Programming with functional blocks using FIBULA

Why a functional block language ?

Digital Signal Processors have very specific architectural characteristics which make them particularly efficient for most signal processing tasks (MAC instruction, parallel instructions, modulo addressing, DMA channels, wired DO loops etc. ...). *However this implies that several instructions are bound together in indivisible blocks, in order to preserve high performance.* Furthermore required resources affected to each block instance such as memory, registers and DMA channels must have been properly reserved and initialized at the beginning of the program.
Conventional languages such as C / C++ are translated on an instruction by instruction basis and therefore fail in performance as compared to DSP native assembly language block programming. Speed ratios between assembly and C higher than 10 can frequently be observed.
The best language for a DSP is thus a set of basic functional blocks. You build your application simply by connecting them together and defining in which order they should execute.

Functional blocks

A block groups following elements:

- Hardware resource reservations
- State variables reservation
- Constant data creation
- Initialization executable code
- Real time executable code
- Optional debug code
- Register modify and timing reports
- Documentation with demo

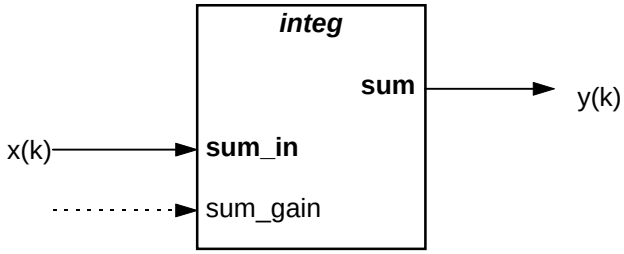


Fig. 1 The integrator block

The block represented on Fig. 1 is a discrete integrator which performs y(k) = y(k-1) + gain ∗ x(k).
It's implementation is given by following macros:

```
                                    ; Connections:
cn      block1 , sum_in             ; define input connection
cn      sum , block2_in             ; define output connection

...                                 ; Execution loop triggered each sampling period:
...                                 ;
integ   sum , 0.01                  ; execute integrator "sum" with gain = 0.01
                                    ;
```

integ is the name of the function performed.
sum is the name of *this* instance of the ***integ*** function, and is also the name of the output connection.
sum_in is the name of the input connection. Inputs and outputs must always be connected somewhere.
Usually, **gain** is a constant parameter which you give as second parameter of the function call. However, gain becomes an input variable if you connect the optional **sum_gain** input somewhere. Such optional inputs are called parameter inputs. They may be left unconnected, in which case the function call parameter, or the default parameter prevails.

Each block function is written as an open source assembly file named *function*.asm placed in the "lib" folder. If a modified version of the block has been copied to "userlib", this modified version will be taken in lieu and place of the standard one. This allows the user to easily customize the language at no risk.

About connections

A connection between **Block_A** and **Block_B_in** is defined by the statement
cn Block_A , Block_B_in

It simply states that **Block_A** and **Block_B_in** represent the same variable. If neither **Block_A** or **Block_B_in** are yet defined, the **cn** statement reserves one word for the common variable and associates the two names to it.
Due to the multi field architecture of a DSP, in order to optimize code, connections must be known before blocks are implemented . Therefore, **cn** statements must appear ahead of executable code.

Connections and data types

Variables bound by a connection have to be the same type. The default type is real single precision (24 bit fractional [-1.0 .. +1.0[format on DSP56xxx). Other types implemented in FIBULA include:

Double (48 bit fractional) :	use	**cnd**	**source,dest**
Complex (2 x 24 bits fractional):	use	**cnc**	**source,dest**
Boolean (single bit):	use	**cnb**	**source,dest**

You may also first define the variable types and their initial values, and then connect them with **cn** :

Single:	**var**	**name [, initval]**
Double	**vard**	**name [, initval]**
Complex	**varc**	**name [, ro , theta , p | , re , im , c]**
Boolean	**flagr**	**name**

Default initial values are always 0.

Blocks hierarchy

Blocks can easily be built from sub-blocks at any level, provided naming conventions are respected. Each sub-block's name begins with the main-block's name followed by an underscore
This allows the programmer to access any variable, buried at any depth in a normalized manner.

Example (Fig. 2): a Sine - cosine generator **g_sincos** implementation named **sc** contains a saw-tooth generator named **sc_phi** and a sine-cosine function named **sc_fsc**. This function itself is made of 2 sub-functions named **sc_fsc_re** (= cosine) and **sc_fsc_im** (=sine) and so on ...

If the user wants to get the phase, of the complex sine-cosine pair **sc**, he can easily connect to the internal variable **sc_phi**.

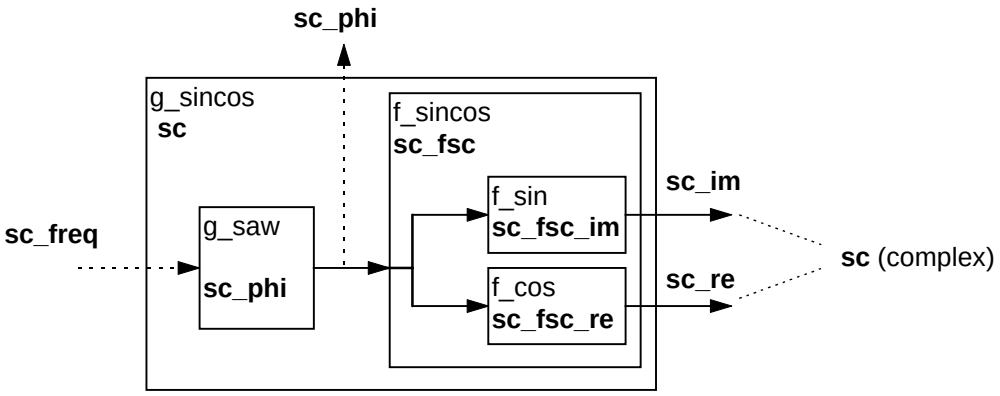


Fig. 2 Internal structure of the **g_sincos** block

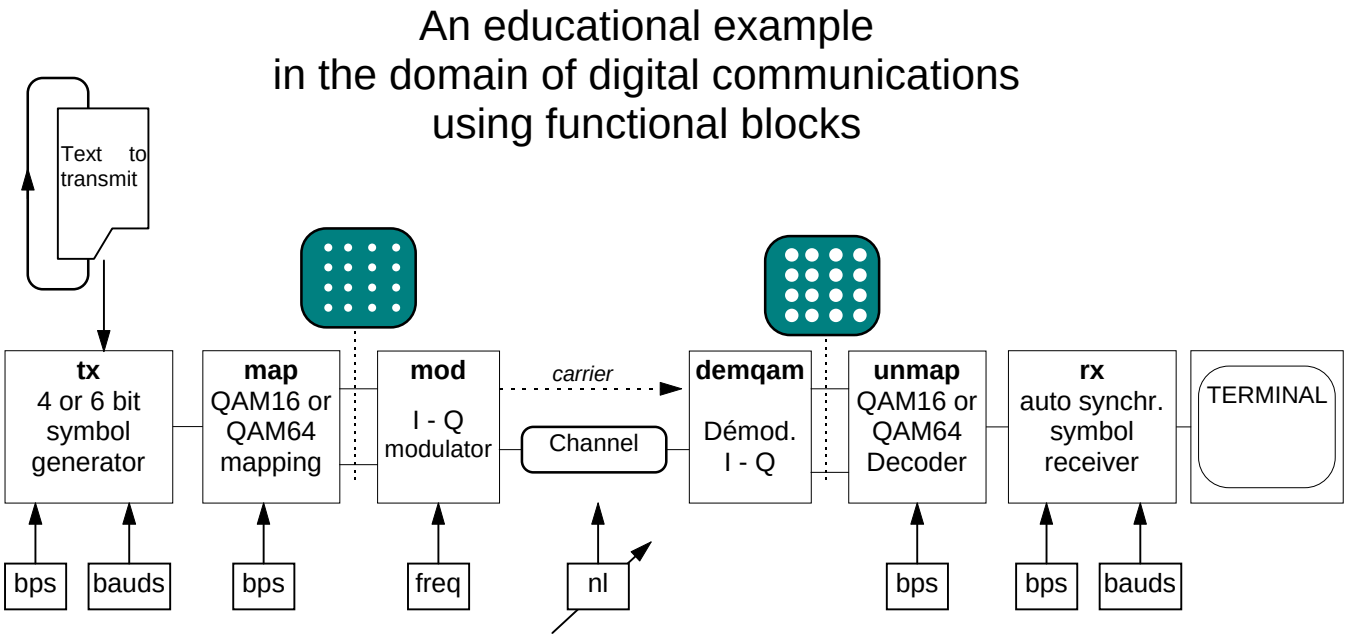


Fig. 3 Influence of channel noise on a QAM modulation

Figure 3 represents a lab experiment which simulates in real time a simplified modem.

An ASCII text continuously read in a loop is split in 4 or 6 bit slices named "symbols". Each symbol is then converted into a phase - amplitude pair (QAM mapping), this pair modulates a sine wave generator in phase and quadrature. The result is sent through a channel which has an internal system function (FIR filter simulating echoes) and where the signal gets polluted by an adjustable amount of external noise (nl).
The signal is then demodulated, unmapped, and reconstructed to ASCII characters and sent to the RS232 line. The received text is displayed on a computer terminal.

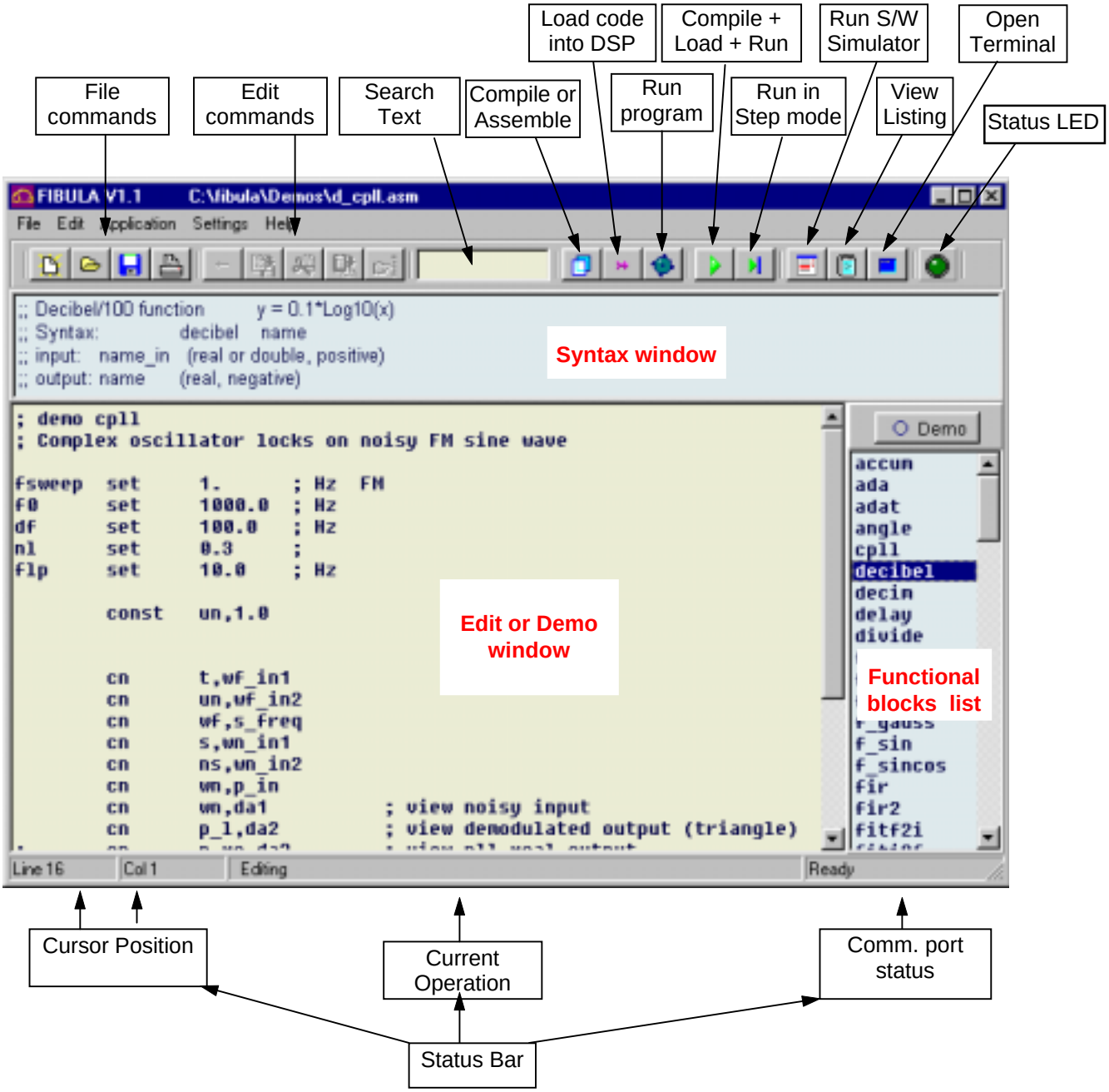
The experiment shows the QAM constellation on an oscilloscope. As the noise level increases, the dots of the constellation become bigger and bigger. Errors appear in the displayed text whenever the noise disks hit each other.

Hereunder the program represented in Fig. 3 written in FIBULA language:

```
;Transmission parameters:

bps     set     6           ; bits per symbol (4 or 6)
bauds   set     1500.       ; symb per sec (1. to 20000.)
freq    set     8000.       ; carrier frequency (Hz)
nl      set     0.1         ; adjustable noise level

; Predefine complex variables
        varc    mp,0,0,c
        varc    md_osc,0,0,c
        varc    dm,0,0,c

; Connexions:
        cn      e,mp_in     ; Transmitter to mapper
        cn      mp,md_in    ; Mapper to modulator
        cn      md,c_in     ; Modulator to channel
        cn      c,dm_in     ; channel to demodulator
        cn      md_osc,dm_car ; transmit carrier
        cn      dm,um_in    ; Demod. to unmapper
        cn      um,r_in     ; Unmapper to receiver

; View received constellation:
        cn      dm_re,da1       ; Scope X channel
        cn      dm_im,da2       ; Scope Y channel

; Program execution:

loop    tx      e,bauds,bps     ; Symbol generator
        map     mp,bps,const64  ; Mapper
        mod     md,freq         ; Modulator
        canal2  c,0,0,nl        ; Channel
        demqam  dm,1./freq      ; Demodulator
        unmap   um,bps,const64  ; Unmapper
        rx      r,bauds,bps     ; Sync. symb receiver
                                ; and  send to RS232
        goto    loop
```

The FIBULA development environment

The FIBULA Integrated Development Environment is aimed to ease the code generation process and the debugging of DSP56300 programs.



The instruction list and the syntax window are available in **Fibula compiler mode.** They appear with the command **Help | Functional Blocks**.

When a demo is available for the selected instruction, this demo may be loaded in the edit window by pressing the Demo button, and may be compiled and run.

Closing the demo restores the current program.

Status Led Color	Description
GRAY	No serial comm.
DARK GREEN	DSP Ready
AQUA, blinking	Compilation active
LIGHT GREEN	Success (compilation or download)
RED	Compilation errors (syntax)
BLACK	Compiler internal problem (may be caused by cyclic références or macro infinite recursion).
PURPLE blinking	Downloading failed (Error during communication)
BLUE, blinking	DSP running
YELLOW	Step mode active.

Compiling, downloading, running the program:

Usually, you will first press the reset button of the DSP card and then activate the compile + download + run button (green triangle). If you just want to check the syntax, use only the compile button. If you want to download the last compiled program, press the download button. Press on the run button to start your program.

Compiler languages:

1 FIBULA language and assembly:

In this mode, you may use a high level interconnected blocks textual description for your program, using several macros from the libraries. Assembly instructions may be used, but you must conform to the rules of the FIBULA language (naming conventions, sections where data and code reside).

2 Assembly with minimal I/O library and startup program

In this mode, your program will be assembled with an epilogue containing an INIT routine, the AD-DA analog I/O, and the SCI serial port I/O routines. Use this mode if you want to learn about the DSP assembly language.

3 Absolute assembly, no library

Use this mode if you want to check a code segment without any external interference.

Running in step mode

When compiling a program in step mode, a software interrupt is added at the end of each block, which allows the user to view on the terminal window input and output values of each executed block.
Pressing the space bar will execute the following block.
Pressing "g" (go) will run the application at it's normal speed.
Pressing "h" (halt) will return in the step mode
Pressing "f" (fast stepping) will run the program in step mode, as fast as possible, limited by the serial communication baud rate.
Pressing Escape will quit the application program and returns to the resident debugger.
If a big block is made from several sub-blocks, stepping through will execute the big block as an entity unless the debug level has been raised by one or more units.

Software simulation

If you are getting trouble while running a program written in assembly language and you want to understand the processor's behavior, you may open the software simulator. The simulation applies to the last compiled program. The simulator is a high performance Motorola product that you might have to configure to meet your specific needs. Every register and memory can be observed while stepping at the assembly instruction level. Inputs and outputs can be simulated using data files.

Listing window

Pressing on the listing button opens the listing window. While using the FIBULA language, provided no errors have occurred at compilation, the listing detail level is set to it's lower value. To get more details, you may add one or more **list** directives at the beginning of your program.
If errors occur at compilation, the listing window automatically opens and shows the errors highlighted in red.

Opening the terminal

The terminal window displays the ASCII serial communication between the PC and the DSP card. However the terminal display function is inhibited during code downloading. You may use the terminal to manually interact with the DSP card using the resident debugger:

Viewing / modifying the memory content:

Type **x 123 <enter>** to view the content of address $000123 in the **X:** space.
Type **<space>** to go to the next address, or **"/"** to go to the previous address;
Type **.567 <enter>** to change the content to a new fractional value or
type **345678 <enter>** to change the content to a new hexadecimal value.
In the same manner, you may view / modify memory contents in the fields **Y:, P:, L:.**

Running a program:

Type **g 100** to run a program located at P:$100
If the program has been downloaded in the **.lod** format, you may use source symbols to retrieve addresses e.g. **x sine_wave <enter>** displays the variable named "sine_wave" in the source program.