,] <out_pipe>)*

Parameters

<range_1>
The initial extreme value of the output sequence.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<range_2>
The other extreme value of the output sequence.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<samps_1>
The number of sample intervals to transition from the first range limit value to the second.
`WORD CONSTANT | LONG CONSTANT`

<samps_2>
The number of sample intervals to return from the second range limit value to the first.
`WORD CONSTANT | LONG CONSTANT`

<init_phase>
Initial phase shift specifier, in sample interval units.
`FLOAT CONSTANT | DOUBLE CONSTANT`

<out_pipe>
Pipe for output data sequence.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

A **BIRAMP** command defines a task that generates a data sequence ramping at a constant rate from the first range limit *<range_1>* to another range limit *<range_2>*, then returning at another constant rate to the first limit. The resulting data are placed into *<out_pipe>*. The remaining parameters define the characteristics of the ramp sequences. The most common application is to sweep a digital-to-analog converter from one voltage value smoothly through a range of values.

The limit value *<range_1>* specifies the initial output, where the first ramping transition begins. *<range_2>* specifies the value where the direction of the slope reverses. If *<range_2>* is chosen to be greater than *<range_1>,* the initial ramp begins in an increasing direction, and the return ramp is decreasing. Or, if *<range_2>* is chosen to be less than *<range_1>,* the initial ramp begins in a decreasing direction, and the return ramp is increasing. Unless specialized phase overrides are applied, the generated waveform starts at the *<range_1>* limit value and reaches both specified limit values.

The durations for ramping are specified in units of sample intervals. *<samps_1>* specifies the number of samples spanned by the first ramping sequence, and *<samps_2>* specifies the number of samples for the return sequence. Both values must be positive integers. The first ramping sequence starts at the *<range_1>* limit value, and produces *<samps_1>* output values. The returning ramp sequence begins with the second extreme value *<range_2>,* and produces *<samps_2>* output values. The next operating cycle begins at the *<range_1>* limit

value again. This pattern continues. The total number of samples for each operating cycle is the sum <*samps_1*> + <*samps_2*>.

For some special cases, it may be desired to start the first ramping sequence at a different value than the first range limit level. This can be done by specifying an override value, positive or negative, for the optional <*init_phase*> parameter. The shift value can be no larger in magnitude than the number of samples in the operating cycle. The effect of a positive phase adjustment is as if the specified number of samples is bypassed to reach the point where the first data evaluation is applied. Since the waveform is cyclic, a positive shift of "*operating cycle length minus **N** locations*" is equivalent to a negative shift of "*minus **N***" locations. When omitted, the default value for the <*init_phase*> parameter is 0.0 samples, which corresponds to a first evaluation occurring exactly at the first range limit point. The <*init_phase*> adjustment is a floating point specification, which means that it can specify either an exact integer or a fractional number of sample positions. If the phase override is not an exact integer value, this can displace the entire sequence so that the ramped data values might no longer attain the range limit values exactly.

Though similar to some waveform generation commands, the **BIRAMP** output data sequences are typically easier to configure to exactly cover certain specific values. The **TRIANGLE** or **SAWTOOTH** commands attempt to preserve correct waveform harmonic properties, but generated values might not include the exact values specified as range limits. A **BIRAMP** sequence is similar to a **TRIANGLE** wave, except that a triangle wave is constrained to ramp upward and downward at the same rates. The **BIRAMP** sequence is similar to a **SAWTOOTH** wave, except that a sawtooth wave has only one ramped slope and a discontinuity point that can sometimes lead to unexpected values.

Any output data type can be selected for the output pipe. However, whichever data type you select, use the same data type to specify the range limit value parameters.

Examples

```
BIRAMP(0,32767, 32767,1, PRAMP)
```

Generate a data sequence that spans the non-negative output range of a 16-bit D-to-A output converter, one converter tick at a time. The initial ramp increases from 0 to 32766, one unit at a time, producing 32767 values. The return ramp begins at the upper limit value 32767. This return ramp produces only one sample value, the value 32767 itself. The next cycle will begin by dropping immediately to value 0. Each complete cycle spans 32768 samples. The data going out to the D-to-A converter are placed in `WORD` pipe PRAMP.

```
BIRAMP(-10000.0f,10000.0f, 256,256, 128.0, PTEST)
```

Produce a cyclic ramped signal with floating point values that ranges from -10000.0 to 10000.0 and then back again. The generated data are placed into the PTEST pipe for `FLOAT` data type. The up and down ramps are equal length, with 256 samples each, for a total cycle duration of 512 samples. The specified "phase" offset of 128.0 sample positions is exactly halfway through the 256 sample up-slope interval, so the first sample evaluation will yield a value of 0.0 exactly, and output values start to increase from there to the upper range limit. The waveform reaches both of the specified range limits exactly.

```
BIRAMP(-5000,5000, 1000,100, OPipe0)
SKIP(IPipe0, 1100,1001,99, PRAMP)
```

Drive a system that is somewhat prone to transient oscillations. The desired analog drive values will range from -5000 to +5000, slowly, with increases of 10 counts at each of the 1000 updates during the up-ramp. On the down ramp, there are 100 steps of 100 output converter ticks per step. This is 10 times faster than the up-ramp, but intended to be gradual enough to avoid exciting undesirable transient response. The data are sent directly to the digital-to-analog converter channel OPipe0, with updates delivered by a clocked output procedure. The response data are observed at input sampling channel Ipipe0, at the same rate as the output updates. The **SKIP** command discards the entire first full cycle of 1100 points, anticipating an undesirable initial transient there. On the next up-ramp interval of 1000 points, the desired data will span from one extreme value to just short of the other, which is covered by the following down-ramp. Thus, the **SKIP** command retains the 1000 up-ramp points plus one more, then discards the remaining 99 samples from the down-ramp to account for the rest of the data cycle.

See Also

TRIANGLE, **SAWTOOTH**

CORRELATE

Define a task that calculates a cross-correlation profile between two data streams.

CORRELATE *(<p1>, <p2>, <n_lead>, <n_lag>, <n_block>, <pOut>)*

Parameters

<p1>
First signal input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<p2>
Second signal input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<n_lead>
The number of leading position shifts of the second stream to analyze.
`WORD CONSTANT | LONG CONSTANT`

<n_lag>
The number of lagging position shifts of the second stream to analyze.
`WORD CONSTANT | LONG CONSTANT`

<n_block>
The number of new samples (data block size) to use for each analysis.
`WORD CONSTANT | LONG CONSTANT`

<pOut>
Output pipe for list of correlation values.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

The **CORRELATE** command computes cross-correlation between two real-valued streams of data, `<p1>` and `<p2>`. The data streams are analyzed in blocks, taking a data block of length `<n_block>` from each stream. These streams can have any data type, but the two data types must match. For these two blocks, a correlation analysis is applied at a number of shift positions specified by the `<n_lead>` and `<n_lag>` parameters, plus one more position for which there is no data shift. The results of the analysis are returned in pipe `<pOut>`, delivering `<n_lead> + <n_lag> + 1` output terms. For best statistical results, `<n_block>` should be much larger than `<n_lead> + <n_lag> + 1`. Outputs can be any data type, independent of the input data type, but floating point data types typically offer the best tradeoffs for precision and range. The analysis repeats for each set of input blocks.

The cross correlation is defined by

$$corr(p1,p2;\ k) = E_i\ (\ p1(i) * p2(i - k)\ )$$

where

`p1` and `p2` are the data streams from the two input pipes

E_i is the expected value operator, spanning all terms indexed by `i` in the blocks of input data

`k` is a shift to be applied to the second data stream before performing calculations.

In effect, the terms in the first data stream are unshifted, while the terms in the second data stream are shifted some number of sample positions indicated by `k`. After this temporary realignment, term-by-term pairwise products are calculated and averaged. Signals with a strong linear relationship will show strong correlations. Signals that are statistically independent will show only "random noise" correlation at any lead or lag position.

Interpretation of the shift index `k` can be a little tricky. Notice how a larger index on the p1 and p2 data sequences means using data that arrived at a later sample time, hence a *time delay*. Increasing the value of the `k` shift, because of the minus operator applied to the `k` term, reduces the data index and thus represents a *time advance*. Correspondingly, a more negative value of the `k` shift corresponds to a more positive net index, and thus represents a *time delay*. Results are presented as correlations from the most negative index (the largest lead time *advance* applied to the second signal relative to the reference) to the most positive index (the largest lag time *delay* applied to the second signal relative to the reference).

Though the operation is described in terms of averaged two-term products, the problem can be recast in the form of a convolution, which can then be calculated using FFT methods. This is what the **CORRELATE** command does. The most efficient evaluations happen to occur when `<n_block> + <n_lead> + <n_lag> + 1` is a size directly suited for an FFT operation. However, this will rarely matter, since the FFT size will be rounded up automatically to the next suitable value if necessary.

Here are examples of applications that can benefit from a **CORRELATE** analysis.

1. Search for a pattern match. The first signal is generated from repeated copies of the same *pattern vector*. If the second signal shows a strong positive correlation to the reference pattern at shift position `k,` this indicates the presence of a matching pattern at that location.
2. Response time. For systems in which a response signal is expected to be very similar to a stimulus signal, but attenuated, delayed, and possibly inverted, the amount of shift in the correlation result indicates the response delay.
3. Crosstalk analysis. Determine whether one of your signal lines is coupling stray energy into another signal line by driving the first line firmly with a reference signal, while the second channel line is terminated to ground and not driven. Look for correlation between the signals from the two lines.

Correlations are in general statistical estimates, hence calculated results show random variations. For best results, use a relatively large number of samples and calculate a relatively limited number of lead and lag positions.

Example

```
CORRELATE(IPipe0,IPipe1,47,2,4000,pCorr)
```

Take the data samples for two signals directly from input sample channels `IPipe0` and `Ipipe1.` The second signal is expected to be lagging the first signal significantly, so a relatively large number of lead shifts is expected to be necessary to align it to the first signal for maximum correlation. Specify a correlation analysis for 47 lead shifts and 2 lag shifts applied to the second data channel, which will result in 47+2+1 = 50 correlation figures returning per block of 4000 samples analyzed from each input channel. Place the correlation values into pipe `pCorr`.

See Also

MIXRFFT

COSINEWAVE

Define a task that generates cosine wave data.

COSINEWAVE *(<pk_min>, <pk_max>, <wave_freq>, [<init_phs>,] <samp_time>, <out_pipe>, [<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])*

COSINEWAVE *(<amplitude>, <period>, <out_pipe>, [<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])*

Parameters

The parameters for a **COSINEWAVE** command are shared by several waveform commands. The common parameter description is provided in the **WAVEFORMS** command description page.

Description

A **COSINEWAVE** command defines a task that generates cosine function data, and places the data in *<out_pipe>*. The sequence is obtained starting from a continuous-time version of the waveform, scaled and phase shifted according to the waveform parameters. Locations of samples are selected along that path, according to the timing parameter *<period>*, or the combination *<wave_freq>* and *<samp_time>*. For a balanced waveform with no phase shift, the output values start at the positive peak value.

Note: A **COSINEWAVE** configured with an initial phase angle -0.5 (implicitly, -0.5 π) is equivalent to a **SINEWAVE**.

Note: A **COSINEWAVE** command configured with the same parameters as a **SINEWAVE** command generates data that can be used for quadrature modulation.

Examples

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

Generate a balanced cosine wave with values ranging from -1000 to 1000, and a period of 100 sample intervals, in 16-bit format. The first output value generated will be 1000. Place the waveform data into pipe P2.

```
COSINEWAVE(0.0, 1.0, 60.0, -1.0, 5.0, PWAVE)
```

Generate samples of a “cosine-on-a-platform” data sequence. The waveform ranges from 0.0 to 1.0. The implicit amplitude of the cosine wave portion is 0.5, one half of the peak-to-peak range. The implicit DC offset is equal to (0.0 + 1.0) / 2.0, or 0.5. The initial phase angle of -1.0 (-1.0 π radians) inverts the cosine portion. The initial value of the inverted cosine wave and the DC offset cancel, so that the first output value equals 0.0, the low limit of the output range. The frequency of the cosine portion is 60.0 Hz when the specified updating interval of 5.0 microseconds is used. The resulting waveform data in a floating point format are placed into pipe PWAVE.

See Also

SINEWAVE, TRIANGLE, SAWTOOTH, SQUAREWAVE

CROSSPOWER

Define a task that calculates a crosspower spectrum from two real-valued data streams.

CROSSPOWER *(<p1>, <p2>, <N>, [<w_key> | <w_vect>], <out_Re>, <out_Im> [, <out_Auto>])*

Parameters

<in_pipe1>
First signal input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<in_pipe2>
Second signal input data pipe.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<N>
The number of terms to use from each signal for each analysis.
`WORD CONSTANT | LONG CONSTANT`

<w_key>
A keyword option to use a predefined window rather than an explicitly defined one.
`KEYWORD`

<w_vect>
A vector defining explicit window multiplier terms to be used by FFT processing.
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<out_Re>
Output pipe for real-valued component of crosspower spectrum.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_Im>
Output pipe for imaginary-valued component of crosspower spectrum.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<out_Auto>
Optional output pipe for real-valued autopower spectrum of first signal.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

The **CROSSPOWER** command computes a crosspower spectrum for two real-valued streams of data, *<p1>* and *<p2>*. Data are analyzed in blocks of length *<N>*. The data types for these two pipes must match. Optionally, the **CROSSPOWER** command can also calculate an autopower spectrum for the first data stream.

For each frequency indexed by k in the Discrete Fourier Transform of input signals x and y, the value of the crosspower spectrum is given by

$$\text{crosspower}(x,y;\ k) = [\text{DFT}^{\wedge}\ (x)]_k * [\text{DFT}\ (y)]_k$$

where

- the x and y terms represent equal-length blocks of data.
- the k (frequency) index selects terms from the spectrum.
- the caret notation indicates a complex conjugate.

Two FFT operations are required to produce the two blocks of DFT spectrum data. These are then combined pairwise, algebraically. Each term is complex-valued, so the pipes <out_Re> and <out_Im> are required to return the real and imaginary parts respectively. If the optional third output pipe is specified, the above crosspower formula is applied to the special case of signal x and itself, producing *autopower* spectrum terms, which are placed into pipe <*out_Auto*>.

$$\text{autopower}(x;\ k) = \text{crosspower}(x,x;\ k) = [\text{DFT}^{\wedge}\ (x)]_k * [\text{DFT}\ (x)]_k$$

From this formula, it is apparent that autopower is always real-valued and requires only one output pipe. The same autopower result could be obtained using the **MIXRFFT** command with the POWER post-processing option. The pipes <*out_Re*>, <*out_Im*>, and <*out_Auto*> must all be of the same data type. When selecting a data type for the output data streams, beware of the limited range of fixed point data, particularly WORD data. It is very easy to obtain crosspower or autopower terms that exceed the representable range.

The **CROSSPOWER** command supplies the FFT calculations internally. The parameter <*N*> specifies the length of each data block to analyze. There are many acceptable values for parameter <*N*>, but with some restrictions. Most "nice" sizes that you might want are acceptable: 256, 500, 720, 1000, etc. See the discussion of the **MIXRFFT** command for complete information about restrictions on the block size.

Chopping a signal into data blocks can have side effects, in the form of spurious smears in the autopower and crosspower spectra. These side effects can be diminished by applying a window operation to the data blocks before the FFT transforms are calculated. While usually producing more benefits than harm, windowing has its own side effects of attenuating spectrum results. Fortunately, the windowing side effects are predictable to a certain extent.

There are two ways to specify windowing, and you must select one of them.. You can use the <*w_key*> option and specify one of the following keywords to select one of the most popular window types. These keywords have no quote delimiters.

```
RECTANGULAR
BARTLETT
VONHANN
HAMMING
BLACKMAN
```

The RECTANGULAR window is equivalent to chopping the stream into blocks without any numerical modifications – in effect, "*no windowing is applied*."

Advanced users who want some other kind of window can specify a window term-by-term, as a DAPL system VECTOR element. Then, specify that vector's name as the <*w_vect*> option, instead of using the <*w_key*> option. See the discussion of the **MIXRFFT** command for details about windowing for FFT calculations.

For purposes of determining a frequency domain transfer function response of a system output signal y to a system input signal x, observe that the ratio of the crosspower spectrum and autopower spectrum reduces to

$$[\text{DFT}^{\wedge}\ (x)]_k * [\text{DFT}\ (y)]_k\ /\ [\text{DFT}^{\wedge}\ (x)]_k * [\text{DFT}\ (x)]_k = [\text{DFT}\ (y)]_k\ /\ [\text{DFT}\ (x)]_k$$

which equals the transfer function. The **TFUNCTION2** command can be used to calculate these ratios.

Example

```
CROSSPOWER(IPipe0,IPipe1,1000,HAMMING,PFreal,PFimag)
```

Take the data samples for two signals directly from input sample channels `IPipe0` and `Ipipe1.` Perform a crosspower analysis on a block of 1000 samples from each channel, reducing block truncation effects by applying a Hamming window before the FFT transforms. Place the crosspower results into floating point data pipes `PFreal` and `Pfimag,` omitting the optional autopower terms for the first signal.

See Also

MIXRFFT, TFUNCTION2

DEXPAND

Define a task that encodes multiple channels for clocked output via output expansion.

DEXPAND *(<in_pipe>, <output_vector>, <out_pipe>, <type>)*

Parameters

<in_pipe>
Input word pipe.
WORD PIPE

<output_vector>
A vector containing a list of the output expansion addresses to which data are sent.
VECTOR

<out_pipe>
A pipe to receive the encoded data stream, typically an output channel pipe.
WORD PIPE

<type>
A string parameter specifying the family of the output expansion board.
STRING CONSTANT

Description

The **DEXPAND** command encodes data and address information for transfer to an expansion board through the Data Acquisition Processor digital port. Designated digital or analog I/O expansion boards can receive the encoded data. *<output_vector>* is a vector containing a list of the output ports to which data are sent. *<in_pipe>* is a word pipe that provides the data to be sent. Multiplexed data from the input pipe must be presented in the order of the output ports listed in *<output_vector>*. For each data word read from *<in_pipe>*, two to four encoded words are generated, with these words including both channel addressing information and the data. The encoded data are written to *<out_pipe>*, which is typically an output channel pipe configured to digital output port B0, the port used to deliver data to expansion boards.

The *<output_vector>* specifies a list of output ports. The port numbers must be within the range 0 through 63 as supported by the expansion boards. See your expansion board manual for more information about port addressing.

The *<type>* parameter specifies the type of output expansion board. The type specifier must be one of the following string keywords, enclosed in double-quotes.

"SI"	Use this when the expansion board is one of the "SI series" isolation boards
"ASERIES"	Use this for all other expansion boards

The encoding produced by the **DEXPAND** command is used only for clocked output procedures. Configurations that bypass hardware clock control and deliver outputs asynchronously using a **DACOUT** or **DIGITALOUT** command do not need the encoding. These two commands generate their own encoding and timing internally.

An **OUTPORT** command is required for output expansion ports, both clocked and asynchronous, and must always be present when you use a **DEXPAND** processing command.

The **DEXPAND** command is necessary only for certain I/O boards, certain DAP models, and certain versions of the DAPL system. You need to check the manual for your particular board model to see whether you need encoding. If you do not need it, you must not use it. If present in the data stream but not expected by the processing, various encoding bits will be interpreted incorrectly as data.

You cannot use output expansion boards with a mix of encoding protocols on the same DAP board at the same time. Boards using one protocol could incorrectly respond to the encoding bits intended for other boards.

Note: The encoding generates a data stream with multiple terms used to transfer one value. If an output procedure is stopped before a multiple-value group is completely delivered, and then another output configuration is started, the first value written to the output expansion port may be invalid.

Example

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

Prepare multiplexed data from pipe `P1` for synchronized analog updating. The data are in groups of four values, which are to be encoded and sent through the digital port hardware to expansion ports 4, 5, 6, and 7 on the expansion board. Encoded data are delivered to the Data Acquisition Processor's digital connector for clock-controlled output through output channel pipe `OPIPE0`. The output expansion board is from the "SI" isolated signal interface board series.

See Also

DACOUT, **DIGITALOUT**, **OUTPORT**

DIGITALIN

Define a task to observe digital input port signal levels independent of input clocking hardware.

DIGITALIN *(<dest>, <port_number> [, <interval>])*

Parameters

<dest>
Where the bit patterns copied from the input port are placed.
`WORD PIPE | WORD VARIABLE`

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

<interval>
Optional interval in microseconds between successive reads of digital port.
`WORD CONSTANT | LONG CONSTANT`

Description

The **DIGITALIN** command enables inspection of bit values present on a digital input port. This activity does not interfere with timed data capture activities, such as multi-channel data capture at high-rates. It does require that the hardware on your Data Acquisition Processor device supports asynchronous access to certain digital hardware ports, however; so check the hardware documentation for your particular device before using this command.

The bit patterns observed by reading the digital input port are placed into the location specified by the *<dest>* parameter. This can be a *data pipe*, or a *shared variable*. If the data are placed into a pipe, a continuing record of all the observed values can be processed. If the data are placed into a variable, only the most recent bit values can be examined by other tasks.

The *<port_number>* parameter specifies a hardware address for the digital input port from which the digital bit patterns are copied. This address must be one of the port addresses for asynchronous input operations supported by your Data Acquisition Processor device. Typically, this port address is 0.

The optional *<interval>* parameter specifies a time interval, in microseconds, that occurs between successive reads of the digital input port. If you do not specify an interval, the command will read the digital port once at each opportunity that the DAPL system allows the **DIGITALIN** command processing to run. If you specify an interval, the DAPL system will not schedule the command processing to run again until after the interval is completed, bounding the amount of processing resources used for watching the digital input port.

A typical application of the **DIGITALIN** command is for triggering data capture processing without using any of the high-rate data channels for the triggering signal, and without any direct involvement of the host system.

Unlike the hardware-driven timing of an input processing procedure, the *<interval>* parameter does not specify exact time intervals. The actual instants at which the digital port is observed are somewhat unpredictable, dependent on the scheduling of other higher-priority or equal-priority tasks. The overall rate should be correct as long as the system is not overloaded.

Examples

```
DIGITALIN (VEVENT, 0, 1000)
PCASSERT (VEVENT, T_START, IPIPE0)
```

Use the command to check the status of digital port 0 once every 1000 microseconds, placing the latest observed bit pattern into the `VEVENT` variable. If any bit is nonzero, this will activate the `PCASSERT` task to post an event timestamp in the `T_START` software trigger, noting the location in the input data reference stream `IPIPE0` so that other tasks can find the selected data.

See Also

DIGITALOUT

DISCARD

Define a task that discards unneeded data from user-defined pipes.

DISCARD *(<in_pipe> [, <in_pipe> , <in_pipe>, ...])*

Parameters

<in_pipe>
User-defined pipes for which contents are to be discarded.
`PIPE` of any data type

Description

The **DISCARD** command provides a simple way to dispose of unneeded data from certain user-defined data pipes specified by the list of *<in_pipe>* task parameters. For example, the **HIGH** processing command reports both a highest value statistic and an optional output index for locating that term. For special situations where you need *only* the index terms, you still need to do something with the stream of high-value terms, or eventually that pipe will fill to capacity and block the **HIGH** task from performing any further processing.

You do not need to use the **DISCARD** command to remove unused data from input channels. For example, you might configure your xDAP unit to measure 8 channels simultaneously, but then choose to process only three of those channels, `IPipe0` through `IPipe2`, from the input buffers. The remaining channels, `IPipe3` to `IPipe7`, can be disregarded, and their data buffers are recycled automatically.

The situation is different for user-defined pipes. When a task you define places data into a pipe that you define, the DAPL system will not discard the data until the last reader task has processed it. This presents a problem when there is no reader task to trigger this automatic cleanup. The **DISCARD** command substitutes for normal processing, informing the DAPL system that the data are used and can be removed safely from the transfer pipe.

The **DISCARD** command looks for new data in the first input pipe in your parameter list to determine when to run. If you have multiple pipes in your parameter lists, and the data arrive at different rates, put the channel that receives data at the highest rate at the front of the list. This will give the command the best opportunity to perform its cleanup operations. If data arrivals are completely unpredictable, you can use a separate **DISCARD** command for each pipe.

Example

```
DISCARD(pLowest)
```

The **LOW** processing command produced a pipe `pLowest` containing the lowest-valued data values from a stream, and you do not need these values, but you do not want data to accumulate and cause problems. Use the **DISCARD** command to dispose of the undesired data.

```
DISCARD(pImag1, pImag2, pImag3, pImag4)
```

You perform some advanced processing with four **FFT** commands in complex variables, but at the finish the **FFT** commands leave imaginary-valued terms that have no useful information. You dispose of the meaningless numbers from the pipes `pImag1, pImag2, pImag3`, and `pImag4` using the **DISCARD** command.

See Also

FIRFILTER

Define a task that applies configurable FIR digital filtering with decimation.

FIRFILTER *(<in_pipe>, [<channels>,] <vect>, <length>, <scale>, <decim>, <align>, <out_pipe>)*

Parameters

<in_pipe>
The pipe from which data values are read.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<channels>
An optional specification for the number of multiplexed data channels from the input data pipe.
`WORD CONSTANT`

<vect>
A vector of coefficients defining the filter characteristic.
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<length>
The number of terms in the coefficient vector.
`WORD CONSTANT`

<scale>
A final divisor for scaling the filter result after all other calculations.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<decim>
A number that selects the decimation factor to apply for reducing the data rate.
`WORD CONSTANT`

<align>
The number of additional padding terms added to the input data, for phase alignment.
`WORD CONSTANT`

<out_pipe>
The pipe to which filtered data values are written.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

The **FIRFILTER** command provides completely configurable digital filtering with optional decimation, supporting filter designs prepared by other software packages such as *Development Studio for DAPL* (*DAPstudio*). The data stream *<in_pipe>* provides the input data to filter, and the corresponding filtered data are placed in the output stream *<out_pipe>*. The input stream can contain data from multiple, multiplexed data channels. If data for more than one channel are present, the *<channels>* parameter must specify the number of channels. If no *<channels>* parameter is specified, the *<channels>* parameter value defaults to 1, and it is assumed that the *<in_pipe>* data stream represents data from a single sampled data source.

The **FIRFILTER** command implements *finite impulse response filtering*, where the filter is represented by a kernel vector *<vect>* to be convolved with an input stream to produce the filtered output stream. When the filter kernel has a symmetry property, the filter will have the property of exactly constant group delay, which means zero phase distortion, superior to one-sided approximations such as Bessel filters. Any data type can be used for the filter vector coefficients. The number of coefficient terms in the filter kernel is specified by the *<length>* parameter. The *<length>* value must be in the range 1 to 1200 terms (typically 30 to 90). If the value 0 is specified for the length,

the **FIRLFILTER** command will count the number of coefficients present in the vector and use that count for its processing. However, it is helpful to explicitly specify the length, so the DAPL system can warn you if there is any inconsistency between the number of terms expected from the filter design and the number of terms that are actually present in the script.

Filter design programs such as DAPstudio can represent filter coefficients with integer values, on the assumption that the processor will perform calculations directly in a fixed point data type. However, this presents a problem, since coefficient values are most often too small for integers to represent them adequately. To avoid accuracy loss, the filter coefficients can be scaled by a relatively large number that is canceled out later when the filter is evaluated. The **FIRLFILTER** command assumes the following implicit scale factors for its coefficient data types:

- `WORD` `32768`
- `LONG` `2147483648`
- `FLOAT` `1.0`
- `DOUBLE` `1.0`

You can think of this as: "*the true coefficient values are the values listed in the vector divided by the scaling factor.*"

However, despite this scaling, the rounding of coefficients to integer quantities can still impair the accuracy of filter results. Some of this loss is avoided by allowing coefficients to be larger by an additional factor, and then dividing out this additional factor in a manner similar to the implicit scaling for the data type. Filter designs using the DAPstudio filter designer will report the value of this extra scale factor, which can then be entered as the value of the <*scaling*> parameter. Mathematically, the scaling factor is just a scalar value, so there is nothing to preclude using this divisor for other purposes.

It is possible for the filtering to produce values that are locally larger than they were in the original signal. This can occur when the filtering removes frequencies that previously caused peaks of a waveform to flatten. Though the results are correct, they can become a problem when new peak values in filtered data exceed the range that the output data type can represent. Range limitations are relatively commonplace when working with 16-bit data coming directly from digital sampling. To avoid severe, artificial sign-reversal discontinuities, the results of `WORD` data type filtering are bounded to the nearest representable value in the range -32768 to +32767. Other data types have enough bits of precision that numerical range limits are easily avoided, so *other data types are given no range error protection*. When using the DAPstudio filter designer, it will warn you of the largest possible increase in the peak values, and, if you wish, you can adjust the <*scaling*> parameter to remove any theoretical exposure to range errors.

A common application of filtering is removing extraneous high frequency noise from a data stream. A relatively high sampling rate is often used, so that high frequencies are correctly represented in the data stream, and harmful *aliasing side effects* from sampling are avoided. After filtering removes the high frequencies from the data stream, there is no reason to retain the initial high sample rate. Specifying an integer value greater than 1 for the <*decim*> parameter tells the **FIRLFILTER** command to include a post-filtering decimation operation. If you specify 0 (no decimation) or 1 (the same thing, 1 sample out per 1 sample in), no decimation is applied. If you specify a larger integer value `N`, then from each group of `N` filtered output samples, one value is retained, and then the next `N-1` values are dropped from the output stream. The **FIRLFILTER** command does this in a relatively intelligent way, avoiding calculations that do not contribute anything to retained output values.

The **FIRLFILTER** command calculations are very basic "multiply and add." Before this process can begin, there must be one value from the input data stream for each of the vector coefficients. Consequently, the first `N` input stream values yield one output value. After that, each new input value yields one new output value (before decimation is applied). When the input and output streams are transferred to the host system, the numbers of terms are different for the two data streams, and they *do not line up*. The **FIRLFILTER** command provides an <*align*> parameter that can be used to adjust this behavior.

- If the value of <*align*> is 0, there is no adjustment. This is the simplest choice when alignment and signal phase are not important. All of the filtered results will be completely valid, based entirely upon input data, but the first filtered output value will not appear until <*length*> input terms are processed.
- The largest allowed <*align*> parameter value is <*length*>*-1*. If this value is specified, <*length*>*-1* artificial input initial data values of 0 are supplied, so that the first real value from the input stream completes a set of <*length*> values needed for the first calculation. This will result in a data stream with the same number of output values as input values, but because of the artificial injected values, the first <*length*>*-1* output values will show artificial transient effects.
- If the filter is *symmetric* (all designs produced by DAPstudio filter designer are symmetric), and the value specified for <*align*> is equal to <*length*>/2, then artificial 0 values are inserted for <*length*>/2 input terms. There will be <*length*>/2 output terms affected by an artificial initial transient, but after that, the input and output data streams will be *aligned in time,* exactly zero relative phase shift at any frequency.
- Since the preceding <*length*>/2 special case is such an important one, a special shorthand notation is provided for it. Specifying the artificial value -1 for the <*align*> parameter means "perform time alignment," and the <*length*>/2 value is calculated and applied automatically.
- General non-symmetric FIR filters can have various alignment properties. The alignment might not be to an exact integer position. The <*align*> parameter can be adjusted to the integer value that produces the closest phase alignment for purposes of the application.

Portability notes

This command is very similar to the **FIRLFILTER** command from the *DAPL 2000* system. Most applications will find that a configuration that worked under *DAPL 2000* will work unchanged, but there are some differences.

1. The *DAPL 3000* version does not support the block-mode skip parameters. If you need blockwise data selection, you must do that using a separate **SKIP** command.
2. Filter computations in *DAPL 3000* use higher internal precision, and this could sometimes result in integer rounding to a different final decimal digit for occasional individual output values.
3. *DAPL 2000* allowed longer filter kernels. This rarely makes any difference, since FFT-based filtering methods are significantly more efficient for extremely large filter kernels, and should be used instead of the **FIRLFILTER** command.
4. The *DAPL 3000* version can apply the same filtering to every channel in a multiplexed stream, but the *DAPL 2000* version supports single-channel filter calculations only.
5. The *DAPL 3000* version allows any data type for the scaling factor, which is not restricted to an integer value or a power of 2 as in the *DAPL 2000* system.
6. The *DAPL 2000* version requires matching data types for the input stream, kernel vector coefficients, and output stream. The *DAPL 3000* versions allows these data types to be selected independently. Most importantly, calculations are automatically selected to preserve appropriate precision for the output data pipe type, and results are automatically converted to that output data type.

Examples

```
FIRFILTER (IPipe3, FVEC, 0, 0, 0, -1, $BinOut)
```

Apply a lowpass filter using default options. The number of channels is not specified explicitly, since a single channel of sampled data is provided by input channel pipe `Ipipe3.` The filter characteristic `FVEC` is a vector previously calculated by the DAPstudio filter designer, with no scale factor. The length field is set to 0, which means that the number of coefficients is determined from `FVEC`. Since no extra scaling is required, the scaling parameter is set to 0 (equivalent to scaling factor 1). There is no decimation, so the decimation parameter is set to 0 (equivalent to decimation factor 1). The special -1 alignment option is specified, so that data streams are time-aligned for plotting on the same graph. The filter results are WORD data that can be sent directly to the host.

```
FIRFILTER (Ipipe(0..7), 8, FVEC61, 61, 2, 4, 0, pFilt)
```

Apply a lowpass filter to each of the 8 channels in the set of multiplexed data channels received directly from the input channel pipe, using adjusted options. The `FVEC61` vector of filter coefficients is supposed to have 61 terms; if there is any other number of terms, a scripting error has occurred. The filter design uses coefficients that have an extra scaling factor of 2, so specify this value to be canceled out of the results. This is a lowpass filter, and, after the filter calculations, the data rate is to be automatically decimated by a factor of 4. No artificial samples are injected into the input data stream for alignment, hence there are no artificial transients, but the first evaluations will not occur until the first 61 input values have been loaded from each channel. The filtered results for the 8 channels are placed in multiplexed fashion into the specified pipe `pFilt,` in the data type of that pipe.

See Also

FIRLOWPASS

FIRLOWPASS

Define a command that automatically applies lowpass digital FIR filtering with decimation.

FIRLOWPASS *(<in_pipe>, [<channels>,] <decim>, <out_pipe>)*

Parameters

<in_pipe>
The pipe from which data values are read.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<channels>
An optional specification for the number of multiplexed data channels from the input data pipe.
`WORD CONSTANT`

<decim>
A number that selects the decimation factor to apply, and therefore indirectly determines the filter characteristic.
`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 **FIRLFILTER** command, using predefined filter characteristics matched to the decimation factor *<decim>* . The filtering is applied to the sampled signal stream from *<in_pipe>*, and the filtered data stream is placed into *<out_pipe>*. The input data stream can be a set of multiplexed input signal channels, in which case the *<channels>* parameter must specify the number of signal channels present. If no *<channels>* parameter is specified, the *<channels>* parameter value defaults to 1, and it is assumed that the data stream represents a single sampled data channel. The same filtering characteristic is applied to each channel in the set. 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.

The **FIRLOWPASS** command is useful for filtering that must eliminate high frequency noise, while exactly preserving low frequency information. It is simple to use, and does not introduce any aliasing artifacts, so it is a safe way to reduce data rates and obtain a more efficient signal representation. It is suitable for anti-aliasing applications, but the filtering is generally more aggressive than is strictly necessary for anti-aliasing alone.

The decimation factor *<decim>* allows a **FIRLOWPASS** task to perform efficient data rate reductions. The value of decimation factor *<decim>* must be in the range 1 through 12. If the value is 1, filtered values are produced at the same rate that new input values are received. For other values of decimation factor *<decim>,* only one output sample is produced for each group of *<decim>* input samples received, reducing the data rate. This allows more efficient representations of the signal when the original sampling rate is much higher than necessary to completely represent the required information. This kind of situation occurs frequently when signals are deliberately *oversampled* at a high rate to avoid aliasing contamination, but the bandwidth of useful information is only a small fraction of the bandwidth covered by the high-rate data stream. The **FIRLOWPASS** filter characteristics are specially designed so that the decimation operation produces *no aliasing side effects*.

Note: *There is nothing that the* **FIRLOWPASS** *command can do if your data set already contains aliasing produced at the time sampling was done at the original rate.*

The gain accuracy (flatness) of the filtering depends on the data type.

- `WORD` data stream: gain accuracy 14 bits.
- `LONG`, `FLOAT`, or `DOUBLE` data stream: gain accuracy 18 bits.

For example, 14-bit accuracy means that a digitized signal spanning the full representable range will have an attenuation error of no more than 4, compared to the full range of 32767. This corresponds to a passband gain error of approximately ±0.001 dB. Stopband output levels can similarly be off by 4 counts, which corresponds to a noise floor -78.3 dB below the passband level. These errors are typically too be small to be distinguished from signal noise.

The predefined filter characteristics provide the following properties:

- The filter gain is 1.0 through the first 50% of the new Nyquist frequency after decimation.
- The frequencies from 75% of the new Nyquist frequency to the original Nyquist frequency are eliminated prior to the decimation.

Because of the symmetry property of the FIR filtering characteristic, the 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.

Example

```
FIRLOWPASS (IPipe3, 10, P)
```

Apply a lowpass filter to the data in input channel pipe `IPipe3,` and decimate by placing every tenth result into pipe `P,` so that the data rate is safely reduced by a factor of 10 without introducing aliasing into the sampled data.

```
FIRLOWPASS (IPipe(0..7), 8, 5, P)
```

Apply lowpass filtering to each channel in the set of 8 multiplexed data channels, from input channel pipes 0 through 7. Decimate all 8 of these signals using the same lowpass characteristic, and then safely decimate by a factor of 5, so that the resulting data stream delivers 1/5 the original number of samples. Place the reduced data streams into pipe `P`.

See Also

FIRFILTER

MIXRFFT

Define a task that calculates fast Fourier transforms with flexible data-block sizes.

MIXRFFT *(<N>, [<direction>,] [<window>, | <windtype>[<alpha>],]*
<pipeinR>, [<pipeinI>,] [<blocks> ,] <post> , <pipeoutR> [, <pipeoutI>])

Parameters

<N>
The number of terms in each data block to process.
`WORD CONSTANT | LONG CONSTANT`

<direction>
An optional keyword parameter specifying the transform direction.
`FORWARD | REVERSE`

<window>
An optional vector specifying a custom window.
`WORD VECTOR | LONG VECTOR | FLOAT VECTOR | DOUBLE VECTOR`

<windtype>
An optional keyword specifying a predefined window type.
`RECTANGULAR | BARTLETT | VONHANN | HAMMING | BLACKMAN | KAISER`

<alpha>
A selectivity parameter used with a Kaiser window type.
`FLOAT CONSTANT | DOUBLE CONSTANT`

<blocks>
An optional keyword specifying block size reduction.
`FULL | HALF`

<post>
A keyword specifying the kind of post-processing to apply.
`PARTS | POWER | MAGNITUDE | POLAR`

<pipeinR>, <pipeinI>
Pipes for blocks of input data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<pipeoutR>,<pipeoutI>
Pipes for blocks of output data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

A **MIXRFFT** task calculates a Discrete Fourier Transform of data blocks using a mixed-radix Fast Fourier Transform algorithm, hence the name: **MIX**ed **R**adix **F**ast **F**ourier **T**ransform. Data are taken in blocks of length

<*N*> from pipes <*pipeinR*> and optionally <*pipeinI*> for complex-valued data. If the data are real-valued, omit the <*pipeinI*> parameter. Calculations are performed in three steps:

1. preprocessing is applied,
2. the transform is calculated,
3. postprocessing is applied.

Processing is configured by optional keywords, represented as reserved *keyword* names with characters in upper case, and without quotation marks. Results are written to <*pipeoutR*> and <*pipeoutI*>. Some processing options produce only one output stream, and for these omit <*pipeoutI*>.

When applied to data blocks with restricted lengths that are an exact power of two, the **FFT** command produces equivalent results with slightly more efficiency. The **MIXRFFT** command, however, is not restricted to blocks that are exact powers of 2, nor is it restricted to blocks of length 16384 or less. The speed difference is only a few percent. Advantages that result from "fine tuning" the block size are often more significant; for example, it is significantly more efficient to calculate a 5000 FFT terms using the **MIXRFFT** command, rather than rounding <*N*> up to 8192 and applying the **FFT** radix-2 command.

Block Length

There are some restrictions on the block length parameter <*N*>, which specifies the number of terms in the input data blocks. The block length must be less than 2^{24} (16777216). The block size must be exactly multiple of prime factors 2, 3, 5, and a "few more prime number factors" of size 19 or less.

For example: the following block sizes are acceptable because they have an acceptable factorization.

1024	2^{10}
1000	$2^3 * 5^3$
1020	$2^2 * 3^1 * 5^1 * 17^1$
200000	$2^6 * 5^5$

The following does not have an acceptable factorization, so it cannot be used as the block length.

860	$2^2 * 5^1 * 43^1$

Direction

You can specify the transform direction using the <*direction*> keyword parameter. It must be one of the following.

FORWARD	from time-sequence to frequency spectrum
REVERSE	from frequency spectrum to time sequence

This selection affects the signs of the phase terms and the scaling. Like the **FFT** command, the 1/N scaling factor of a forward-reverse transform pair is applied to the forward direction transform. If you omit the <*direction*> parameter, you will get a FORWARD transform.

Window Preprocessing

Windowing can be applied to the data set during preprocessing. If you provide a window vector, the terms of the window are applied in sequence. If you specify one of the pre-determined window types, the window is calculated and applied, term-by-term, to each term of the input data. The windowing is less efficient than the pre-computed window processing of the **FFT** command, but avoids allocating very large blocks of extra storage. Unlike the **FFT** command, the windows produced by predefined window options can be used with both forward and reverse transform directions. In the forward direction, data significance is best preserved at the center of the block, attenuated at the block ends. In the reverse direction, frequency significance is best preserved at low frequencies, attenuated at the Nyquist frequency of the sampling.

The Kaiser window is the only window type with a selectivity parameter, and the only window type that has no closed-form expression for the window values. The selectivity parameter has no default, and it must be a positive number less than 12.0. High values produce smoother results, with better isolation of widely separated frequency bands, but poorer resolution of nearby frequencies so that spectrum peaks are rounded. Kaiser window calculations use an approximation formula that is accurate to approximately 8 decimal digits.

If you omit both windowing options, the RECTANGULAR window is the default. This is the same thing as making no changes to the input data before the transform.

Postprocessing

The <*post*> keyword parameter must be specified to select the kind of post-processing to apply. Raw FFT results are complex numbers. Often, it is helpful to convert these to alternative forms for analysis. The <*post*> option must be one of the following:

PARTS	return the raw complex terms unmodified
POWER	convert the results to power spectral density
MAGNITUDE	convert the results to magnitude (square root of power)
POLAR	convert the results to polar form, magnitude and phase

See the FFT chapter in the DAPL manual for a discussion of how phase angles for the POLAR option are represented in each data type.

The PARTS and POLAR post-processing options produce two output data streams, <*pipeoutR*> and <*pipeoutI*>. For PARTS post-processing, the two output streams must have the same data type as each other.

The POWER and MAGNITUDE post-processing options produce only one output data stream, <*pipeoutR*>. For these options, omit <*pipeoutI*> from the parameter list.

The POWER, MAGNITUDE, and POLAR postprocessing calculations behave differently for the case when the second half of the spectrum is discarded (see the next section) and pipe <*pipeinI*> is omitted. When there are no imaginary terms, frequencies above and below the Nyquist frequency at location <N>/2 in the data block cannot be distinguished in sampled data. Assuming that the sampling was valid, the transform will show above the Nyquist frequency an artificial mirror image of the frequencies shown below the Nyquist frequency. So, during post-processing for these configurations, the effects of the two halves are recombined before discarding the second half-block.

Note: Postprocessing checks whether imaginary input data are provided, but does not check values. A stream of zero-valued imaginary terms can produce different results than omitting the imaginary terms, even though the transform operation is the same.

Block-Size Reduction

You can use the <*blocks*> option to specify that extraneous terms in the upper half of the spectrum should not be sent to the output data pipes. This parameter must be one of the following.

FULL	Do not discard any terms.
HALF	Discard the second half of the transform block.

This is useful primarily for the case of real-valued input data, when the second half-block is symmetric and contains no new information. If the <*blocks*> parameter is not specified, the default will depend on the kind of input data.

- If you provide imaginary-valued input terms, the default will be FULL.

- If you do not provide any imaginary-valued input terms, the default will be HALF.

Data Streams

Any numeric data type can be used for input data. The input data blocks are provided via pipes <*pipeinR*> and <*pipeinI*>, and must be of the same data type. If you omit the <*pipeinI*> (data are real-valued), zero values are assumed for the imaginary input parts during the transform. When <*pipeinI*> is omitted, typically:

- The data are from time-domain sampling, and the transform direction is FORWARD.
- Due to the symmetry properties, the second half of the spectrum has no useful new information.

Using an output data type with more precision is sometimes helpful for preserving range. For example, using WORD-type input data, the terms in a very long forward FFT tend to become very small and lose accuracy to integer truncation. You can avoid this problem by using a FLOAT output type. Similarly, using WORD-type input data in a very long reverse transform tends to produce results that are poorly scaled, with many terms saturated to the range limits. You can avoid this problem by using a LONG output data type.

Unlike many other DAPL system commands, the data types of input and output data streams are not required to match. The output data streams depend on the postprocessing option, and the postprocessing can produce any data type. Output data types are not restricted to being the same as the input data type. That means, for example, you do not need to convert input samples to FLOAT type to get an output spectrum in FLOAT type.

Transform Mathematics

The algorithm of the mixed-radix FFT is based on the Singleton FFT, developed by R. C. Singleton at Stanford Research Institute in 1968, and published by IEEE. The brilliance of the mathematics was inversely proportional to the clarity of the coding. Major restructuring was applied to make the code reentrant and friendly to modern processors, so it could run effectively in the DAPL environment. Transforms are computed in place, using one small supplementary buffer as an internal scratchpad. No pre-computed table of "twiddle factors" is used. A pre-computed permutation table, adjusted for each specified block size, is used for the final shuffling operations.

Examples

```
MIXRFFT ( 1000, P1r, PARTS, P2r, P2i )
```

Read blocks of 1000 sampled-data values of WORD data type from pipe P1r. The signal is real-valued, so the imaginary parts are zero and are omitted. By default, perform a forward transform of each input block with no windowing, and place the complex results in pipes P2r and P2i. Perform no other post-processing calculations. Because the second half of the spectrum will have no new information, by default only the first 500 terms of the transform result are delivered to the output data pipes.

```
MIXRFFT ( 256000, FORWARD, HAMMING, DPr, DPi, \
        FULL, MAGNITUDE, Pmag )
```

Read blocks of 256000 samples from input data pipes DPr and DPi, which provide real and imaginary parts obtained from quadrature demodulation. The data types are double on all pipes to preserve as much precision as possible. Apply a Hamming window to the data sets to reduce block truncation effects. After computing the forward transform, convert the real and imaginary transformed parts to magnitude values. The high and low frequency terms are not combined because the input stream is complex valued. A full block of 256000 spectrum magnitude values for each 256000 complex input terms is sent to the Pmag pipe.

See Also

FFT

Also refer to the chapter “Fast Fourier Transform” in the DAPL 3000 Manual for more information about frequency spectra, sampling, aliasing effects, data representation, and window operators.

MRBLOCK

Define a task that delivers the newest complete data block on demand.

MRBLOCK *(<pin>, <request>, <bsize>, <pout>)*

Parameters

<pin>
Input stream consisting of equal-size blocks of data.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<request>
The location where requests for data transfer are posted.
`WORD PIPE | WORD VARIABLE`

<bsize>
The number of data values in each block.
`WORD CONSTANT | LONG CONSTANT`

<pout>
Output stream receiving the requested data blocks.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

A **MRBLOCK** task temporarily buffers complete data blocks in memory. Data arrive in the *<pin>* pipe. Requests to deliver the most recent block of this data arrive via the source specified by the *<request>* parameter. Data are processed in blocks of a size specified by the *<bsize>* parameter. The request specifies the number of data blocks to provide, starting with the most recent one preserved in memory. Most of the time, the request is for one block, and the response occurs immediately. The most current data block is delivered immediately to the *<pout>* output stream. While there are no requests, the output stream receives no data. Any data blocks not requested are discarded.

Processes send requests for data via the *<request>* variable or pipe, specifying the number of blocks to deliver. Any requests for less than 1 block are removed and ignored. When the *<request>* parameter is a variable, the request value is set back to zero after processing. When the *<request>* parameter is a pipe, requests are processed one at a time, so the total number of data blocks delivered equals the sum of the valid request values. Though the first block comes out of memory with minimum delay, the subsequent blocks come directly from the source stream. At slow sampling rates, this could mean some delay to collect and deliver all of the required data

Here are some common applications for this command.

1. Reducing data skew. An application samples at high rate so that a complete set of sensor readings is available at any time. This produces many more sample blocks than the external process requires. An **MRBLOCK** task delivers one block of readings covering all channels when the requesting process is ready and sends a request.

2. GUI displays. While graphical “live” displays on a PC monitor seem fast on a human scale, they are very slow compared to high-speed data acquisition rates. Using an **MRBLOCK** task, the processing on a PC host can accept new data at its own pace, whenever the need arises.

3. Status summary reports. Some processes on a PC host are driven by requests from external data collection activity, typically through a network. An **MRBLOCK** task allows the host application to pass these requests through to the processing on the DAP, which returns the most recent status summary data.

Examples

```
MRBLOCK(pFFT,Cp2In,10240,Cp2Out)
```

Read data blocks from data pipe `pFFT`, saving the most recent block in memory. Each block of 10240 values contains 1024 values for each of 10 channels, resulting from an FFT analysis. Ordinarily, waiting for this analysis to complete on request would cause unacceptable delays. When the application requests one of these blocks for display, the application passes the value 1 to the **MRBLOCK** task via the communication channel `Cp2In`. Each time a request arrives, the results of the most recent completed analysis are returned via the `Cp2Out` communication pipe.

See Also

MTSFILT

Define a task that aligns input samples to the same instants of time.

MTSFILT *(<pinmux>, <nchan>, [<ngroup>,] <ndecim>, <poutsync>)*

Parameters

<pinmux>
Input stream of multiplexed data, as captured by sampling multiple channels.
`WORD PIPE`

<nchan>
The number of channels included in the input data stream.
`WORD CONSTANT`

<ngroup>
The number of channels sampled simultaneously by input hardware.
`WORD CONSTANT`

<ndecim>
Decimation to apply to the data stream after processing.
`WORD CONSTANT`

<poutsync>
Output stream of multiplexed data, time-aligned and decimated.
`WORD PIPE`

Description

A **MTSFILT** task uses DSP interpolation methods to perform a time shift correction on multiple channels. It is useful when the channels are sampled using multiplexing hardware, which produces samples at a sequence of positions in time, but samples for all channels are desired at the same position in time. Time-alignment of samples greatly simplifies certain kinds of phase analysis. A definitive solution is to use specialized hardware devices with signal capture features to latch every channel independently and simultaneously. The command is intended as a lower-cost alternative without the special hardware.

A **MTSFILT** task receives multiplexed input data from the *<pinmux>* pipe. The total number of multiplexed channels, common to the input and output streams, is specified by the *<nchan>* parameter. The number of channels sampled simultaneously by the hardware is specified by the *<ngroup>* parameter. The value of *<nchan>* must be an exact integer multiple of *<ngroup>*. If omitted, the default value of the *<ngroup>* parameter is 1, meaning that only one hardware converter was used to produce the data stream, valued captured sequentially.

A nontrivial DSP resampling analysis reconstructs every signal, and then evaluates the signals in such manner that time displacements due to the multiplexer sampling time intervals are corrected, without distorting the signals. The samples are aligned to the time of the last channel in the input channel list. The output stream values are placed into the *<poutsynch>* pipe. After the time-shift correction, the resulting sample values are indistinguishable from samples as they would be obtained from true simultaneous sampling by hardware on all channels

There is one important restriction. *Frequencies present on all of the input channels must be suitably band-limited.* To preserve accuracy, signals must all be at least 2X oversampled; that is, the highest frequency present in the input signal must not exceed 1/4 of the sampling frequency. Resampling is closely related to interpolation by fitting a curve through the data sequence and using locations on the curve to establish values at intermediate

positions. It makes sense that the best results will occur when there are more points available for fitting the curve, so that the curve is smoother. The bandlimit restriction forces a degree of smoothness on the input data set.

If there is a sharp discontinuity in the input data, such as a step disturbance, this local violation of the bandlimit restriction above will result in an artificial transient disturbance in the data set. This is not necessarily harmful, as long as you recognize this as a side effect of the processing and not something actually present in the physical signal.

One way to obtain an acceptable signal meeting the bandwidth restrictions is to attenuate high frequencies with a simple filter in hardware, then sample at a higher rate than necessary. For example, if sampling one value every 100 microseconds is sufficient, sampling at 25 microsecond intervals would provide an additional factor 4 oversampling. Because this produces 4 times more data than you actually need, the <*ndecim*> parameter can be set to 4. After the time-shift corrections are applied, extra samples from the oversampling are discarded automatically. Thus, with <*ndecim*> equal to 4, you will get 1 output sample for every 4 samples captured by the hardware.

The time shift correction filters requires a few samples in the local neighborhood of each location where it performs its calculations. A few samples must be obtained from each channel before output results start to appear. If you use an **MTSFILT** command in your input sampling procedure, you must allow extra samples to supply these initializer samples for the reconstruction filters.

Examples

```
MTSFILT(IP(0..3),4,5,PSYNC)
```

Take sampled data from an input channel pipe with 4 channels, captured by hardware having a single converter, with samples captured at equal intervals of time. It is desired to record one sample in each channel every millisecond. To guarantee that there are plenty of data for the time-interpolation, the signals are filtered in hardware before sampling, and the initial sampling interval is 200 microseconds between samples in each channel. At the boosted sampling rate, 5 samples are collected for each 1 sample needed in the output stream, so 5 is specified as the decimation parameter. The **MTSFILT** command places the time-aligned and decimated data into the output pipe `PSYNC`. The results are as if all four channels were simultaneously sampled at 1000 microsecond intervals.

```
MTSFILT(IP(0..31),32,8,5,PSYNC)
```

Process 32 signal channels, sampled with a Data Acquisition Processor model that has 8 hardware converter channels that operate simultaneously. Because there are 32 channels but only 8 converters, the hardware alone cannot sample all of the channels simultaneously. The configuration is like 8 of the configurations shown in the previous example, running in parallel. A sample value is desired for each of the 32 input channels each millisecond. The sampling groups of 8 channels each are captured at 200 microsecond intervals. The **MTSFILT** command takes the groupings of simultaneously sampled channels into account when applying the time shift corrections. The results are as if all 32 channels were simultaneously sampled at 1000 microsecond intervals.

See Also

SAWTOOTH

Define a task that generates sawtooth wave data.

SAWTOOTH *(<pk_min>, <pk_max>, <wave_freq>, [<init_phs>,] <samp_time>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

SAWTOOTH *(<amplitude>, <period>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

Parameters

The parameters for a **SAWTOOTH** command are shared by several waveform commands. The common parameter description is provided in the **WAVEFORMS** command description page.

Description

A **SAWTOOTH** command defines a task that generates sawtooth wave data, and places the data in <*out_pipe*>. The waveform is obtained by taking a continuous-time version of the waveform and selecting values along that path, according to the timing parameter <*period*>, or the combination <*wave_freq*> and <*samp_time*>.

A **SAWTOOTH** waveform has an unusual special property that it is discontinuous. If the sample position happens to align exactly with the discontinuity, the value of the waveform is undefined. In such cases, the value that is halfway between the two extrema is substituted.

In cases where the waveform is balanced in the positive and negative directions, (as will always be the case when using the traditional parameter form) the first output value is 0, which occurs halfway through the upward trajectory of the waveform. The output values continue on an upward trajectory at a constant rate until reaching or passing the point of discontinuity, halfway through the cycle. At this point, the waveform jumps down to or very near the lower extreme value. Subsequent values continue the upward trajectory at the original constant rate to reach the end of the cycle. This behavior is different than it was in the DAPL 2000 System, where the initial evaluation point was at the discontinuity point.

For generating a ramp sequence that is easier to configure to reach certain output levels specifically, consider the **BIRAMP** command as an alternative.

Examples

```
SAWTOOTH(1000,100,P2)
```

Generate a balanced sawtooth wave signal with nominal range -1000 to 1000, and a period of 100 sample intervals, in 16-bit format suitable for delivery to D-to-A output hardware. Place the waveform data into pipe P2.

```
SAWTOOTH(0.0, 10000.0, 60.0, 50.0, PWAVE,"AMPLITUDE", PAMOD)
```

Generate samples of a sawtooth wave having a fundamental frequency 60 Hz when the output values are updated at a 50 microsecond per sample rate. The amplitude peaks range from 0.0 to 10000.0 before modulation (with all modulation factors equal to 1.0). The data are amplitude modulated by values from the `PAMOD` pipe. The resulting modulated waveform data in floating point format are placed into pipe `PWAVE`.

See Also

COSINEWAVE, SINEWAVE, TRIANGLE, SQUAREWAVE , BIRAMP

SCAN (for input definitions)

Define the scan interval for capturing one value per sampled-input channel.

SCAN <*interval*>

Parameters

<*interval*>

The time interval during which every sampled input channel receives a sample.
`explicit number, in an integer or floating point notation`

Description

The **SCAN** command specifies the time interval to capture one sampled input value for each channel defined in an input sampling configuration. This is the recommended way to specify the timing for input sampling. The DAPL 3000 system will determine how to schedule the low-level sampling sequences.

The specified time interval <*interval*> for completing the scan must be within minimum and maximum limits specific to the xDAP model. The time interval is expressed in units of microseconds, as an integer value or as a decimal fractional value with resolution to 0.001 microseconds. If the specified value is not an exact integer multiple of the sampling timer resolution, the DAPL 3000 system will issue a warning message and round the <*interval*> value down to the nearest allowable multiple. The DAPL 3000 system will diagnose an error if the xDAP hardware is unable to perform all of the necessary sampling operations fast enough to meet the requirements of the **SCAN** command.

The **SCAN** command must appear after all of the **SET** commands that define signal channels for the input sampling configuration. If the **SCAN** command is omitted, a **TIME** command must be specified; for this case, the DAPL 3000 system will internally construct an equivalent **SCAN** command interval equal to the number of channels times the time interval specified on the **TIME** command.

For the following example configuration, suppose that the xDAP model connects pins `d0` and `d1` to channel selector 0 and pins `d14` and `d15` to channel selector 7.

```
IDEFINE uses2
 CHANNELS 5
 SET ipipe0 d0
 SET ipipe1 d1
 SET ipipe2 d14
 SET ipipe3 d1
 SET ipipe4 d15
 SCAN 10.0
END
```

The number of sampling events required to obtain a sample for every channel in the channel list depends on the selected channels and the channel selector architecture of the xDAP model. For the example above, channel selector 0 is used three times within the scan interval, for capturing samples from pins `d0, d1,` and `d1` in that sequence. Channel selector 7 is used only two times, capturing pins `d14` and `d15`.

By default, the DAPL system will attempt to distribute the sampling events through the channel scan interval. This does not in general imply exactly equal time intervals, particularly for operations on an individual channel selector. In the example above, there is no activity for channel selector 7 while the third sampling event captures pin `d1` using channel selector 0. The distribution of sampling events can be adjusted using an additional **TIME** command following the **SCAN** command.

The sizes of individual sampling intervals are automatically adjusted so that each one is consistent with the timing resolution of the sampling clock. For the example given previously, perfectly-spaced sampling events would need to occur at precise intervals of `3.333333333` microseconds, which the sampling clock can't match exactly. The intervals between sampling events are adjusted in a way that preserves the total <*interval*> length specified by the **SCAN** command. The DAPL system might need to make similar small adjustments to coordinate different kinds of input devices.

Examples

```
IDEFINE   capture12
  CHANNELS  12
  SET  ipipe0  D0
  . . .
  SET  ipipe11  D11
  SCAN  35
END
```

Configure input sampling to capture data for 12 differential input channels during each time interval of 35 microseconds. The DAPL system will adjust for the fact that 12 samples per 35 microseconds is not a perfect match for the time resolution of the sampling clock.

See Also

SINEWAVE

Define a task that generates sine wave data.

SINEWAVE *(<pk_min>, <pk_max>, <wave_freq>, [<init_phs>,] <samp_time>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

SINEWAVE *(<amplitude>, <period>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

Parameters

The parameters for a **SINEWAVE** command are shared by several waveform commands. The common parameter description is provided in the **WAVEFORMS** command description page.

Description

A **SINEWAVE** command defines a task that generates sine function data, and places the data in *<out_pipe>*. The sequence is obtained starting from a continuous-time version of the waveform, scaled and phase shifted according to the waveform parameters. Locations of samples are selected along that path, according to the timing parameter *<period>,* or the combination *<wave_freq>* and *<samp_time>*. For a balanced waveform, the output values start at an initial value of zero, which is the sine function value corresponding to phase angle zero.

Note: A **SINEWAVE** command configured with an initial phase angle 0.5 (implicitly, 0.5 π) is equivalent to a **COSINEWAVE** command.

Note: A **SINEWAVE** command configured with *<pk_min>* and *<pk_max>* parameters that are not balanced produces the equivalent of a balanced sine wave combined with a DC offset level equaling *(<pk_min>* + *<pk_max>)/2*.

Examples

```
SINEWAVE(1000,100,P2)
```

Generate a balanced sine wave with values ranging from -1000 to 1000, and a period of 100 sample intervals, in 16-bit format. Place the waveform data into pipe P2.

```
SINEWAVE(0.0, 1.0, 60.0, 25.0, PWAVE)
```

Generate samples of a sine wave plus DC offset so that the value is never negative. The frequency is 60.0 Hz when the specified updating interval of 25 microseconds is used. Because of the DC offset level, halfway between the range limits of 0.0 to 1.0, the first output will have the value of 0.5. The resulting waveform data in a floating point format are placed into pipe PWAVE.

See Also

COSINEWAVE, **TRIANGLE**, **SAWTOOTH**, **SQUAREWAVE**

SQUAREWAVE

Define a task that generates square wave data.

SQUAREWAVE *(<pk_min>, <pk_max>, <wave_freq>, [<init_phs>,] <samp_time>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

SQUAREWAVE *(<amplitude>, <period>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

Parameters

The parameters for a **SQUAREWAVE** command are shared by several waveform commands. The common parameter description is provided in the **WAVEFORMS** command description page.

Description

A **SQUAREWAVE** command defines a task that generates square wave data, and places the data in <*out_pipe*>. The sequence is obtained by taking a continuous-time version of the waveform and selecting values along that path, according to the timing parameter <*period*>, or the combination <*wave_freq*> and <*samp_time*>. The output values start at the high level, equal to <*pk_max*>, and, halfway through the cycle, switch over to low output values, equal to <*pk_min*>.

A **SQUAREWAVE** waveform has an unusual property that (without amplitude modulation) the only values it can produce are the two extreme values. If a discontinuity location is reached exactly, the mathematical value of the waveform is undefined there. If this happens, the transition is considered to happen first, and the selected output value will match the values for the points immediately following.

Note: When the period of the waveform spans an odd number of samples, the number of values at the high output level and the number of values at the low output level cannot be equal. This results in a waveform with a small DC imbalance.

Examples

```
SQUAREWAVE(1000,100,P2)
```

Generate a balanced squarewave signal with range -1000 to 1000, and a period of 100 sample intervals, in 16-bit format. Since the period is an even number, the waveform is exactly balanced, alternating 50 high values with 50 low values. Place the waveform data into pipe P2.

```
SQUAREWAVE(0.5, 2.4, 1000.0, 4.0, PWAVE)
```

Generate samples of a square wave that cycles between 0.5 volts, a voltage sufficiently low to serve as TTL logic low, and 2.5, a voltage sufficiently high to serve as TTL logic high. The fundamental frequency of the waveform is 1000 cycles per second when the output values are updated at the specified 4.0 microsecond per sample rate. The resulting waveform data in floating point format are placed into pipe `PWAVE`.

See Also

COSINEWAVE, SINEWAVE, TRIANGLE, SAWTOOTH

TIME (for input definitions)

Specify a time interval between sampling events during a channel list scan.

TIME (*<interval>*)

Parameters

<interval>

The time interval desired between sampling events.
`explicit number, in an integer or floating point notation`

Description

The **TIME** command in an input sampling configuration provides a means for specifying the desired time interval between sampling events. The **TIME** command also has a secondary interpretation for back-compatibility with legacy application configurations.

The specified time interval <*interval*> between sampling events is expressed in units of microseconds, as an integer value or as a decimal fractional value with resolution to 0.001 microseconds. The actual times of sampling events will be adjusted internally to integer multiples of the sampling timer resolution. An error will be diagnosed if the specified <*interval*> value is too fast for the xDAP hardware, or too slow to collect samples from all channels within the time interval allowed by the **SCAN** command.

A sampling configuration that does not specify a **TIME** command has its sampling events distributed as evenly as possible through the scanning interval defined by the **SCAN** command. For some applications, this default behavior might not be desirable. For example, it might be preferable to approximate simultaneous sampling over a very large number of channels by collecting data for all channels in very rapid steps, and then wait for the next scan cycle to repeat the pattern. To request this behavior, use a **TIME** command with a small <*interval*> value, in combination with a relatively large **SCAN** value.

If a **TIME** command is specified, but a **SCAN** command is not, the configuration is presumed to be a legacy DAPL 2000 configuration. For such configurations, the DAPL 3000 system will construct an internal equivalent **SCAN** command with scan interval equal to the number of input channels times the sampling time interval specified on the **TIME** command. This yields a channel list scan time that matches the channel list scan time under the DAPL 2000 system. But, because the xDAP architecture performs sampling actions simultaneously, using multiple channel selectors, the moments at which individual samples are captured will not necessarily align exactly with the other DAP architectures.

Example

```
IDEFINE   dual3phase
  CHANNELS  12
  SET  ipipe0   D0
  . . .
  SET  ipipe11  D11
  SCAN  34.72
  TIME   1.0
END
```

Configure input sampling for an application measuring AC voltages and AC currents on two three-phase high voltage power transmission lines, with 480 samples per 60 Hz waveform, on every signal channel, for a total of 12 channels. To capture all 12 channels at a rate that aligns to the power grid frequency, the `SCAN` time is set to 34.72 microseconds. For each transmission line, the three phase voltages and phase currents are routed through six separate channel selectors, allowing simultaneous three-phase measurements on that line. To capture samples for the two transmission lines with only a very short delay between the measurements, the `TIME` interval is set to 2.0 microseconds.

See Also

SCAN

TRIANGLE

Define a task that generates triangular wave data.

TRIANGLE *(<pk_min>, <pk_max>, <wave_freq>, [<init_phs>,] <samp_time>, <out_pipe>, [<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])*

TRIANGLE *(<amplitude>, <period>, <out_pipe>, [<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])*

Parameters

The parameters for a **TRIANGLE** command are shared by several waveform commands. The common parameter description is provided in the **WAVEVORMS** command description page.

Description

A **TRIANGLE** command defines a task that generates triangular wave data and places the data in <*out_pipe*>. The data sequence is obtained by taking a continuous-time version of the waveform and selecting values along that path, according to the timing parameter <*period*>, or the combination <*wave_freq*> and <*samp_time*>. For a balanced waveform, the output values start at a value of zero, and initially are increasing. One quarter of the way through the cycle, the output values begin decreasing. Three quarters of the way through the cycle, the output values return to increasing, and continue this until the end of the cycle.

For generating a ramp sequence that is easier to configure to reach certain output levels specifically, and that allows different rates for the ascending and and descending portions of the cycle, consider the **BIRAMP** command as an alternative.

Examples

```
TRIANGLE(1000,100,P2)
```

Generate a balanced triangular wave with values ranging from -1000 to 1000, and a period of 100 sample intervals, in 16-bit format. Place the waveform data into pipe P2.

```
TRIANGLE(0.0, 30000.0, 24.0, 50.0, PWAVE)
```

Generate samples of a triangular wave that completes 24 cycles per second when output updates are generated at the specified 50 microsecond intervals. The output values are all non-negative, ranging from 0 to 30000. The resulting waveform data in a floating point format are placed into pipe PWAVE.

See Also

COSINEWAVE, **SINEWAVE**, **SAWTOOTH**, **SQUAREWAVE**, **BIRAMP**

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 provides notification of event locations.
`TRIGGER`

<pre>
The number of values to transfer before the trigger event.
`WORD CONSTANT | LONG CONSTANT`

<post>
The number of values to transfer at or 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 the *<trigger>* reports an event location, the task transfers values from *<in_pipe>* to *<out_pipe>* at that location, retaining the number of values determined by the *<pre>* and *<post>* parameters. The data types of *<in_pipe>* and *<out_pipe>* must match.

Triggering events are produced by another command, typically by analyzing either *<in_pipe>* or a related data stream that has data flowing at the same rate. An event, sometimes called a *trigger assertion*, indicates a position of interest in the data stream. If the data rates in *<in_pipe>* and the analyzed data stream are not the same, the number of samples in the two streams will diverge, and positions in the two streams will not correspond. In many cases, a **TRIGSCALE** command can correct the alignment.

The numbers *<pre>* and *<post>* determine how many samples are retained from the *<in_pipe>* stream.

- The number *<post>* is often called the post-trigger count. A sample that occurs exactly at the event is included in *<post>*. Usually *<post>* is positive, because an event most commonly signals when data of interest start in the data stream. This parameter can be considered the one that determines where to *stop* retaining data. If *<post>* is omitted, the **WAIT** task transfers data continuously once started.
- The number *<pre>* is often called the pre-trigger count. It specifies the number of samples to be retained prior to the location of the event. It is typically zero, but is usually set to a positive number to retain data prior to a triggering event. This parameter can be considered the one that determines where to start retaining data. A value of zero means that no samples prior to the triggering event are retained.
- The number of samples retained for each event is *<pre>* + *<post>*, which must be a positive total. This rule is invariant, even for extension cases where *<pre>* or *<post>* is negative.

These *<pre>* and *<post>* numbers are typically non-negative integers, but they have extended meanings when given negative values.

- A negative value of <*pre*> extends its interpretation about where to start retaining samples. An ordinary positive number means to begin retaining data before the trigger event position. By extension, a negative value means to start retaining data at a delayed position, after the trigger event position. For example, a '-2' value would mean to omit the sample at the triggering event position and the one to follow, and begin retaining data at the sample position after that.
- A negative value of <*post*> extends its interpretation about where to stop retaining samples. An ordinary positive number means to stop after the samples at the event location plus some number of successor locations are taken. By extension, a negative value means to stop taking data before the position of the triggering event. For example, a 1 value would mean to stop retaining the data at the last sample preceding the triggering event.

The <*pre*> and <*post*> values cannot both be negative; otherwise, the total number of retained samples <*pre*> + <*post*> would be negative, and the data transfer would stop before taking any data.

If triggering events are sufficiently isolated, every event results in a separate data transfer. If any new events arrive too soon, so that some of the <*pre*> + <*post*> samples already involved with the current data transfer would be required, the data retention conditions are are impossible to satisfy. In such cases, the events will be ignored. Some processing is involved to evaluate and ignore these extraneous triggering events, so your processing will be most efficient if you configure trigger analysis tasks to avoid producing extraneous events.

When the <*in_pipe*> source of data is an input channel pipe, the **WAIT** command gives it special treatment. The **WAIT** command assumes that the task asserting the trigger tested only a single channel (it would be hard to define a meaningful test on mixed multiple channels). An input channel pipe with `N` signal channels will carry `N` times more data than any one channel. Ordinarily, *this would be a problem,* because the positions found by the triggering analysis for one channel and the positions of data in a combination of `N` signal channels would not align. However, for this special case of input channel pipes, the **WAIT** command accounts for the `N` multiplier automatically. As a general rule: if you generate a trigger event by analyzing one channel from the input channel pipe, you retain capture data from any number of channels from the input channel pipe using a **WAIT** command, without taking any special action. However, you must take care to include the `N` multiplier explicitly when you specify the <*pre*> and <*post*> counts. For example, if you have 100 channels and want two pre-trigger samples and two post-trigger samples in each channel, you must set the <*pre*> and <*post*> counts each to 200.

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, placing the data in 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 starting at 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

LIMIT, **TRIGSCALE**

WAVEFORMS

A generalized description of task configuration parameters for waveform generator tasks.

General parameter form

waveform *(<pk_min>, <pk_max>, <wave_freq>, [<init_phs>,] <samp_time>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

Traditional parameter form

waveform *(<amplitude>, <period>, <out_pipe>,*
[<mod_type1>, <mod_pipe1>] [<mod_type2>, <mod_pipe2>])

Parameters

For the general parameter form:

<pk_min>
The lowest extreme value of the waveform.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<pk_max>
The highest extreme value of the waveform.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<wave_freq>
The fundamental frequency (inverse of cycle time) for the generated wave, in Hertz.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<init_phs>
Optional normalized initial phase shift specifier, dimensionless.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

<samp_time>
The time interval to be spanned by each output sample, in microseconds.
`WORD CONSTANT | LONG CONSTANT | FLOAT CONSTANT | DOUBLE CONSTANT`

For the traditional parameter form:

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

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

Parameters common to both forms:

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

<*mod_type1*>
A first optional modulation keyword, represented as a double-quoted string.
`STRING`

<*mod_pipe1*>
A pipe providing modulation signal data, when first modulation keyword is specified.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

<*mod_type2*>
A second optional modulation keyword, represented as a double-quoted string.
`STRING`

<*mod_pipe2*>
A pipe providing the modulation signal data, when second modulation keyword is specified.
`WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE`

Description

This description is not for one particular waveform generation commands; rather, it is a generalized description of several of the waveform commands to define tasks that operate in very much the same way, except generating different kinds of waveform shapes. These waveform generating tasks include:

- **COSINEWAVE** - cosine wave data
- **SAWTOOTH** - sawtooth wave data
- **SINEWAVE** - sine wave data
- **SQUAREWAVE** - square wave data
- **TRIANGLE** - triangular wave data

Read the description for the particular command to get details specific to that command.

Waveform generating tasks have two available schemes for describing the characteristics of the wave.

General parameter form

This form gives more control over the waveform configuration, and describes a wave in terms of external properties.

The range of the wave is specified, from minimum <*pk_min*>, to maximum <*pk_max*>. Note that if <*pk_min*> is not the negative of <*pk_max*>, this produces a wave with a non-zero DC offset level. The amplitude specifiers must have the same type, but do not need to match the output data type, as long as they both represent values the output data type can represent exactly.

The fundamental wave frequency to be generated is specified by the <*wave_freq*> parameter, in units of Hertz. The time interval <*samp_time*>, in units of microseconds, specifies the update time interval intended to be used to present the output waveform data. The combination of <*wave_freq*> and <*samp_time*> parameters implies the number of samples required to span each cycle of the waveform. The most efficient operation is achieved when this combination yields an integer number of terms per cycle, less than 4096.

The optional <*init_phs*> parameter can be used to determine which waveform value is generated first. When specified, the value must be in the range -1.0 to 1.0. It has an implied multiplier of π radians. If omitted, the default value is 0.0, yielding an initial output value halfway between the minimum and maximum extreme value, provided that this is an allowable value for the particular waveform. For example:

- a phase value of either -1.0 or 1.0 (equivalent to a phase shift of π) has the effect of inverting the waveform.
- a phase value of 0.5 is the equivalent of a phase angle shift of π/2 ; applying this to a **SINEWAVE** command makes it equivalent to a **COSINEWAVE** command.

Traditional parmeter form.

This form cannot produce all of the waveforms that the general parameter form can, but it is often the easiest for producing balanced waveform cycles that span an exact number of sample intervals.

The <*amplitude*> parameter sets the absolute magnitude (positive, one half the peak-to-peak range) of the output wave. The <*amplitude*> specification does not need to match the output data type, but it must represent a value that is possible for the output data type to represent exactly. For example, if the output data type is `long`, 500.0 would be an acceptable amplitude value, but 500.5 would not.

The <*period*> parameter specifies the number of sample values to generate within each waveform cycle. For example, specifying exactly 1024 samples per waveform cycle will align the waveform to a conventional 1024 point FFT data block. Integer values of moderate size are the most efficient, but cycles lengths that are not an exact integer can be specified using a floating point parameter value. Arbitrarily long periods or periods that are mixed fractional numbers are allowed, but the most efficient operation is achieved using integer-length waveform cycles of length less than 4096 terms.

An initial waveform phase or DC offset cannot be specified using the traditional parameter form.

Common to both forms

Depending on the specifications for the range, period, and data type of the waveform, the numerical range of the peak values observed in the <*out_pipe*> stream might not attain the peak values specified by the <*amplitude*>, <*pk_min*>, or <*pk_max*> parameters. See the "**Understanding output range**" subsection below for more information about this.

There are two modulations options. One, both, or neither can be applied. Each modulation specification consists of a selection string (`"AMPLITUDE"`, or `"FREQUENCY"`, including the double quotes) followed by a pipe name identifying the modulation data source. One value is processed from each modulation signal pipe for each output sample value generated. For fixed point modulation data, the maximum representable value (32767 for `WORD` data type, or 2147483647 for `LONG` data type) corresponds to a factor of 1.0, and lower values correspond to proportional fractions less than 1.0. For floating point modulation data, the range is typically 0.0 to 1.0, with natural scaling. The two modulation options are:

1. *Amplitude modulation. Modulation specifier string* `"AMPLITUDE"`
   Values of the modulation signal multiply the corresponding values of the original waveform one-for-one. This modulation applies only to the varying part of the wave, and does not shift the DC offset level. So, for example, the range of the waveform is 0 to 10000 when a modulation factor 1.0 (i.e. "no modulation") is applied. But waveform output values are all equal to 5000 (the DC offset implied by the range limits) when a modulation factor 0.0 is applied.

2. *Frequency modulation. Modulation specifier string* `"FREQUENCY"`
   Values of the modulation signal act as a multiplier on the waveform frequency. A frequency modulation factor of 1.0 produces data at the nominal frequency, as established by the <*period*> or the combination of the <*wave_freq*> and <*samp_time*> parameters. As the modulation factor reduces, the frequency reduces proportionally and the cycle length grows inversely. For example, if the <*period*> is 1000, and the values

from the frequency modulation pipe are 0.5, the frequency is cut in half and the effect is the same as locally elevating <*period*> to 2000.

Understanding output range

Waveform values are calculated at specific discrete points lying along a theoretical continuous and *ideal* waveform curve, in much the same way that a continuous waveform would be sampled at points equally spaced in time to produce a sampled waveform. This causes a difficulty for waveforms that are not continuous (**SQUAREWAVE** and **SAWTOOTH** commands) and have no defined value at certain sample locations. Commands can use different strategies to resolve this. See the individual command descriptions for these details.

If you use parameters in the traditional form, discontinuities can sometimes be avoided by displacing the <*init_phs*> parameter (for example, by ½ of the effective phase advance per sample). While this might result in a well-balanced wave cycle that has all of the correct harmonic and amplitude properties, the nominal peak values might not be attained exactly.

Examples

```
SAWTOOTH(1000,100,P2)
```

Generate a balanced sawtooth wave signal using the traditional parameter form, ranging from -1000 to 1000, with a period of 100 samples, in 16-bit format suitable for delivery to D-to-A output hardware. Place the waveform data into pipe P2.

```
SINEWAVE(0.0, 10000.0, 2500.0, 4.0, -0.25, PWAVE, "AMPLITUDE", PAMOD)
```

Generate samples of a sine wave using the general parameter form. Values will range from 0.0 to 10000.0 (like a sine wave of amplitude 5000 riding on a DC offset of 5000) while the modulation factor is at its maximum value 1.0. The generated waveform is for a 2500.0 Hz wave if an updating interval 4.0 microseconds is used as specified. Specify an initial phase adjustment of -0.25, which is the equivalent of -0.25 π radians at the first evaluation of the *sine* function.

See Also

COSINEWAVE, **SAWTOOTH**, **SINEWAVE**, **SQUAREWAVE**, **TRIANGLE**, **BIRAMP**