

DAPL 3000 Extensions for xDAP Family

*Supplement to
DAPL 3000 Manual, Version 1*

Version 1.02

Microstar Laboratories, Inc.

This manual is protected by copyright. All rights are reserved. No part of this manual may be photocopied, reproduced, or translated to another language without prior written consent of Microstar Laboratories, Inc.

Copyright © 2009-2010, Microstar Laboratories, Inc.

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

Microstar Laboratories, DAPcell, DAPtools Software, Data Acquisition Processor, DAP, xDAP, DAPL, DAPL 2000, DAPL 3000, Developer's Studio for DAPL, and DAPstudio are trademarks of Microstar Laboratories, Inc.

Microsoft, MS, and Windows are trademarks of Microsoft Corporation. Intel is a registered trademark of Intel Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Microstar Laboratories requires express written approval from its President if any Microstar Laboratories products are to be used in or with systems, devices, or applications in which failure can be expected to endanger human life.

Contents

- 1. Introduction..... 2**
- 2. New Notations..... 4**
 - Number Notations 4
 - Signal Routing Notations 5
 - Multiple-Sampling Notations 6
- 3. New Commands and Variants..... 9**
 - DEXPAND 10
 - DISCARD**..... 12
 - DIGITALIN 14
 - MIXRFFT 16
 - MRBLOCK..... 21
 - MTSFILT..... 23
 - SCAN (for input definitions) 25
 - TIME (for input definitions) 28
 - WAIT 30

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.

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 $0000FF00
```

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
```

The default data type used to represent fixed-point numbers is a signed 16-bit integer. The default data type used to represent floating-point numbers is 32-bit IEEE 754 floating point. The other supported numeric data types are signed 32-bit integers and IEEE 754 double precision. When there is no context or override to otherwise decide the data type for representing a number, the value must be representable by the default data type.

Some contexts have advance information about data types, and more flexibility is allowed in the number notations. In the following examples, the DAPL system knows the data type specified for a `CONSTANT` element, 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 = 100.0
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 can accept number parameters with 16-bit integer or 32-bit integer data type.

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

Lacking any other information, the DAPL system will interpret the task parameter notation as a 16-bit integer data type if the value is representable as a 16-bit integer, or as a 32-bit integer data type if the value is representable as a 32-bit integer. Lacking any other information, it will represent a floating-point notation as a 32-bit float value.

For building some processing tasks, the DAPL system will have specifications providing extra information about parameter types. If it is known that a parameter must be of a certain data type, the DAPL system can force the number representation to that data type, allowing more flexibility in the notation.

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 supported suffix 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 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.

```
DEFINE 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 to produce and dispose of a large excess of samples. If two sampling configurations are used, one could support the high-rate channels and the other support 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 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, nothing can prevent them from trying to operate that device 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. Listed input procedures will be started as *designated input procedures* if referenced by name explicitly within one or more of the listed processing configurations, or if referenced explicitly by procedures that are already running.
3. If the start group lists any other input procedure configuration that is not explicitly designated anywhere in the listed or currently running processing procedures, the DAPL system will attempt to start it as a *default input configuration*. A default procedure will provide data for any input channel pipe references that do not 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 will be diagnosed as an error. Once started as a default procedure, it is not possible to “upgrade” a running input procedure into a designated procedure later, to allow some other configuration to 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 procedure should be started prematurely as a *default input configuration* if needed as an explicitly designated procedure for processing to be started later.
6. If no input procedure names are listed on the START command, all of the currently defined procedures will be started as one *start group*. All explicitly designated input sampling configurations, referenced in any of the defined processing procedures, will be started. If an unstarted input procedure remains, it will be started as the *default input configuration*.

Avoid defining unused sampling configurations. Suppose, for example, that a processing procedure P and the 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 all three procedures without error. Procedure B 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 likely will lead to a 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, with no corresponding samples from the slow stream to pair with them. 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
```

3. 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.

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

DEXPAND 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.

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. Unexpected encodings will be interpreted incorrectly as data.

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

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 signals intended for other boards.

Note: The encoding generates a data stream with multiple values for encoding 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 conditioning board series.

See Also

[DIGITALOUT](#), [DACOUT](#), [ODEFINE](#), [OPTIONS](#), [OUTPUT](#)

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. For example, the **HIGH** processing command determines the highest value statistic for an input data block. It also provides an optional output index for identifying which term in the data block produced that maximum. For special situations where you need *only* the index terms, the **HIGH** command still leaves you with a pipe containing the high values. If you don't apply the **DISCARD** command or do something else with that data, eventually the pipe will fill to capacity and block the **HIGH** processing command from performing any further processing.

You do not need to use the **DISCARD** command to remove unused data from input channels. For example, suppose that you artificially increase the number of logical input channels from 8 to 10, by defining two extra input channel pipes IPipe8 and IPipe9. This makes it much easier to align precisely to a scan interval of 100 microsecond. You can disregard the sample values from channels IPipe8 through IPipe9, because these are never taken out of the main buffer memory, where they 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 the 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 **DISCARD** command the best opportunity to perform its cleanup operations.

Example

```
DISCARD(pLowest)
```

The **LOW** processing command produced a pipe pLowest containing the lowest-valued data values that you do not want. You also do not want data to accumulate and cause problems. Use the **DISCARD** command to dispose of the undesired data.

`DISCARD(plmag1, plmag2, plmag3, plmag4)`

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 `plmag1`, `plmag2`, `plmag3`, and `plmag4` using the **DISCARD** command.

See Also

PCOUNT

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 **DIGITALIN** 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 **DIGITALIN** 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 **DIGITALIN** 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, **IDEFINE**

MIXRFFT

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

```
MIXRFFT ( <N>, [<direction>], [<window>], | <windtype>[<alpha>],  
  <pipeinR>, [<pipeinl>], [<blocks>], <post>, <pipeoutR> [, <pipeoutl>] )
```

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>, <pipeinl>

Pipes for blocks of input data.
WORD PIPE | LONG PIPE | FLOAT PIPE | DOUBLE PIPE

<pipeoutR>, <pipeoutl>

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 $\langle N \rangle$ from pipes $\langle \text{pipeinR} \rangle$ and optionally $\langle \text{pipeinI} \rangle$ for complex-valued data. If the data are real-valued, omit the $\langle \text{pipeinI} \rangle$ 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 $\langle \text{pipeoutR} \rangle$ and $\langle \text{pipeoutI} \rangle$. Some processing options produce only one output stream, and for these omit $\langle \text{pipeoutI} \rangle$.

When applied to limited-size data blocks with lengths that are an exact power of two, the results are equivalent to the results of the **FFT** command. 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. In general, the **MIXRFFT** processing is somewhat less efficient, but the net speed differences are relatively small, and this can sometimes lead to relative efficiency. For example, you might get a faster result by using the **MIXRFFT** command on a block $\langle N \rangle$ of 5000 terms, rather than rounding $\langle N \rangle$ up to 8192 and applying the radix-2 **FFT** command.

Block Length

There are some restrictions on the block length parameter $\langle N \rangle$, 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 factorable into an arbitrary number of 2, 3, or 5 factors, 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 $\langle \text{direction} \rangle$ 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 $\langle \text{direction} \rangle$ 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 $\langle N \rangle / 2$ 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 re-combined 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 contains no new information. However, it can also be useful for complex data if it is known that there is no useful information in the high frequency terms. 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

[DFT, FFT](#)

Also refer to the chapter “Fast Fourier Transform” in the DAPL 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. The `MRBLOCK` command 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 the MRBLOCK command, data display processing on a PC host can accept new data at its own pace.
3. Status summary reports. Some processes on a PC host are driven by requests from external data collection activity, typically through a network. The MRBLOCK command allows the host application to pass these requests through to the processing on the DAP, which returns the most recent status measurements.

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 **MTSFILT** command is intended as a lower-cost alternative without the special hardware.

The **MTSFILT** command receives multiplexed input data from the *<pinmux>* pipe. The number of multiplexed channels, common to the input and output streams, is specified by the *<nchan>* parameter. The number of channels in groups sampled simultaneously by the hardware is specified by the *<ngroup>* parameter. If omitted, the default value of the *<ngroup>* parameter is 1, meaning that there is only one converter in the hardware. A nontrivial DSP resampling analysis reconstructs 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 that would be obtained from actual simultaneous sampling in 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 a **COUNT** command in your input sampling procedure, you must allow extra samples to supply these initializer samples.

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

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; and for this case, the DAPL 3000 system will internally construct a **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 above, 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

```
DEFINE 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

[TIME](#)

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 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 as closely as possible, by collecting data for all channels rapidly and then waiting for the next scan cycle to repeat the pattern. To request this behavior, use a **TIME** command with a small <interval> 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 **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 match the other DAP architectures.

Example

```
DEFINE 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 the correct rate, 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 a minimum delay between the measurements, the *TIME* interval is set to 1 microsecond.

See Also
[SCAN](#)

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 <trigger> reports an event location, **WAIT** then transfers values from <in_pipe> to <out_pipe>, retaining a 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.

The numbers <pre> and <post> determine how many samples are retained from the <in_pipe> stream. These numbers are typically nonnegative integers, but they have extended meanings when given negative values.

- 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 often set to a positive number to retain data leading up 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 $\langle pre \rangle + \langle post \rangle$, which must be a positive total. This rule is invariant, even for extension cases where $\langle pre \rangle$ or $\langle post \rangle$ is negative.

Using negative values for a $\langle pre \rangle$ or $\langle post \rangle$ count extends the ordinary roles of these parameters for determining where to start or stop retaining data.

- A negative value of $\langle pre \rangle$ extends its interpretation about where to start retaining samples. An ordinary positive number means to begin retaining data sooner, before the trigger event position. By extension, a negative value means to start retaining data later, 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.
- A negative value of $\langle post \rangle$ extends its interpretation about where to stop retaining samples. An ordinary positive number means to stop after the samples at the event location and 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 $\langle pre \rangle$ and $\langle post \rangle$ values cannot both be negative; otherwise, the total number of retained samples $\langle pre \rangle + \langle post \rangle$ would be negative, and the data transfer would stop before taking any data.

If triggering events are sufficiently isolated, every event results in a data transfer. If any events arrive too soon, and would require some of the $\langle pre \rangle + \langle post \rangle$ samples that are involved with the current data transfer, those events will be ignored. Some processing is involved to evaluate and ignore extraneous triggering events, so your processing will be most efficient if you configure trigger analysis tasks to avoid producing extraneous events.

When the $\langle in_pipe \rangle$ 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. When the input channel pipe has a channel list with N signal channels, the combined amount of data it carries is N times greater than in any one channel. Ordinarily, this would be a problem, because the positions found by the triggering analysis and the positions of data in the combined pipe would not match. However, for this special case, the **WAIT** command applies the N multiplier automatically. As a general rule: if you generate a trigger event by analyzing one channel from the input channel pipe, you can 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 $\langle pre \rangle$ and $\langle post \rangle$ counts. For example, if you have 100 channels and want two pre-trigger sample and two post-trigger samples in each channel, you must set the $\langle pre \rangle$ and $\langle post \rangle$ 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

[DLIMIT](#), [LIMIT](#), [LOGIC](#), [TRIGSCALE](#)