Supplement to
Developer's Toolkit for DAPL Manual

Command module developer's
toolkit for DAPL 2000
operating system

Supplement to Version 5.03

Microstar Laboratories, Inc.

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

Supplement to Part Number MSDTDM500-0104

Overview

This document provides supplementary information about additional DSP service functions available starting with release 5.03 of the Developer's Toolkit for DAPL.

The new functions very much resemble the FIR filtering and FFT transform functions provided by preceding versions of the Developer's Toolkit. The difference is that the new functions support all of the data stream types available under the DAPL system. This makes the full capability of the built-in `FIRFILTER` and `FFT` commands available to custom command developers for customized DSP processing.

The new functions provide all of the functionality of the older forms, so there is no reason to use the old function forms for new applications. But on the other hand, there is no need to convert existing applications that do not use any of the extended functionality.

Who Might Benefit?

Developers who need transforms or filters with wide dynamic range or extra precision can benefit from the new DSP functions. The best choice of data type depends on each application's requirements.

- WORD This type provides the fastest processing, is the most compact, and is the natural data type for processing sampled data in most cases.
- LONG This type provides twice as much storage, with consequently slower processing time as more data pushes through the processor cache, but more precision than most applications will ever need.
- FLOAT This type offers not quite as much precision as LONG, not quite the speed of WORD, but scales very well, allowing a wide dynamic range. It is easy to use, often more compatible with host applications.
- DOUBLE This type is bulky and slow, but extremely versatile. It offers the widest range and the best precision for demanding applications.

As an example, consider the FFT of a broad-band signal. *Broad-band* means a lot of frequencies contribute to the waveform. Because there are many frequencies contributing signal power, none of them can contribute very much. Consequently, when an FFT analysis is performed, many of the transform terms are small.

Rounding to an even integer sometimes produces a disproportionately large effect, for example, erratic estimates of phase. Using a floating-point representation, a small or large value makes no difference, because each value contains separate precision and scaling components.

As a second example, suppose that a digital filter is applied to a low-level signal. Rounding of the filtered output to integer levels is roughly the equivalent of introducing an interfering white noise. This might have the effect of degrading the signal-to-noise ratio in the filtered data stream.

As a practical matter, floating-point representations avoid most of the problems of rounding, limiting, and scaling. When application demands are not severe, floating-point operations are sometimes easier to work with.

What Systems Are Required?

Custom command modules developed using the 5.03 release of the Developer's Toolkit for DAPL require the DAPL 2000 2.53 operating system, or a more recent version, and a Data Acquisition Processor model supported by that system. Custom command modules developed using the 5.03 release of the Developer's Toolkit for DAPL will not work on Data Acquisition Processor boards running an earlier version of the DAPL system.

Custom command modules using floating-point data types with the new DTD functions will be slow on DAP models whose processors do not provide floating-point hardware support.

The New Function Set

The new functions in the 5.03 release of the Developer's Toolkit for DAPL are the following:

`fftb_init`	replaces	`fft_init`
`fftb_chngbuf`	replaces	`fft_chngbuf`
`fftb_request`	replaces	`fft_request`
`fftb_postop`	replaces	`fft_postop`
`firb_init`	replaces	`fir_init`
`firb_advance`	replaces	`fir_advance`
`firb_change`	replaces	`fir_change`
`firb_apply`	replaces	`fir_apply`

The functions act very much like their older counterparts. The main difference is that the functions use *generic storage types* rather than short integer types only. The *generic storage types* are defined by the file GENTYPES.H, which is included automatically with each compiled module by the main DTD.H header file. When accessing stored items using the *generic types* it is necessary to specify the data type with each access. Circumventing the compiler's type checking in this manner is inherently dangerous, but unfortunately there is no better means for moving data through an interface where various data representations might be allowed. Be sure to pick one data type and stay with it. If you assign data using one type, and later tell the compiler to access the same binary values as if they are a different data type, the compiler will do exactly what you say but the results will not have any meaning.

For example, suppose that you define two 91-term arrays of FIR filtering coefficients in a `float` notation.

```
static float   coeffset1[91];
static float   coeffset2[91];
GENERIC_PTR   gpCoeff1;
GENERIC_PTR   gpCoeff2;

gpCoeff1._pFloat = &(coeffset1[0]);
gpCoeff2._pFloat = &(coeffset2[0]);
```

These functions will then accept the *generic pointers*, but of course supplementary information from the control structure (the `FIRB` or `FFTB` block) is needed to interpret the data within the generic storage. For example:

```
firb_change( myFIRB, gpCoeff2, 91, iScale, iDecimate);
```

To associate a data type with the generic buffer storage, so that functions like `firb_change` can access the stored data correctly, initialization functions must establish a data-stream type. The appropriate data type codes are defined in the file DTDCNSTS.H, which is included automatically with each compiled module by the main DTD.H header file. For FIR filtering, you will need to select one of the following data type codes for the input stream.

```
FIR_WORDTYPE
FIR_LONGTYPE
FIR_FLOATTYPE
FIR_DBLTYPE
```

Similarly, for FFT transforms, you will need to select one of the following data type codes.

```
FFT_WORDTYPE
FFT_LONGTYPE
FFT_FLOATTYPE
FFT_DBLTYPE
```

If data type flags are omitted, the data-stream type defaults to `WORD` data type.

Just as data streams have a data type, windowing vectors have a data type as well. If you specify a custom-designed windowing vector, you can specify its data type using one of the following window data type codes.

```
FFT_WORDWIN
FFT_LONGWIN
FFT_FLOATWIN
FFT_DBLWIN
```

`LONG` data type is suggested for all fixed-point transforms for best accuracy. For floating-point data types, the windowing-vector data type should match the data-stream type.

An FFT Example

Suppose that an FFT must operate on real-valued floating-point data, producing 512 output values for 512 input values, using a Hamming window, with power density as the output format. The following initialization function call will set up the FFT for this.

```
FFTB   myFFT;
GENERIC_PTR   gpRealbuf, gpImagbuf;
GENERIC_PTR   gpWindow;
float   myRealArray[512];
int   options;

gpRealbuf._pFloat = &(myRealArray[0]);
gpImagbuf._pVoid = NULL;
gpWindow._pVoid = (void *)FFT_HAMMING;
options = FFT_REALIN | FFT_REALOUT | FFT_FULLOUT |
FFT_FLOATTYPE;

myFFT = fftb_init(
    512,     /* length of transform block */
    gpRealbuf, gpImagbuf,     /* Input/output storage */
    gpWindow,     /* window selection code */
    FFTDIR_FORWARD,    /* transform to frequency domain */
    FFTPOST_POWER,    /* post process to compute power
density */
    options );
```

Comparing to the old `fft_init` function, we can note that the new version is very much the same except for the following.

- Input/output storage are generic instead of restricted to short int arrays.
- Window is passed as a generic pointer instead of long int.
- There is no “fast/accurate” parameter for back compatibility with version 4 of the Developer’s Toolkit, as this has no purpose with 32-bit processors.
- The options include the one additional flag value for data type

An FIR Filtering Example

Suppose that we require a FIR filter to apply a vector of 71 filtering coefficients, operating on real-valued floating-point data, decimating by 4 and reducing the values by a scaling factor 1024. The following initialization function call will set up the FIR filtering for this.

```
FIRB   myFIR;
GENERIC_PTR   gpRealCoeffs;
GENERIC_PTR   gpMyData;
GENERIC_SCALAR   gScale;
float  myFloatVector[71];
float  myInputBuffer[512];
int   typeoption;

gpRealCoeffs._pFloat = &(myFloatVector[0]);
gpMyData._pFloat = &(myInputBuffer[0]);
gScale._float = (float)1024.0;
typeoption = FIR_FLOATTYPE;

myFFT = firb_init(
    gpRealCoeffs,     /* filter characteristic */
    71,     /* length of my filter characteristic */
    gScale,    /* scaling divisor */
    4,    /* decimation, retain 1 term of 4 */
    typeoption );
```

Comparing to the old `fir_init` function, we can note that it is very much the same, with the following exceptions.

- Coefficients match the stream in data type and are passed using a generic pointer.
- The scaling is a numeric divisor rather than a bit-shift reduction when using floating point.
- The new option flag indicates the data type.

Function Reference

This section of the Developer's Toolkit for DAPL manual supplement provides complete details of the new FFT and FIR filtering functions with multiple data type support.

fftb_chngbuf

Switch FFT to a different set of input/output data buffers.

```
void  fftb_chngbuf (
   FFTB * fft,                    // FFT control block handle
   GENERIC_PTR  real,             // Generic pointer to storage
   GENERIC_PTR  imag              // Generic pointer to storage
   );
```

Parameters

fft

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

real

Type-independent pointer to data storage for real-valued terms.

imag

Type-independent pointer to data storage for imaginary-valued terms.

Return Values

There is no return value.

Description

The function **fftb_chngbuf** changes the real and imaginary data pointers previously installed in an FFTB. The control block is identified by the handle *fft*. This function allows a single FFTB to be "switched" from one block of data storage to another, allowing operations upon multiple data streams. The change takes effect with the next operation that uses the specified FFTB.

See Also

fftb_init

fftb_init

Prepare for an FFT by defining an FFT control block.

```
FFTB  *fftb_init (
   int size,
   GENERIC_PTR  realbuf,          // Pointer to storage
   GENERIC_PTR  imagbuf,          // Pointer to storage
   GENERIC_PTR  windowvec,        // Enumeration pointer
   int direction,                 // Enumeration
   int post,                      // Enumeration
   int options                    // Bit mask
   );
```

Parameters

size

The length of the FFT and required data areas. It specifies the number of complex input items N, where $N = 2^M$ for integer M in the range 2 to 14.

realbuf

Generic pointer to a data storage area for real-valued terms. The binary format used for this storage must match the data type specified in the options parameter.

imagbuf

Generic pointer to a data storage area for imaginary-valued terms. The *imagbuf* pointer can be assigned NULL if imaginary data storage is not needed for either input data or output data.

windowvec

A generic pointer to an array of windowing coefficients, matching in size the length of the FFT data block, and with type as indicated by the *options* parameter. Optionally, if this pointer value is assigned one of the predefined codes for window operators, the windowing vector will be supplied automatically. Predefined enumeration codes include the following:

```
WINDOW_RECTANGULAR
WINDOW_HANNING
WINDOW_HAMMING
WINDOW_BARTLETT
WINDOW_BLACKMAN
```

direction

One of the following codes:

```
FFTDIR_FORWARD
FFTDIR_REVERSE
```

post

One of the following codes:

```
FFTPOST_DEFER
FFTPOST_REAL
FFTPOST_CPLX
FFTPOST_POWER
FFTPOST_MAGNITUDE
FFTPOST_MAG_PHASE
```

options

"Flag" bits that are combined using bitwise OR operations to select additional processing options, and to specify data types of streams and window vectors. One option from each of the groups may be selected:

```
FFT_REALIN
FFT_CPLXIN

FFT_SEPARATED
FFT_PAIRWISE

FFT_HALFOUT
FFT_FULLOUT

FFT_WORDTYPE
FFT_LONGTYPE
FFT_FLOATTYPE
FFT_DBLTYPE

FFT_WORDWIN
FFT_LONGWIN
FFT_FLOATWIN
FFT_DBLWIN
```

Return Values

The function returns a pointer to an FFTB configuration block, used by all other FFT functions.

Description

The function **fftb_init** allocates an FFT control block structure and initializes it with the options that define the characteristics of the FFT and its related operations. The actual operations are performed separately.

The *realbuf* and *imagbuf* parameters specify pointers to data storage areas for real-valued and imaginary-valued terms respectively. The *imagbuf* pointer can be NULL if imaginary data storage is not needed for either input data or output data. The **fftb_request** function will fetch input data using these pointers. Depending on processing options, it also uses the same storage for returning results.

The storage must be allocated by the custom command and must cover all input and output requirements. The **ralloc** function can be used to obtain storage blocks. The number of items to reserve is closely related to the number specified by the *size* parameter. Some examples:

- *Complex input data.* When the input data is complex and stored in multiplexed fashion using the FFT_PAIRWISE option, both real and imaginary terms are provided by one data source, the *realbuf* array. The *realbuf* array requires *2 * size* terms.
- *Half-length output data.* With processing options FFT_HALF and FFT_CPLX, the number of real input terms equals *size*, but after the transform, *1/2 * size* terms each are used for storing the real and imaginary results separately.
- *Power output post-processing.* Using real input data and the post-processing options FFTPOST_POWER and FFT_FULLOUT with WORD data, the number of terms returned is *size*, but the returned data type is long int rather than short int. The *realbuf* array must allow for *2 * size* terms rather than *size* terms in its memory allocation.

The *window* parameter specifies either a pre-defined enumeration code for a window operator or a generic pointer to an array of length *size* containing window-operator terms. The DAPL system can distinguish pointer values from enumeration codes, so the meaning of the parameter is unambiguous. Unfortunately, C++ syntax does not allow a data type that can be either a pointer to various types or a scalar value, so a GENERIC_POINTER is used. An enumeration code must be cast to a void pointer to assign into this data type. A code in the *option* parameter specifies the window data-array type. Type long int should

be used for integer-valued data streams to preserve accuracy. Data types matching the input streams should be used for floating-point types.

The *direction* parameter specifies a forward transform, typically used for transforming from time-domain data to frequency-domain, or a reverse transform, typically for transforming from frequency-domain data to time-domain.

The *post* parameter specifies the kind of post-processing operations to perform. FFT results are often displayed as magnitude and phase or power density rather than the raw complex numbers. The data types and buffer organizations specified in the *options* parameter might be restricted by the choice of post-processing; for example, a power-density result never produces a complex-valued output.

The *options* parameter provides control over the organization of the input data and the output results. The *option* codes also provide information about data type. See Chapter 7 of the Developer's Toolkit for DAPL Manual for instructions on selecting option codes to describe the organization of input and output buffers. The additional data type codes specify the data types of the data streams and windowing vector.

```
FFT_WORDTYPE
FFT_LONGTYPE
FFT_FLOATTYPE
FFT_DBLTYPE

FFT_WORDWIN
FFT_LONGWIN
FFT_FLOATWIN
FFT_DBLWIN
```

See Also

`fft_request`, `ralloc`

fftb_postop

Apply post-processing to an FFT result.

```
int fftb_postop (
   FFTB *fft,                          // FFT control block handle
   GENERIC_PTR realbuf,                // Pointer to storage
   GENERIC_PTR imagbuf,                // Pointer to storage
   int post,
   int options
   );
```

Parameters

fft

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

realbuf

Pointer to a data storage area for real-valued terms.

imagbuf

Pointer to a data storage area for imaginary-valued terms.

post

One of the following codes:

```
FFTPOST_REAL
FFTPOST_CPLX
FFTPOST_POWER
FFTPOST_MAGNITUDE
FFTPOST_MAG_PHASE
```

options

"Flag" bits that are combined using bitwise OR operations to select additional processing options related to post-processing. One option from each of these groups may be selected:

```
FFT_SEPARATED
FFT_PAIRWISE

FFT_HALFOUT
FFT_FULLOUT
```

Return Values

The function returns a nonzero error code if a parameter error is detected, or a 0 code if the operation is completed.

Description

The function **fftb_postop** performs post-transform processing on an FFT result after FFT computations are completed but before a subsequent FFT is performed using the same FFTB configuration block. This operation allows additional processing, beyond that done by the original FFT operation. That means it is possible to preserve the original transform values and then apply more than one style of post-processing without losing any information.

When the FFTPOST_DEFER option is selected in the call to the **fftb_init** function, the **fft_request** function does not return any data. Using this option, a call to the **fftb_postop** function is required to access the transform computation results.

The parameters are very similar to the processing options of the **fftb_init** function, but related only to post-transform processing.

The *realbuf* and *imagbuf* fields must specify pointers to data storage areas for real-valued and imaginary-valued output terms. The custom command must allocate sufficient storage to cover all output requirements. The data type in the buffers is determined by the initial configuration and cannot be changed by this function.

The *post* option specifies the desired post-processing operation. The *options* parameter provides a limited number of options for organizing real and imaginary parts of the transform results.

See Chapter 7 in the Developer's Toolkit for DAPL Manual for more information about the various post-processing options.

See Also

fftb_init, **fftb_request**

fftb_request

Initiate FFT processing.

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

Parameters

fft

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

Return Values

There is no return value. The results of the FFT computation are returned in the FFT control block.

Description

The function **fftb_request** initiates FFT computation, using the configuration previously established by the **fftb_init** function. The custom command is required to place the input data for the FFT operation into the storage arrays prior to making this function call.

There is no difference between this function and the older **fft_request** function. The alternate form simply provides consistent naming conventions.

See Also

fftb_init

firb_advance

Bypass selected FIR filter computations for data reduction or decimation.

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

Parameters

fir

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

count

A value specifying the number of items to be removed from the data source.

Return Values

The function returns the number of additional items that must be removed from the data source.

Description

The function **fir_advance** is an optional function to advance data through a FIR filter internal shift register, bypassing selected filtering operations. A normal filtering operation removes old data from the filter, adds new data to replace them, and then performs filter computations. The **firb_advance** function removes old data, without replacing with new data, and without performing any filter computations.

The function **firb_advance** reports the number of additional items that must be removed from the data source. If just a few items are bypassed, and the filter shift register is not emptied, the function returns the value zero, and filtering resumes automatically when enough new data are provided by function **firb_request** to refill the shift register. If the *count* is larger than the number of items present in the shift register, **firb_advance** reports the number of additional items that must be skipped from the source stream by the calling program before refilling the filter shift register.

The most common application of function **firb_advance** is data skipping, for example, capturing data at a high sampling rate to preserve high-frequency information, but eliminating large blocks to avoid excessive data volume. Another application is specialized decimating filters.

There is no difference between this function and the older **fir_advance** function. The alternate form simply provides consistent naming conventions.

See Also

firb_request

firb_change

Modify FIR characteristics.

```
int firb_change (
   FIRB *fir,                    // FIR filter control block handle
   GENERIC_PTR  coeffs,          // Pointer to coefficient array
   int length,
   GENERIC_SCALAR  scale,        // Shift or gain adjustment
   int decimate
   );
```

Parameters

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

coeffs
An array containing the coefficients that determine the computational characteristics of the filter.

length
A value specifying the number of terms in the *coeffs* array, up to 1024.

scale
A value specifying a non-negative scaling constant. For fixed-point data types, it specifies a final scaling shift. For floating-point data types, it specifies a final scaling divisor.

decimate
A non-negative number specifying a decimation rate.

Return Values

If the function succeeds and the change is installed successfully, the return value is 0. If the space previously allocated for the filter is not sufficient, or if any of the new filter characteristics are invalid, a nonzero error code is returned.

Description

The function **firb_change** changes filter characteristics after initialization by the **firb_init** function. The parameters of this function correspond closely to the

parameters of the **firb_init** function. This function modifies the FIRB content but does not allocate any new elements or change data type.

This function should be used with care, because it can affect efficiency, output continuity, phase, and latency. For example, if the filter is made longer, the internal shift register previously fully filled is suddenly not fully filled. The filter will cease generating output values until a number of new samples are provided. Similarly, reducing the filter length can leave the filter shift register somewhat overfilled, causing an unexpected burst of output results the next time a filtering operation is requested. The filter reserves extra space for computational efficiency when it is initialized, but efficiency may drop if that extra space is consumed by a longer filter structure. Usually, the best strategy for changing filter characteristics is to keep the length the same, with enough padding zeroes at the ends to allow for filter length changes.

The safest way to "tune" coefficients is to compute them in separate array storage, and then switch to the new array with a call to **fir_change**. However, it is possible to adjust coefficient values in their original storage provided that no concurrently active filters share the coefficients.

All parameter values must be specified. If some of the parameters are unchanged, specify the old values.

See Also

firb_init

firb_init

Prepare for FIR filtering by defining a FIR control block.

```
FIRB *firb_init (
   GENERIC_PTR  coeffs,                  // Pointer to coefficient array
   int length,
   GENERIC_SCALAR scale,
   int decimate,
   int option
   );
```

Parameters

coeffs
A generic array containing the coefficients that determine the computational characteristics of the filter.

length
A value specifying the number of terms in the *coeffs* array, up to 1024.

scale
A value specifying a non-negative scaling constant. For fixed-point data types, it specifies a final scaling shift. For floating-point data types, it specifies a final scaling divisor.

decimate
A non-negative number specifying a decimation rate.

option
A flag indicating the type of data in the input stream, coefficient vector, and scaling factor. It must be one of the following:

```
FIR_WORDTYPE
FIR_LONGTYPE
FIR_FLOATTYPE
FIR_DBLTYPE
```

Return Values

The function returns a pointer containing a handle value required by all subsequent filter operations. It is non-NULL if allocation and initialization are successful.

Description

The function **firb_init** allocates a FIR digital filter control block structure and initializes it with the options that define the characteristics of the filter. Actual filtering operations are performed later.

The coefficients, which determine the computational characteristics of the filter, are provided to the function **firb_init** in the array *coeffs*. The *length* parameter specifies the number of terms in the *coeffs* array, up to 1024. The length of the filter equals the length of this vector. The data type of the coefficients must match the data type of the data stream to be filtered.

The *scale* parameter specifies a non-negative scaling constant. The scaling is applied after other filter computations, dividing the intermediate filter result by the specified amount to produce the final filter result. For fixed-point data types, the scale factor must be an exact integer power of 2 and smaller than the *length* parameter. The final scaling operation is bypassed if the *scale* parameter has a value 1 or 0, interpreted as “divide by 1” and “no scaling” respectively. For floating-point data types, the scaling factor also specifies a magnitude reduction, but it is not restricted to a power of 2, and it must have the data type that matches the data stream. For no scaling, specify the value 1.0.

The *decimate* parameter is a non-negative number. If the *decimate* parameter is greater than 1, one filter value is computed and then *decimate-1* values are skipped, so that *decimate* values are consumed for each filter output value generated. A *decimate* value of 1 or 0 indicates that no decimation is to be applied, and each input value will generate one corresponding output value.

The *option* parameter is a code specifying the data type for the filtering operation. For consistent indexing and preservation of precision, the data types of the data stream, coefficient vector, and scaling divisor must all match the specified data type.

The returned value is a handle required by all subsequent filter operations. If this returned pointer is a NULL pointer, there is a parameter error, and the **firb_init** function was unable to configure a filter as specified.

See Chapter 7 of the Developer’s Toolkit for DAPL Manual for more information about the meaning and application of the various configuration options.

See Also

firb_change, **firb_request**

firb_request

Perform FIR filter processing using data provided.

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

Parameters

fir

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

data

An array containing the data to which the filter is applied. Result values will replace the original data in this array.

count

The number of filter input values provided in the data array.

Return Values

If the amount of data provided in the `data` array is not sufficient to fill the internal filter shift register, and computations cannot proceed, a 0 is returned. If there is sufficient data in the shift register to perform some filtering computations, the returned value indicates the number of results generated.

Description

The function **`firb_request`** initiates digital filter computations, using the configuration previously established by the **`firb_init`** function. The filter operation is applied to data provided in the *`data`* array. The *`count`* parameter specifies how many items are provided to the filter. Result values replace the original data in the *`data`* array. The number of output terms computed and buffered is returned. The output data type matches the data type of the input stream as established by the **`firb_init`** function. Contents of any *`data`* array locations not used for returned results are undefined.

See Also

`firb_init`