Teaching Real Time Signal Processing in a friendly environment.

Jean-Marie ORY

Ecole Supérieure des Sciences et Technologies de l'Ingénieur de Nancy
2, rue Jean Lamour F-54500 Vandoeuvre les Nancy

Abstract

In order to answer to the specific needs of fast process control and instrumentation we developed a Digital Signal Processing card around the DSP56309 processor from Motorola. We are also using this card for teaching Real Time Signal Processing in our school.

In order to make it easy to program by anybody while preserving the whole processor power, we have written a very simple language based on the interconnection of functional blocks. In this paper, we describe the language genesis and how we use it at school for practical works.

1. Introduction

Most Digital Signal Processor (DSP) cards available on the market are dedicated either to the communications or to the audio / video markets. A few are designed for controlling electrical motors. We found it worth developing a universal autonomous DSP card which would both have fast control and accurate metrology features, and which could withstand harsh industrial environments. This card is named mu.psi (**Fig. 1**) and is manufactured by Arnatronic, a local electronics company specialized in power control (see www.arnatronic.com).

At school, we are using this card to teach Real Time Signal Processing, but of course, that should not become a tedious assembly or C language exercise. Therefore we had to develop a specific compiler which converts a project description based on functional blocks into near-optimal native machine code.

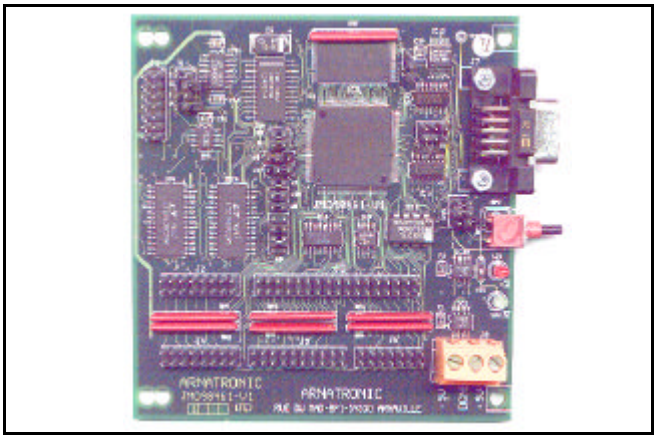


Fig. 1 The "mu.psi" DSP card

2. The "mu.psi" DSP hardware

The card is built around the DSP56309 from Motorola which has been chosen for its high (24 - 48 bit) resolution, its speed (100 DSP-MIPS) and its triple memory space Harvard architecture allowing efficient parallel data transfers. The chip has 102Kbyte internal RAM, 6 DMA channels, 5 interrupts, 3 high

resolution timers, 2 synchronous serial ports, one asynchronous serial port, one parallel port, and a JTAG test and emulation port.

External components consist in a 256Kbyte flash e2prom for resident debugger, application programs and constant data, two 12 bit AD converters, and two 12 bit DA converters. Sampling rates can be programmed from 1Hz to 1MHz.

Since this card is aimed at driving Kilo-Ampere power converters, a particular attention has be brought to security and noise immunity.

3. A textual block language

DSP design tools are seldom satisfactory. This is due in our opinion to 2 major reasons:

- The description of a RT signal processing algorithm is fundamentally different from a microcontroller or CPU program. A conventional CPU program can be best described by its *program flow*, while a DSP program is best described by its *data flow*. Some data flow description languages have well been developed in the past (e.g. SILAGE or Synchronous Data Flow), but they hardly emerged within the DSP programmers community .
- Digital Signal Processors have many different architectures, thus it is difficult to program them in a standardized way

3.1. Usual DSP programming tools

Most of the time, the programmer who wants to get the maximum efficiency out of his DSP has to use assembly language. However there is nowadays a quasi dogmatic attitude which preaches that everything should be programmed in C++. In that case, the tradeoff in code efficiency would be compensated by choosing a more powerful DSP chip.

Many constructors propose several sophisticated graphical environments with visual object programming, source debugging capability, simulation etc. ... Most of the time, these platforms support different DSP chips. Therefore, they cannot generate optimal native machine code.

3.2. Design with functional blocks

Actually DSP design looks more like to analog design. One will acquire a signal, filter it, analyze it, measure it, transmit it etc. ... Each elementary operation can be described as an independent functional block with inputs and outputs. Connecting blocks to each other defines the data flow. Block execution is triggered by the availability of data.

Programming in such a way makes projects much more understandable than conventional languages do. Now let's try to answer to the question:

"Is it possible to generate near optimal DSP machine code in this way ?"

3.3. Compiler specifications

- *Assembly language basement*

In order to exploit the full potential of the machine, each individual block function should be written in assembly language.

- *All data structures are static*

Since execution speed is the most important factor, time consuming stack operations other than subroutine calls and interrupts are prohibited.

- *Blocks hierarchy*

Since big blocks would sometimes lead to inextricable code, a big block should be made of several smaller blocks connected together and so on until we reach the elementary "atomic" block level. This involves that splitting into smaller blocks and connecting shouldn't introduce any tradeoff in the block's code performance.

- *Encapsulation*:

The invocation of one block should have following consequences:

Reserving the required H/W resources

Reserving variables

Creating the constant data and tables
Generating an initialization code segment
Generating the real time code segment
Installing new routines or ISRs if any
Providing context information
Providing timing information

- *Naming convention, heritage*

Each block has a unique name; sub-blocks names are generated by concatenating the parent block name with different strings, in other words, they inherit their names from parent block names. This also applies to embedded data structures naming. This convention insures different block names at any level, thus all internal variables are accessible at any depth. This also makes blocks hierarchy readable. For example a DSP radio receiver could contain following internal blocks:

Level	Block name
1	radio
2	radio_rf
3	radio_rf_mixer
4	radio_rf_mixer_oscillator
5	radio_rf_mixer_oscillator_pll
6	radio_rf_mixer_oscillator_pll_phasecomp

3.4. Assembler advanced features

The Motorola DSP relocatable macro assemblers have several advanced features which allow an efficient implementation of a textual functional block language:

- Macros with formal arguments
- Macros can be nested at any level
- Macros defined into macro-libraries
- Conditional assembly
- Sections with global / local attributes
- Name length up to 512 characters
- Embedded scientific calculator
- Cycle count at assembly time

3.5. Language syntax

Functional Blocks are described by assembly macros. A block invocation would therefore have following syntax:

[label] **funct** **name,arg1,arg2,...,argn**

where:
label is an optional label
funct is the function performed by this block
name is the unique identificator associated with this implementation of **funct**.
arg1,..,argn are parameters passed to **funct**

The arrangement of block invocations defines in which order real time data should be processed.

Connections define data flow paths. A connection syntax is:

cn **source,destination**

where
source is a block's output name
destination is another block's input name

source must have been defined before invoking the connection.

Optimal code is generated if **destination** has been defined prior the destination block invocation, but this condition is not compulsory.

If the programmer wants to get optimal code in any configuration, he/she may predefine block outputs by:

var	**name**[**,iv**]	for single 24b types
vard	**name**[**,iv**]	for double 48b types
varc	**name**[**,re,im**]	for complex types
varcd	**name**[**,re,im**]	for double complex

where
name is the output name of a later defined block
iv or **re:im** is the optional variable initial value at program startup

Then, he/she would define all the connections and finally define the blocks execution sequence.

4. Example: band-pass filter

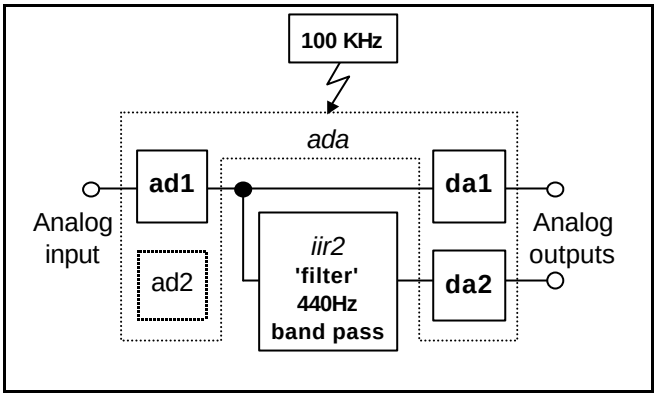


Fig. 2 Simple iir filter example

```
; filter example program
      cn      ad1,da1
      cn      ad1,filter_in
      cn      filter,da2
loop
      ada     1e5
      iir2    filter,bp,440.0,1000.,abs
      goto    loop
```

In the example described above, the **ada** block function acquires 2 analog signals and updates 2 DA converters at the specified sampling rate 1e5 (Hertz) given as argument; **ada** is a special case where no name has to be provided because this block may not be duplicated. Therefore output and input variables have predefined names called **ad1**, **ad2**, **da1**,**da2** respectively.

At execution **ada** waits until samples are present. Note that other AD acquisition methods would be possible: **irqada** executes an interrupt service routine on sampling events; **addma** uses a DMA channel to perform fast AD acquisitions directed into a memory buffer.

Connecting **ad1** to **da1** means that converter **da1** should be updated with the value acquired on **ad1** converter. The **da1** analog output is in that case a signal used to verify that analog acquisition does work.

filter_in is the input name of the iir2 block named 'filter'; the connection between **ad1** and **filter_in** means that the ad1 signal is actually the filter input.

filter is the filter's output which is connected to the **da2** converter.

The **iir2** block is a second order recursive filter which has following syntax:

iir2 *name,type,freq,q,mode*
where
name is the identification of this **iir2** instance
type defines which filter type to perform; *type* may be one of following strings: lp, bp, hp, bs corresponding respectively to low pass, band pass, high pass, and band stop.
freq is the characteristic frequency of the filter
q is it's quality factor defined as in a R-L-C circuit
mode expresses whether *freq* is a relative or absolute value. If relative, *freq* is given as a fraction of Fs/2 where Fs is the sampling frequency. If absolute, *freq* is given in Hertz. (Note that this implies that the compiler knows the value of the actual block's execution frequency).

The **goto** macro is a simple branch to the specified label.

5. Available block functions

Input / output	***Arithmetic***
Analog ad-da, polling	Gain
Analog ad-da, interrupt	Sum of weighted inputs
Analog ad-da, DMA	Multiply, divide
PWM polling	Schmitt trigger
PWM interrupt	
RS232 polling	***Functions***
RS232 DMA	Trigonometric
	Polynomial
Software Timers	Square root, Log, Decibel
Integer single shoot	1D Table read/interpolate
Integer periodic	2D Table read/interpolate
Fractional single shoot	
Fractional Periodic	***Control***
	Delay
String handling	Linear PID
Fractional to decimal string	Adaptive neural network controller
Integer to decimal string	1D, 2D Piece wise linear adaptive model
Integer to hex string	1D, 2D Polynomial adaptive model
Decimal string to fract.	

Block functions (continued)

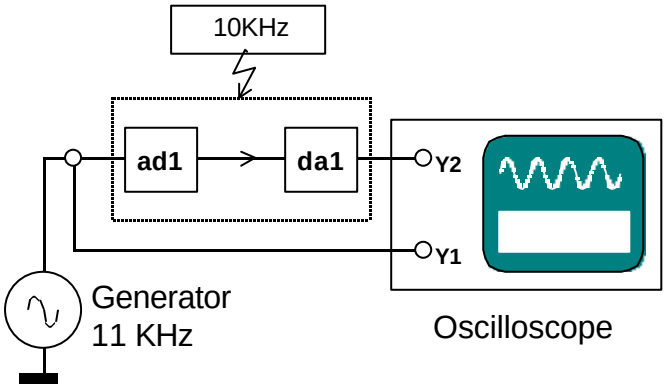
Signal generators	***Complex signals handling***
Saw tooth, Triangle	Complex multiply
Rectangle, bipolar PWM	Complex exponential generator, complex PLL
Sine / cosine	Hilbert Transformer
Random uniform	Complex IIR, FIR filters
Random gaussian	Complex oscillator
	Spectrum analyzer dB, power, amplitude modes
Filters	
1st order IIR lp, hp	***Boolean***
2nd order IIR lp, bp, hp, bs	Flag set, clear, toggle
FIR filter	Jump on (no) flag
Nonlinear	Test-clear-and-jump
LMS adaptative FIR	Wait on flag

6. Some lab experiments

Here are some examples of using the language for teaching signal processing:

6.1. Sampling and aliasing

The very first contact of the student with a RT discrete system will consist in observing a sine wave in its analog and sampled forms:

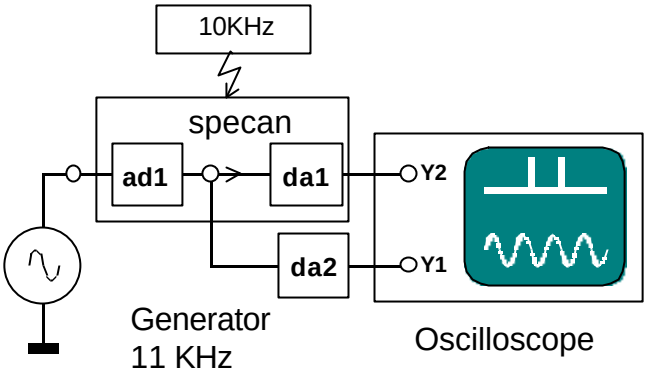



```
        cn          ad1,da1

loop    ada         1e4
        goto        loop
```

Fig. 3 Viewing the aliasing effect

Then, they will have a frequency domain explanation of the aliasing effect if they connect a software spectrum analyzer at the sampled signal source:

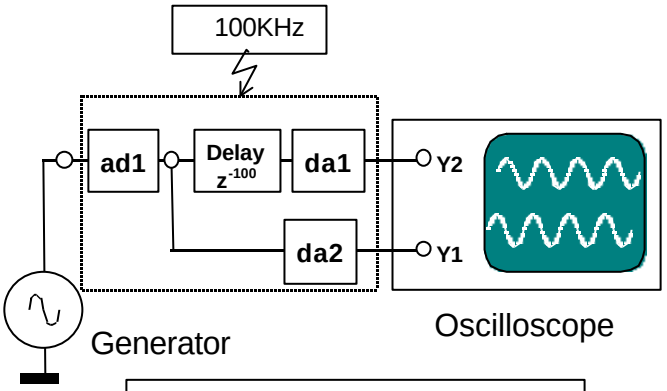



```
        cn          ad1,da2
loop
        specan      1e4,1024,pow,r,f
        goto        loop
```

Fig. 4 Aliasing effect in the frequency domain

6.2. Filters

FIR filters are introduced by studying the frequency to phase relationship introduced by a simple delay line:




```
        cn          ad1,dl_in
        cn          ad1,da2
        cn          dl,da1

loop    ada         1e5
        delay       dl,100
        goto        loop
```

Fig. 5 Phase shift of a delay line

Then they will experiment what happens if we add or subtract the delay line input and output signals (comb filters) etc..

One pleasant experiment is using an LMS adaptive FIR filter to get rid of echoes, which can be demonstrated by suppressing Larsen effect between a loudspeaker and a microphone.

6.3. Non linear control

Non linearities can be obtained in several ways. The most trivial is saturation.

Another one is a polynomial function. Polynomials are easy to generate with a DSP and they consume few execution time.

Any general 1D or 2D function can be easily obtained by the Table Read and Interpolate block. In a 2D table, if one of the entries "X" is a periodic saw tooth and the other one "Y" is an independent control input, we can produce a progressive change of a periodic wave form (morphing).

Students will observe the harmonics generated at a NL system output when excited with a sine wave at the input. They will then observe stability conditions in a recursive NL system and use the first harmonic method.

Adaptive NL control can be realized with the Neural Network block. We can for instance tell to the network that we would like to get a sine output while the input is an asymmetric saw tooth.

Chaotic systems can easily be realized. Students can observe stability basins as 2D images on an X-Y oscilloscope display.

6.4. Digital communications

Real time modulation and demodulation can be realized at high frequencies with functional blocks.

For instance, AM demodulation can be realized in the ideal way by using a Hilbert transformer associated with a complex PLL:

AM: $x(t) = (1+m(t))\cos(\omega t)$

Hilb: $x_H(t) = (1+m(t))\exp(j\,\omega t)$

PLL: $y(t) = \exp(j\,\omega t)$

Demod: $x_H(t) \times y^*(t) = 1+m(t)$

A software superheterodyne receiver has been demonstrated in the LW range.

Modem coding and noise effects are shown using V32 QAM modulator. A French literature text is encoded into a flow of 4 bit symbols, these symbols are then QAM modulated, polluted by a variable amount of noise, demodulated, decoded and displayed on a screen. Students observe the increase of errors within the received text when noise disks begin to hit each other on the screen of the oscilloscope.

7. Industrial applications

We used our functional block language within a huge industrial control application: An intelligent TIG welding machine

A single mu.psi card controls all the functions of a latest generation Tungsten Inert Gas welding machine. Basically the DSP has to control a 80KHz Pulse Width Modulator which is the command input of a 600 Ampere inverter. The inverter output is an AC time varying wave form current source. The precision is better than 1%. Since the arc is a non linear time varying process, a polynomial adaptive learning controller has been developed. Many additional blocks are necessary: multiple working modes, securities, electrode thermal model, man-machine interfaces, etc. The whole application written in functional block language is less than 50 lines long. When compiled, however, the generated assembly listing extends to 30 000 lines !

8. Conclusions

We have tried to show how easy and efficient the Functional Block Programming method could be. Teaching RT Signal processing no longer looks like to a tedious C programming language exercise !