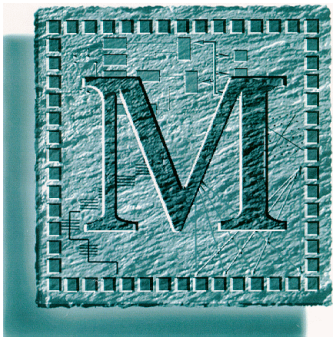

MASSANA

FILU-50 Programming Model

Massana Inc.
51 E. Campbell Ave
Campbell, CA, 95008.

Tel: (408) 871 1414
Fax: (408) 871 2414

Massana Research Ltd.
5 Westland Square
Dublin 2, Ireland.

Tel: (+353 1) 602 3999
Fax: (+353 1) 602 3977

E-Mail: info@massana.com
www.massana.com

Version 1.2
15 April 1999

Contents

1. FILU Programming Model

1.1 Introduction

The FILU is a pre-programmed DSP co-processor which has a *run time library* of DSP functions which are accessed by C function calls from a Host processor.

This document describes how the Host is the programmed to call these functions. Included are descriptions of :

- the Host Application Program Interface (API)
- the FILU Run Time Library (RTL)
- the FILU instructions and conventions
- some application examples.

An appendix contains a detailed description of each of the above.

A Bit Exact C-Model of the FILU is available that supports both the API and the FILU *run time library* and can be used to simulate the operation of the FILU.

1.1.1 General Description

The FILU acts as a DSP coprocessor to a Host processor, for example a micro-controller or RISC. The Host calls DSP functions which are executed by the FILU. Communication between the HOST and the FILU is via a shared RAM (FILU RAM). All programming information, data pointers and input / output data is communicated to the FILU via the shared RAM. Figure 1.1 shows a block diagram of the FILU, shared RAM, Host processor and Host RAM. The FILU RAM is memory mapped into the Host address space.

While the FILU is executing a DSP function, i.e. is busy it sets a BUSY bit in the RAM and it has control of the RAM. The HOST must initiate a handshake to read or write the FILU RAM. It can do this at any time even when the FILU is ‘BUSY. The FILU operation is stalled during HOST read/ write operations.

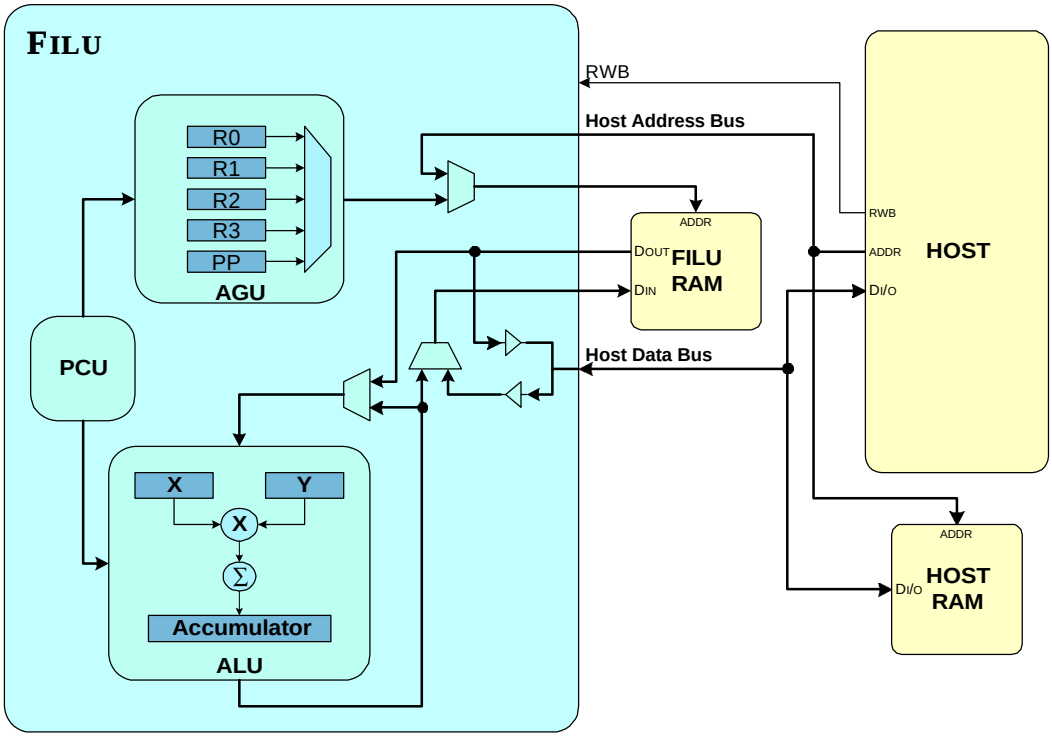

Figure 1.1 **FILU** *block diagram and Host Interface.*

1.2 Software Interface

This software interface between the HOST and the FILU DSP coprocessor is in two parts. The first part is the Host API which allows the Host to control the FILU. The API functions are invoked using standard C function calls and they allow the HOST to:

- initialise the FILU.
- read data from the FILU.
- write data to the FILU.
- load function parameters for the FILU DSP functions.
- call FILU DSP functions using C function calls.
- poll the FILU operating status.

The second part of the software interface is the *run time library* which is the set of DSP functions which are stored in ROM and can be executed by the FILU. These include:

- an FIR filter.
- a first order IIR filter.
- a second order IIR filter.
- an N point in-place FFT where N is a radix-2 number and N ≤ 256.
- a correlation function.

- a Taylor series.

1.2.1 HOST API Functions

The HOST API functions are tabulated below. The details of these functions together with the formal parameter lists and return values are given in FILU API

Function Name	Description
ResetFILU	Initialises the FILU.
StartFILU	Calls a FILU DSP function.
ReadFILU	Reads data from the FILU RAM.
WriteFILU	Writes data to the FILU RAM.
CheckFILUStatus	Determines the operating status of the FILU.
LoadFILUParameters	Loads the DSP function parameters into the FILU RAM.

Table 1.1 *FILU Application Program Interface Functions.*

1.2.2 FILU Run Time Library

The FILU has a number of DSP ROM functions which are used as a *run time library* and called using C function calls. The functions are tabulated below. All these functions are included in the C-Model. These functions are called by the HOST using the API and execute in the FILU. Details of the exact functionality of each of these functions are given in FILU Run Time Library.

Function	Parameter1	Parameter 2	Parameter 3	Parameter 4
CORR	X Address	Y Address	Data Length	Output Address
FIR	Input Data Address	Data length	Output Address	Coefficient Start Address
IIR_1	Input Data Address	Data Length	Output Address	Coefficient Start Address
IIR_2	Input Data Address	Data length	Output Address	Coefficient Start Address
FFT	Real Data Address	Imaginary Data Address	$\log_2 N$	N
PowerSeries	X	Coefficients	Result	-

Table 1.2 *FILU Run Time Library Functions.*

1.3 Calling a FILU Run Time Library Function

FILU functions are called using a standard C function call. The FILU API is used to pass the FILU function name (effectively a pointer to the FILU function) and the list of arguments to the FILU. The API function StartFILU is used to call the FILU FIR function as follows:

```
StartFILU(filu.FIR, data_in, N, data_out, coefficients);
```

or to call the FILU FFT function the function call is:

```
StartFILU(filu.FFT,  real_data, imaginary_data, number_of_stages, length);
```

The function StartFILU has a variable parameter list. The first parameter is always a pointer to the function. In the situation where a run time library function is called repeatedly there is no need to reload the parameters every time. The StartFILU function can be a single argument - the function pointer and is called as follows:

```
StartFILU(filu.FIR);
```

In summary, the API function StartFILU can invoke a FILU run time library function with a list of function parameters or it can simple pass the function pointer if the parameters are already in place from a previous call.

1.3.1 Cascading FILU Functions

Many applications will call a number of FILU ROM functions in sequence. These ROM function calls can be cascaded into a single function call to the FILU. This is a FILU RAM function, as the constituent ROM function calls are programmed in the FILU RAM. All of the parameters for this RAM function are passed in one go to the FILU. The HOST can poll the FILU to determine when the RAM function is complete.

A example RAM function is shown in Figure 1.2.

```
void RAM_function() {

        filu.FIR();
        filu.FFT();
        filu.CORR();

return;
}
```

Figure 1.2 *Example RAM function.*

In this example the API StartFILU function calls will be as follows:

```
StartFILU(RAM_function,"%x %x %x %x %x %x %x %x %x %x %x %x",
FIR_INPUT_DATA_ADDRESS, FIR_DATA_LENGTH, FIR_OUTPUT_ADDRESS,
FIR_COEFFICIENT_ADDRESS, FFT_REAL_DATA_ADDRESS, FFT_IMAGINARY_DATA_ADDRESS,
FFT_LOG2N, FFT_N, CORR_X_ADDRESS, CORR_Y_ADDRESS, CORR_DATA_LENGTH,
CORR_OUTPUT_ADDRESS);
```

1.3.2 Using FILU Functions

The general procedure for using a FILU ROM or RAM function is:

1. Reset the FILU using the API function *ResetFILU().*
2. Load the input data / coefficients into the FILU RAM using the API function *WriteFILU().*
3. Call the FILU RAM function using the API function *StartFILU().*
4. Continue other Host processing in parallel with FILU operation
5. Poll the FILU to determine when the function has finished using the API function *CheckFILUStatus()*.
6. Read the results of the computation using the API function *ReadFILU().*
7. Repeat steps 2 to 6.

1.4 Application Examples

The following two simple examples illustrate how the Host can use the FILU to implement simple DSP functions. The first example calls a single FILU run time library function while the second example shows how FILU run time library functions are cascaded.

1.4.1 Example of Single ROM Function call - FIR Filter

The FILU can perform a FIR filtering operation on a buffer of data. Where adequate memory is available the entire record may be filtered in one pass. In a single pass filtering operation the programmer need only ensure that the filter memory is zero i.e. for a filter of order *p* the first *p* memory locations should be zeroed.

Where a very long record must be filtered and inadequate memory is available the record must be filtered in blocks. The FILU automatically adjusts the filter memory on subsequent passes of the filter so that no action is required on the part of the programmer. The method is illustrated in Figure 1.3.

This example illustrates how the FILU can filter a data record of length 5120 in blocks of 512. It is assumed that the HOST input and output buffers are provided separately.

```
  // define the coefficients for the FIR filter
#define NO_BLOCKS 10        // number of blocks of data
short coefficients[] = {FILTER_LENGTH, 127, -2125, -5546, -3095, 5444, 10415,
                5444, -3095, -5546, -2125, 127};


void  FirFilter(short *A_D_Buffer, short *Host_Buffer) {

  short j;


       // reset the FILU
      ResetFILU();

       // load the coefficients into the FILU memory
      WriteFILU(coefficients, FIR_COEFFICIENT_ADDRESS, FILTER_LENGTH+1);

       // load the first data block into the FILU data memory
      WriteFILU(A_D_Buffer, FIR_DATA_ADDRESS, DATA_RECORD_LENGTH);

          // filter the data
      for (j=0; j<NO_BLOCKS; j++) {

          // call the FIR function
        StartFILU(filu.FIR,"%x%x%x%x", FIR_DATA_ADDRESS, DATA_RECORD_LENGTH,
                  FIR_OUTPUT_ADDRESS, FIR_COEFFICIENT_ADDRESS);

        //
        //
        //  Other Host processing can go here
        //
        //

          // Check has FILU finished processing block
        while (CheckFILUStatus());

          //  read data from FILU to HOST memory
        ReadFILU(FIR_OUTPUT_ADDRESS, Host_Buffer+j*DATA_RECORD_LENGTH,
                 DATA_RECORD_LENGTH);

          // write the NEXT block of data to FILU from HOST memory
        WriteFILU(A_D_Buffer+(j+1)*DATA_RECORD_LENGTH, FIR_DATA_ADDRESS,
                  DATA_RECORD_LENGTH);

      }
return;
}
```

Figure 1.3 *FIR filter example.*

1.4.2 Example of Cascade of ROM Functions - Knock Detection

Engine knock is a phenomenon where the fuel-air mixture in an internal combustion engine is detonated too soon due to poor quality fuel with catastrophic consequences for the engine. Knock can be detected by an energy detection method. A data stream from an accelerometer mounted on the engine block is filtered using a FIR filter and frequency content determined using an FFT. Finally, the energy content in a number of frequency ranges is determined by a sum-of-squares procedure. The FILU is ideally suited to an application like this.

The knock detection algorithm running on the HOST filters a data stream from an analog-digital (A-D) converter in blocks of length 256, performs an FFT on the filter output and then calculates the sum-of-

squares of the FFT output returning a double precision result to the HOST. This double precision result is used as a knock index. The HOST maintains the A-D buffer.

The above sequences of function calls can be implemented as a RAM function which is implemented as follows.

```
void  KnockDetect()  {

    filu.FIR();         // call the FIR function
    filu.FFT();         // call the FFT function
    filu.CORR();        // call the Correlation function

return;
}
```

Figure 1.4 *Example RAM function for Knock detection.*

All the parameters are passed in one go by the API. Hence the Knock detect function is called as follows.

```
StartFILU(KnockDetect,"%x %x %x %x %x %x %x %x %x %x %x %x", FIR_DATA_ADDRESS,
RECORD_LENGTH,FIR_OUTPUT_ADDRESS,FIR_COEFFICIENT_ADDRESS, FIR_OUTPUT_ADDRESS,
FIR_OUTPUT_ADDRESS+256, 8, 256, FIR_OUTPUT_ADDRESS, FIR_OUTPUT_ADDRESS, 256,
CORR_OUTPUT_ADDRESS);
```

The above pointers have been chosen so that the output of the FIR function provides the input to the FFT function which in turn provides the input to the CORR function.

```
/*
 *
 *  Function Knock_Init()
 *
 *  Initialises the FILu prior to calling knock detection algorithm.
 *
 */

void  Knock_Init() {

    // reset the FILU
  ResetFILU();

    // load the FIR filter coefficients into the FILU memory
  WriteFILU(coefficients, FIR_COEFFICIENT_ADDRESS, FILTER_LENGTH+1);

  return;
}


/*
 *
 *  Function Knock_Process()
 *
 *  Implements the Knock Detection Algorithm.
 *
 */


void Knock_Process(short *A_D_Buffer, short * Host_Buffer, short *Knock_Result) {


    // load the A/D buffer into the FILU data memory
  WriteFILU(A_D_Buffer, FIR_DATA_ADDRESS, 256);

    // start knock detection algorithm
  StartFILU(KnockDetect,"%x %x %x %x %x %x %x %x %x %x %x %x",
            FIR_DATA_ADDRESS, 256, FIR_OUTPUT_ADDRESS,
            FIR_COEFFICIENT_ADDRESS, FIR_OUTPUT_ADDRESS,
            FIR_OUTPUT_ADDRESS+256, 8, 256, FIR_OUTPUT_ADDRESS,
            FIR_OUTPUT_ADDRESS, 256, CORR_OUTPUT_ADDRESS);

    //
    //
    //  Other Host processing can go here
    //
    //

    // Check has FILU finished processing block
  while(CheckFILUStatus());

    //  read data from FILU to HOST memory
  ReadFILU(CORR_OUTPUT_ADDRESS, Knock_Result, 3);

return;
}
```

Figure 1.5 *Knock detection example.*

Appendix A: FILU API

A.1 HOST API

The Application Program interface is a set of C functions used to control the FILU i.e. call a FILU ROM function, write data to the FILU, read results from the FILU, etc.

The following are the details of the API functions.

	ResetFILU
Description	Initialises the FILU.
Function Call	**short * ResetFILU()**
Parameters	None
Remarks	Resets the FILU to a known condition. The Program Counter is cleared, BUSY bit in the status register is cleared.
Returns	No return value.

	CheckFILUStatus
Description	Determines the FILU status.
Function Call	**short CheckFILUStatus()**
Parameters	None
Remarks	Determines the FILU operating status i.e. BUSY or IDLE. This function polls the BUSY bit. If BUSY = 1 then the FILU is executing a run time library function or user defined function. If BUSY = 0 the FILU has completed a function.
Returns	FILU status, 1 = BUSY, 0 = IDLE.

	LoadFILUParameters
Description	Loads a function parameter list into the FILU parameter space
Function Call	**void LoadFILUParameters(short *parameter_address, …)**
Parameters	Variable list of parameters corresponding to FILU function parameters. At least 1 parameter must be provided.
Remarks	Loads a list of parameters into the FILU parameter space. Pointers passed are converted into offsets from the base address of the FILU RAM. The list of parameters is variable.
Returns	no return value

	StartFILU
Description	Calls a FILU Run Time Library function or user function.
Function Call	**void StartFILU (short *function_pointer, const char *format_string,...)**
Parameters	This function passes a pointer to a run time library function and a list of parameters for a run time library function to the FILU. The list of parameters is variable and the types of the parameters are defined in *format_string*. The format specifiers are the same as those used in the C run time library function *scanf()* but are limited to the types x, d, o and i. This function can be invoked with a single argument which must be a pointer to a function. In this case the parameters must be in place prior to the call.
Remarks	Loads the function pointer and the list of parameters into the FILU. The FILU function is started. The BUSY flag is set when the function is started and cleared when the function exits.
Returns	no return value

	ReadFILU
Description	Reads data from the FILU memory.
Function Call	**void ReadFILU (short filu_offset, short *host_memory, short N)**
Parameters	Copies N data points from the FILU RAM to the HOST RAM.
Remarks	The shared RAM handshaking is also implemented in this function.
Returns	no return value

	WriteFILU
Description	Writes data to the FILU memory.
Function Call	**void WriteFILU (short *host_memory, short filu_offset, short N)**
Parameters	Copies N data points from the HOST RAM to the FILU RAM.
Remarks	The shared RAM handshaking is also implemented in this function.
Returns	no return value

Appendix B: FILU Run Time Library

The *run time library* is a suite of ROM based DSP functions. The operation of the *run time library* functions is given below. The library can be extended by the user. Details are given later.

B.1 CORR Function

The FILU can compute a cross-correlation of two data records, X and Y according to (B.1) where N is the correlation length. The cross-correlation can be of arbitrary length. An auto-correlation function can be computed by setting the X and Y address pointers to the same location. Note that the full 40 bit accumulator is written to memory in three successive write operations. The least significant word is written first.

$$r_{xy} = \sum_{i=0}^{N-1} x_i y_i \qquad \text{(B.1)}$$

The Correlation Command requires 4 parameters:

1. X Address : This is the address in the FILU RAM of record X.
2. Y Address : This is the address in the FILU RAM of record Y.
3. Data Length N: This is the correlation length.
4. Output Address : This is the address in the FILU RAM of the result.

B.2 FIR Function

The FIR filter command can implement a FIR filter of arbitrary order according to (B.2) where p is the order of the filter, i.e. there are p+1 coefficients. The number of coefficients must be at least 2.

$$y_n = \sum_{k=0}^{p} a_k x_{n-k} \qquad \text{for } 0 \le n \le N-1 \qquad \text{(B.2)}$$

The FIR filter command requires 4 parameters:

1. Input Data Address. This is the address in the FILU where the input data to the filter and filter memory are stored.
2. Data Length N. This is the number of data points to filter.
3. Output Address. The address where the filtered data is stored.
4. Coefficient Start Address. This is the address in the FILU RAM where the FIR Filter coefficients are stored. The first element in the list must be the number of filter coefficients. The coefficients are stored in the order $\{p+1,\ a_0,\ a_1,\ \ldots\ a_p\}$ where p is the number of coefficients and a_k are the coefficients.

The first time the function is called the user should ensure that the first *p* points of the data record correspond to the filter memory, i.e. $x_{-p} \rightarrow x_{-1}$. These are the *p* memory locations prior to the input data address as shown in Figure B.1. At the end of the function the last *p* locations of the data record are copied down to the *p* locations prior to the input data address. This ensures that this function can be used to process an array of data larger than the available FILU RAM by filtering it in blocks of data without the user having to manage the filter memory.

The diagram below shows the arrangement in RAM.

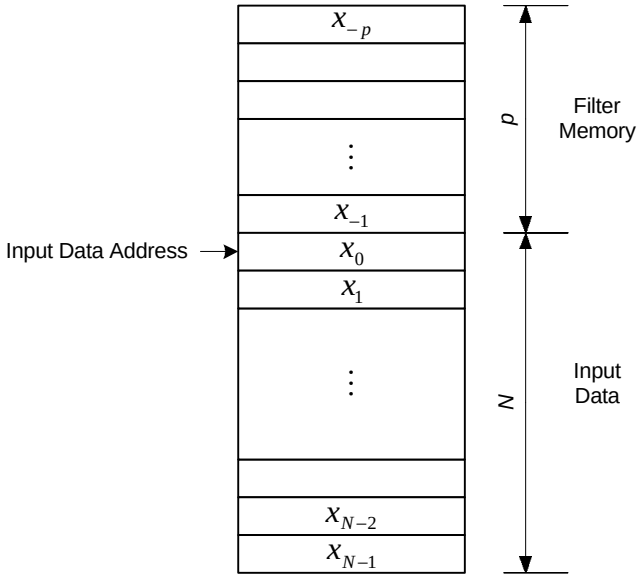

Figure B.1 *Organisation of filter memory in RAM for FIR function.*

B.3 IIR_1 Function

The IIR_1 implements a first order IIR filter according to (B.3) . The IIR_1 filter has two coefficients, i.e. a feed-forward and a feedback coefficient.

$$y_n = bx_n - ay_{n-1} \qquad \text{for } 0 \le n \le N-1 \tag{B.3}$$

The IIR filter command requires 4 parameters:

1. Input Data Address. This is the address in the FILU RAM where the input data to the filter is stored.
2. Data Length. This is the number of data points to filter.
3. Output Address. The address where the filtered data is stored.
4. Coefficient Start Address. This is the address in the FILU RAM where the IIR Filter coefficients are stored. The coefficients are stored in the order $\{b, a\}$.

The first time the function is called the user should ensure that the memory location prior to the output data address corresponds to the output filter memory, i.e. y_{-1}. At the end of the function the last location of output data record is copied down to the memory location prior to the output data address. This ensures that this function can be used to process an array of data larger than the available FILU RAM by filtering it in blocks of data without the user having to manage the filter memory.

The diagram below shows the arrangement in RAM.

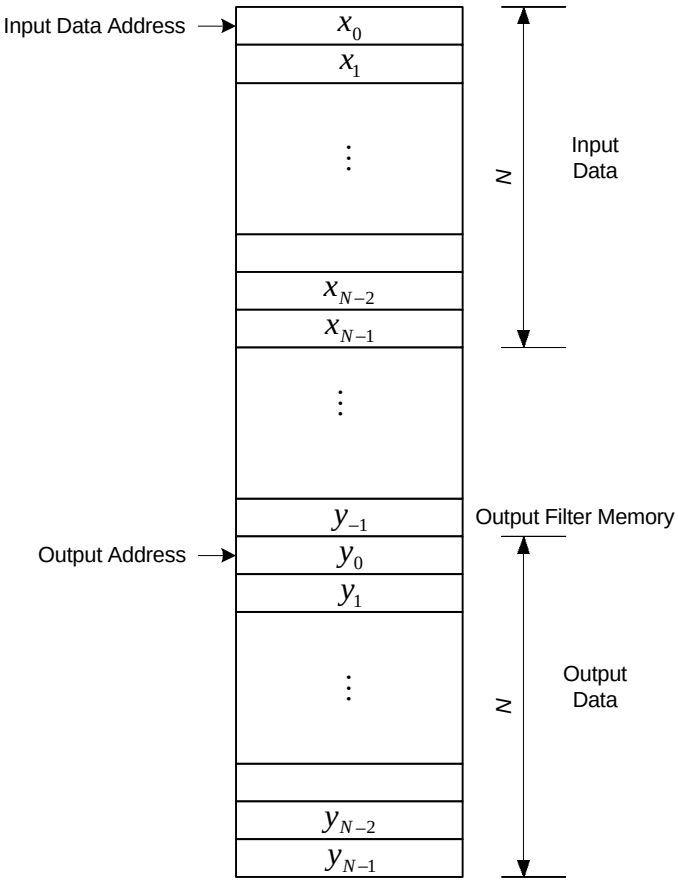

Figure B.2 *Organisation of filter memory in RAM for IIR_1 function.*

B.4 IIR_2 Function

The IIR_2 function implements a second order IIR filter according to (2.4). The IIR_1 filter has two coefficients, i.e. a feed-forward and a feedback coefficient.

$$y_n = b_0 x_n + b_1 x_{n-1} - a_1 y_{n-1} - a_2 y_{n-2} \qquad \text{for } 0 \le n \le N-1 \tag{B.4}$$

The IIR_2 filter command requires 4 parameters:

1. Input Data Address. This is the address in the FILU RAM where the input data to the filter is stored.
2. Data Length. This is the number of data points to filter.
3. Output Address. The address where the filtered data is stored.
4. Coefficient Start Address. This is the address in the FILU RAM where the filter coefficients are stored. The coefficients are stored in the order $\{b_0, b_1, a_1, a_2\}$.

The first time the function is called the user should ensure that the memory location prior to the input data address corresponds to the input filter memory, i.e. x_{-1} and that the two memory

locations prior to the output data address correspond to the output filter memory, i.e. y_{-2} and y_{-1}. At the end of the function the last location of the input data record is copied down to memory location prior to the input data address and the last 2 locations of the output data record are copied down to the two memory locations prior to the output data address. This ensures that this function can be used to process an array of data larger than the available FILU RAM by filtering it in blocks of data without the user having to manage the filter memory. The diagram below shows the arrangement in RAM.

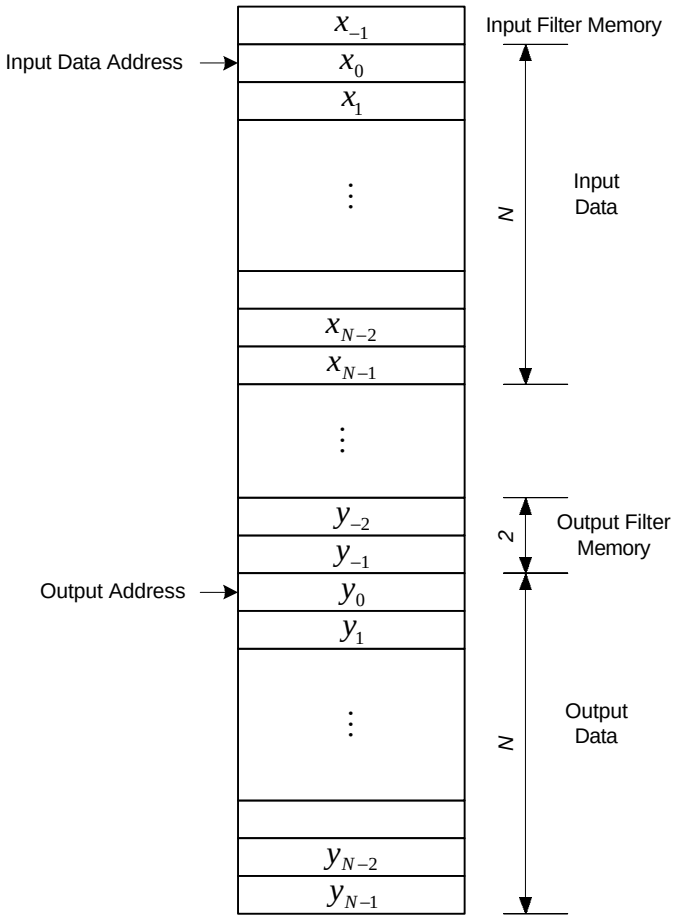

Figure B.3 *Organisation of filter memory in RAM for IIR_2 function.*

B.5 FFT Function

The FILU executes an N point, complex, in-place FFT according to (B.5) . The length of the FFT is limited to 256 complex points. Where only a real FFT is needed the imaginary data record should be zeroed. The start addresses of the real and imaginary data sequences must be provided as parameters to this function.

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn} \quad \text{for } 0 \le k \le N-1 \quad \text{where} \quad W = e^{j2\pi k/N} \tag{B.5}$$

The parameter list is as follows:

1. Real Data Address. The address at which the real input data is stored.

2. Imaginary Data Address. The address where the imaginary input data is stored. For a real FFT the imaginary data is zeroed.
3. The number of stages in the FFT i.e. $\log_2 N$.
4. The length of the FFT, N. N must be a radix-2 number.

B.6 PowerSeries

The PowerSeries function implements a generalized N[th]-order real power series.

$$\begin{aligned} y &= \sum_{n=0}^{N} a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots))) \\ &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots \end{aligned}$$

A 2^{M-1}-scaled version is output by the function.

$$\hat{y} = 2^{M-1} y, \qquad M \geq 1$$

The input x is passed directly in the parameter list.

The number of coefficients less 1 (N+1-1 = N) of the series to be computed is passed as the first element of the coefficient list.

The scale factor M is passed as the second element of the coefficients' file. When M = 1, the scale factor is 1, when M = 4, the scale factor is 8, etc.

Hence the coefficient list should be $\{N, M, a_n, a_{n-1}, \ldots, a_0\}$.

The parameter list is as follows:

1. Input x. This is the input.
2. Coefficient Start Address. This is the address in the FILU RAM where the power series coefficients are stored. The first element of the list must be the number of coefficients less 1. The second element of the list must be the scale factor.
3. Result address. The address where the result is stored.

Appendix C: Installing The FILU C-Model and API

C.1 Installing The FILU C-Model

The C- Model requires Microsoft Developer Studio 97™, Visual C++ V5.0 (or V6.0) to run. The model is provided as a .zip file which contains a series of header files and a library file.

The installation procedure is as follows:

1. Unzip the files using WinZip.
2. Install MicroSoft Development Studio V5.0 (or V6.0)
3. Start a new project in MSD Studio. You should choose a WIN32 Console Application.
4. Add the header files and the library file to your project directory.
5. Include the list of header files in your source code files.
6. Add the library file 'filuModel.lib' to the list of Source Files in your project workspace.
7. Write your own program to call the FILU functions.
8. Compile, Link and Run

The list of header files provided is as follows:

1. defines.h
2. accum.h
3. addr.h
4. macreg.h
5. ctrreg.h
6. filu.h
7. external.h

These header files should be included in any of your application programs in the order shown above.

C code for the application examples are included with the release.

Appendix D: Extending the FILU Run Time Library

The FILU C-Model can be extended to include new run time library functions. The procedure is:

1. Write the new run time library function.
2. Add this function to the model.

D.1 Writing a new Run time Library Function

Consider the run time library function CORR as an example. This function is written as follows:

```
void FILU::CORR() {

    R0 = *PP++;         // Load X data pointer
    R1 = *PP++;         // Load y data pointer
    D0 = *PP++;         // Load correlation width
    R2 = *PP++;         // load output data pointer

    A  = 0L;            // clear A
    X  = *R0++;         // load X data point
    Y  = *R1++;         // load Y data point

       do {
          A = A + X*Y; // multiply- accumulate
          X = *R0++;
          Y = *R1++;
                     }
       while (D0--);

    *R2++ = A.A0;       // save LSP
    *R2++ = A.A1;       // save MSP
    *R2++ = A.A2;       // save XP

return;
}
```

Figure D.1 *ROM function CORR.*

All the operations are standard C/C++ syntax. All new functions are defined as member functions of the class **FILU. There are some points to note about writing a new function:**

1. FILU functions will in general require a list of parameters. The parameters are written to the FILU RAM and the ROM function must then load them from RAM. When the list and order of parameters is known the programmer must then load the FILU pointer and counter registers from the parameter list. This is done via the Parameter Pointer Register (PP)**.** The Parameter Pointer will point to the first parameter in the list at entry to the function**.** In the example above the Correlation function requires four parameters as detailed in B.1.
2. Standard C statements can be used to implement control loops but there are some restrictions as detailed in C Statements

D.1.1 Adding the new function to C- Model

To add the new run time library function to the model:

1. Add the function declaration, in this case **void CORR(),** to the member function declaration section of the class FILU in the header file <filu.h>
2. Add the new function to your project.
3. Compile and link.

D.2 Adding RAM Functions

RAM functions can be added to the model and executed by the FILU. A user defined RAM functions can call FILU run time library functions in cascade without intervention by the HOST. The FILU API passes all the parameters for the cascade of functions in a single call to the user defined RAM based function**.** A example RAM function is shown in Figure D.2.

```
void RAM_function() {

        filu.FIR();
        filu.FFT();
        filu.CORR();

return;
}
```

Figure D.2. *Example RAM function.*

In this example the API StartFILU function calls will be as follows:

```
StartFILU(RAM_function,"%x %x %x %x %x %x %x %x %x %x %x %x",
FIR_INPUT_DATA_ADDRESS, FIR_DATA_LENGTH, FIR_OUTPUT_ADDRESS,
FIR_COEFFICIENT_ADDRESS, FFT_REAL_DATA_ADDRESS, FFT_IMAGINARY_DATA_ADDRESS,
FFT_LOG2N, FFT_N, CORR_X_ADDRESS, CORR_Y_ADDRESS, CORR_DATA_LENGTH,
CORR_OUTPUT_ADDRESS);
```

Appendix E: FILU Instructions

E.1 Register Set

The FILU register set consists of:

- a single 40 bit Accumulator A which can also be viewed as an 8 bit register A.A2 and 16 bit registers A.A1, A.A0.
- 2 X 16 bit MAC Input Registers X, Y.
- 4 X 12 bit Address Registers, R0, R1, R2, R3.
- 1 X 12 bit Index Register N.
- 1 X 12 bit parameter pointer register PP (effectively an address register).
- 3 X 12 bit loop Counter Registers, D0, D1, D2.

The organisation of the 40 bit Accumulator is shown below. The Accumulator register consists of three sections:

1. A.A0 = least significant fractional portion (16 bits)
2. A.A1 = most significant fractional portion (16 bits)
3. A.A2 = an 8 bit extension register

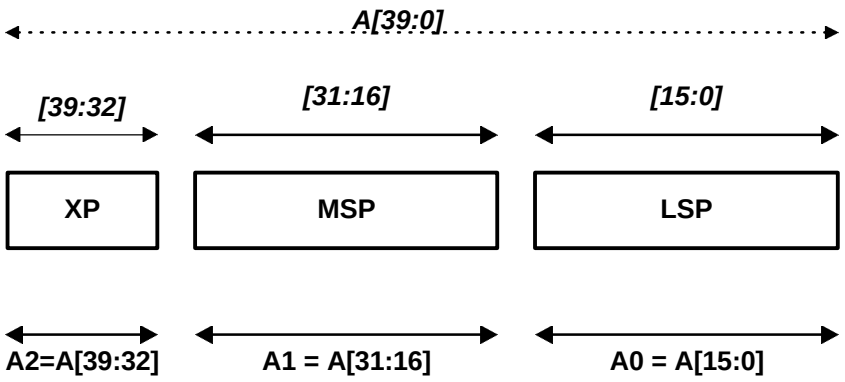

Figure E.1 *Organisation of the Accumulator register.*

E.2 Addressing Modes

The addressing modes available are:

- register direct.
- register-indirect with or without post-increment or decrement.
- register-indexed, i.e. post increment/decrement by register N.
- immediate addressing limited to A = 0.

E.3 Instruction Set

E.3.1 Arithmetic Instructions

Arithmetic instructions operate on the MAC input registers X and Y and store the result in Accumulator register A**.** All operations are 2'compliment using fractional 1.15 arithmetic**.**

The allowable arithmetic instructions are:

Instruction	Operation
`A = X.Y`	Multiply.
`A = - X.Y`	Multiply Minus.
`A = X*Y + RND`	Multiply with rounding.
`A = -X*Y + RND`	Multiply Minus with rounding.
`A = A + X*Y`	Multiply & Accumulate.
`A = A - X*Y`	Multiply & Minus Accumulation.
`A = A + X*Y + RND`	Multiply & Accumulate with rounding.
`A = A - X*Y + RND`	Multiply & Minus Accumulation with rounding.
`A = A + RND`	Round Accumulator
`A = A + Y`	Add Y to Accumulator (add 16 bit Y to A[39:16]).
`A = A - Y`	Subtract Y from Accumulator (sub 16 bit Y from A[39:16])**.**
`A << 1L`	Arithmetic shift A left 1 bit with, A[0] = 0.
`A >> 1L`	Arithmetic shift A right 1 bit, A[39] = A[38].
`A = 0L`	Clear accumulator.
`A = Y`	Load A with Y. A[31:16] = Y, A[39:32] = Y[15], A[15:0] = 0
`A = - Y`	A = Minus Y**.** A[31:16] = -Y, A[39:32] = ~Y[15], A[15:0] = 0
`A = neg(A)`	Negation of A. A[39:16] = -1*A[39:16], A[15:0] = 0
`A = abs(A)`	Absolute value of A. A[39:16] = abs(A[39:16]), A[15:0] = 0
A = A/Y	Divide primitive**.** See E.3.3 Division.

Table E.1 *Arithmetic Instructions.*

Note that the instructions `A = 0L`, `A << 1L` and `A >> 1L` must include the string literal suffix “L” to denote that the constant is a long value**.** This must be done to maintain compatibility with C**.** A warning is issued if a shift value is greater than one and no shifting will take place.

E.3.1.1 Rounding

Rounding to 16 bits is implemented using convergent rounding**.** This is achieved by adding the RND bit to the Accumulator. For example to round the result of a multiply & accumulate operation the syntax is:

```
A = A + X*Y + RND
```

E.3.2 Move Instructions

E.3.2.1 Register Direct

Instruction	Operation
[Rn, Dn, N] = [Rm[1], Dm, N]	Move Address Register Rm to Address Register Rn

Table E.2 *Register Direct Move Instructions.*

Register direct moves are only supported between address registers.

E.3.2.2 Register Indirect

Instruction	Operation
[X, Y, A] = *Rn	Register-indirect (write register).
[X, Y, A] = *Rn++	Register-indirect post increment (write register).
[X, Y, A] = *Rn--	Register-indirect post decrement (write register).
[X, Y, A] = *(Rn+=N)	Register-indexed post increment (write register).
[X, Y, A] = *(Rn-=N)	Register-indexed post decrement (write register).
*Rn = [A, A.A2, A.A1, A.A0]	Register-indirect (read register).
*Rn++ = [A, A.A2, A.A1, A.A0]	Register-indirect post increment (read register).
*Rn-- = [A, A.A2, A.A1, A.A0]	Register-indirect post decrement (read register).
*(Rn+=N) = [A, A.A2, A.A1, A.A0]	Register-indexed post increment (read register).
*(Rn-=N) = [A, A.A2, A.A1, A.A0]	Register-indexed post decrement (read register).
Rn++	Increment Address Register
Rn--	Decrement Address Register
Rn+=N	Add Index Register to Address Register
Rn-=N	Subtract Index Register from Address Register
Dn--	Decrement Counter Register

Table E.3 *Register indirect addressing move instructions.*

The accumulator register A can be read or written. The accumulator registers A.A2, A.A1 & A.A0 can only be read. The X and Y registers can only be written.

When the accumulator register A is written (from RAM or by via A = Y) the 16 bit number is sign extended and zero filled to 40 bits, i.e. A.A0 is set to all zeros and A.A2 is set to the sign bit.

[1] Rn, Rm is one of the address registers R0, R1, R2, R3

E.3.2.3 Saturation

The FILU supports saturation of the Accumulator output on the 32 bit boundary when writing the Accumulator to memory**.** When the Accumulator is saved to memory i.e. in a write operations like:

```
*Rn++ = A;
```

the contents of the Accumulator are checked for overflow beyond bit 31 and if overflow is detected then a 16 bit saturated value i.e. the largest representible positive 16 bit integer or the smallest representible negative 16 bit integer is written to memory. The contents of the Accumulator are unchanged.

Where the register A.A1 is cited explicitly in the write instruction such as in the instruction:

```
*Rn = A.A1
```

no saturation takes place.

E.3.3 Division.

A non-restoring division algorithm is implemented in the FILU. The division operation produces a single quotient bit per cycle. The divisor must be in the Y register with the 32 bit dividend in A. The quotient is returned in A0 and the 32 bit remainder (least significant 16 bits of) in A1. The syntax is:

```
A = A/Y
```

or, equivalently

```
A /= Y.
```

Sixteen cycles are required to produce a full precision result. A full 16 bit division can be implemented as follows:

```
do {
   A = A/Y;
}while(D0-- > 1);
```

E.3.3.1 Divide Errors.

Non restoring division algorithm gives rise to some errors when the divisor is negative which the programmer must correct.

In signed division where the divisor is negative the quotient will usually be 1 LSB less than the true result; the only exception is the case where the quotient is 0x8000 when it will be correct. This error can only be avoided by first taking the absolute value of the signed divisor**.** No compensation is provided in the FILU for this error; it has to be handled explicitly by the programmer.

E.3.3.2 Divide Limitations.

The division algorithm is limited in the following ways:

1. The FILU handles only two complement numbers.
2. In fractional arithmetic the absolute value of the numerator must be less than the absolute value of the denominator in order to return a fractional quotient**.** Otherwise it may not be clear where the binary point is located after the division.

E.3.4 Status Register

The status register stores information about the accumulator, which is used for certain ALU functions. The bits defined in this register are

1. **Z: Zero**. This bit is set high if the contents of the upper 24 bits of the accumulator (A2:A1) are zero. It is used internally by the DIV iteration.
2. **N: Negative**. This bit is set if the most significant bit of the accumulator is set, else it is cleared.
3. **C: Carry**. This bit is the carry out of the accumulator.
4. **V: Overflow**. This bit is set if there is an overflow in the 40-bit result, it is cleared otherwise.
5. **L: Limit**. This bit is set if the overflow bit is set. It is cleared when the Busy bit is set, i.e. when the FILU starts processing.
6. **E: Extension**. This bit is cleared if the Carry bit, A2 and the MSB of A1 are all zeros or all ones.

The values of the status bits reflect the current state of the accumulator. The status **bits are updated with a data move operation** into the accumulator. Some DSPs only update the status bits on arithmetic or logical operations.

The status bits can be read by the Host at RAM address zero. The status bits are read only.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	*r*	*r*	E	L	V	C	N	Z	*r*	*r*	*r*	*r*	*r*	R	M	B

Table E.4 *Location of status bits in RAM address zero.*

Bits marked *r* are reserved. The bits R, M & B are Reset, Master and Busy control bits for the Host interface.

Appendix F: C Statements

Standard C statements can be used in FILU programs but there are a number of restrictions. The allowable statements are:

1. **do** <*statement*> **while** (*condition true*);
2. **while** (*condition true*) <*statement*>;
3. **return** ;

The loop control parameters are limited to a subset of the FILU register set.

F.1 The do…while statement

There are restrictions on the use of the **do..while** statement in that only the counter registers D0, D1 and D2 can be used in evaluating the loop conditions. An example of a **do…while** loop used in the FILU is given below:

```
do {
       A = A + X*Y;  X = *R0++;  Y = *R1++;
       }
    while  (D0-- > 1);
```

Note that the FILU implements loops exactly in accordance with the C syntax above. The loop counter D0 is tested and the condition evaluated following which evaluation the counter is decremented; so it is a true post-decrement operation.

F.2 The while statement

There are restrictions on the use of the **while** statement in that only the counter registers D0, D1 and D2 can be used in evaluating the loop conditions. An example of a **while** loop used in the FILU is given below:

```
while  (D0-- > 1)
       {
       A = A + X*Y;  X = *R0++;  Y = *R1++;
       }
```

F.3 The return statement

The ***return*** statement is used to terminate a function. The return is always of type void.

SUNSTAR 商斯达实业集团是集研发、生产、工程、销售、代理经销 、技术咨询、信息服务等为一体的高科技企业，是专业高科技电子产品生产厂家，是具有 10 多年历史的专业电子元器件供应商，是中国最早和最大的仓储式连锁规模经营大型综合电子零部件代理分销商之一,是一家专业代理和分銷世界各大品牌 IC 芯片和電子元器件的连锁经营綜合性国际公司，专业经营进口、国产名厂名牌电子元件，型号、种类齐全。在香港、北京、深圳、上海、西安、成都等全国主要电子市场设有直属分公司和产品展示展销窗口门市部专卖店及代理分销商,已在全国范围内建成强大统一的供货和代理分销网络。 我们专业代理经销、开发生产电子元器件、集成电路、传感器、微波光电元器件、工控机/DOC/DOM 电子盘、专用电路、单片机开发、MCU/DSP/ARM/FPGA 软件硬件、二极管、三极管、模块等，是您可靠的一站式现货配套供应商、方案提供商、部件功能模块开发配套商。商斯达实业公司拥有庞大的资料库，有数位毕业于著名高校——有中国电子工业摇篮之称的西安电子科技大学（西军电）并长期从事国防尖端科技研究的高级工程师为您精挑细选、量身订做各种高科技电子元器件，并解决各种技术问题。

微波光电部专业代理经销高频、微波、光纤、光电元器件、组件、部件、模块、整机；电磁兼容元器件、材料、设备；微波 CAD、EDA 软件、开发测试仿真工具；微波、光纤仪器仪表。欢迎国外高科技微波、光纤厂商将优秀产品介绍到中国、共同开拓市场。长期大量现货专业批发高频、微波、卫星、光纤、电视、CATV 器件： 晶振、VCO、连接器、PIN 开关、变容二极管、开关二极管、低噪晶体管、功率电阻及电容、放大器、功率管、MMIC、混频器、耦合器、功分器、振荡器、合成器、衰减器、滤波器、隔离器、环行器、移相器、调制解调器；光电子元器件和组件：红外发射管、红外接收管、光电开关、光敏管、发光二极管和发光二极管组件、半导体激光二极管和激光器组件、光电探测器和光接收组件、光发射接收模块、光纤激光器和光放大器、光调制器、光开关、DWDM 用光发射和接收器件、用户接入系统光光收发器件与模块、光纤连接器、光纤跳线/尾纤、光衰减器、光纤适 配器、光隔离器、光耦合器、光环行器、光复用器/转换器；无线收发芯片和模组、蓝牙芯片和模组。

更多产品请看本公司产品专用销售网站:

商斯达微波光电产品网:HTTP://www.rfoe.net/

商斯达中国传感器科技信息网：http://www.sensor-ic.com/

商斯达工控安防网：http://www.pc-ps.net/

商斯达电子元器件网：http://www.sunstare.com/

商斯达消费电子产品网://www.icasic.com/

商斯达实业科技产品网://www.sunstars.cn/ *射频微波光电元器件销售热线:*

地址：深圳市福田区福华路福庆街鸿图大厦 1602 室

电话：0755-83396822 83397033 83398585 82884100

传真：0755-83376182 （0）13823648918 MSN: SUNS8888@hotmail.com

邮编：518033 E-mail:szss20@163.com QQ: 195847376

深圳赛格展销部：深圳华强北路赛格电子市场 2583 号 电话：0755-83665529 25059422

技术支持: 0755-83394033 13501568376

欢迎索取免费详细资料、设计指南和光盘 ；产品凡多，未能尽录，欢迎来电查询。

北京分公司：北京海淀区知春路 132 号中发电子大厦 3097 号

TEL：010-81159046 82615020 13501189838 FAX：010-62543996

上海分公司：上海市北京东路 668 号上海赛格电子市场 D125 号

TEL：021-28311762 56703037 13701955389 FAX：021-56703037

西安分公司：西安高新开发区 20 所(中国电子科技集团导航技术研究所)

西安劳动南路 88 号电子商城二楼 D23 号

TEL：029-81022619 13072977981 FAX:029-88789382