OBJECT-ORIENTED PROGRAMMING OF DSP PROCESSORS: A CASE STUDY OF QuickC30

Matti Karjalainen

Helsinki University of Technology, Acoustics Laboratory
Otakaari 5 A, SF-02150 Espoo, Finland
E-mail: matti_karjalainen@hut.fi

ABSTRACT

Object-oriented programming (OOP) is a new paradigm and methodology for software development and design. This paper is a case study of applying OOP to the programming of the TMS320C30 signal processors in time-critical applications. Data types and structures are defined as classes (abstract data types) in a systematic way without need for compromising between flexibility and efficiency. Procedurally the environment supports method (function) definitions in a high-level language, low-level macro instructions, and middle-level method instruction definitions integrated by a common syntax so that even low-level instructions may refer to the high-level OOP data types. The system is implemented in Common Lisp and follows the syntax of Common Lisp Object System (CLOS). Several examples of DSP programming are given in the paper.

INTRODUCTION

The programming of digital signal processors remains a demanding real-time programming task where often all the efficiency features of the processor are to be exploited. This is typically achieved by the use of an assembler language. On the other hand, higher-level languages like C are utilized for less time-critical parts of programs. Such a multiple language approach doesn't fully support coherence and integration of software. It also lacks the very desirable features of the new programming paradigm, object-oriented programming (OOP), including high reusability of software, fast prototyping and easy redesign, good control over complex data structures, etc.

OOP languages (e.g. C++, Smalltalk, CLOS, Flavors) show a new improved way of programming and software design. New data types are defined as classes and the behaviour of objects instantiated from classes is defined by methods (method functions). Several advanced DSP programming environments have been based on the OOP approach (e.g. ISP by Kopec [1], SPLICE by Myers et al. [2], QuickSig by the author [3]. OOP is becoming a standard programming tool also in DSP along with the use of C++ as the major extension to the C language. The advantages of OOP have not been exploited, however, in programming of time-critical applications on DSP processors. It is important to increase the productivity of programming also here.

The original contribution of this paper is to show one systematic approach to do this. In the paper an experimental software environment called QuickC30 is described where OOP methodology is used to meet both the programming flexibility and the runtime efficiency requirements that are inherent in complex DSP applications. The environment has high-level support for defining data types and structures as OOP classes. From a procedural point of view there exist several levels from high-level method definitions (processor independent) down to assembly code (with the possibility to refer to high-level data types and structures). This mixture of high and low level features is fully integrated (also syntactically coherent) by using the Common Lisp language and CLOS (Common Lisp Object System) [4] as the system implementation language. The target processor of this study is the TMS320C30 floating-point DSP processor.

QuickC30 OBJECTS

Like any modern OOP environment the present one is based on the definition of classes as data types, methods as class-specific functions and procedures, as well as instantiation of objects based on classes which are able to provide functionality by the methods. The definition of a class in QuickC30 is similar to the form used in CLOS, e.g.:

```
(defclass point-2D ()        ; class point-2D, no inheritance
  ((x :type float :initform 0.0 :initarg :x :accessor x)  ;;; c1
   (y :type float :initform 0.0 :initarg :y :accessor y))) ;;; c2
```

where comments are:
c1) slot x, type float, default value 0.0
c2) slot y, type float, default value 0.0

This creates the class **point-2D** of points in a two-dimensional space. Slots (also called instance variables) x and y keep the state (i.e. the coordinate values: x and y) of a point. Initform keywords specify default forms for the slot values and initarg keywords may be used for giving initial values for the slots, see the make-instance example below. Accessor functions (also named x and y) are created for reading and writing the coordinate values. In contrary to normal CLOS the slot types must be explicitly declared. Multiple inheritance is available for the definition of new classes as specializations and combinations of existing ones.

0-7803-0532-9/92 $3.00

The instantiation of objects is carried out by using **make-instance** forms such as:

```
(make-instance 'point-2D :x 2.3 :y 3.4)
```

that creates a point-2D instance with coordinate values x=2.3 and y=3.4. In addtion to the initargs (:x and :y above) several system supported keywords may be used, e.g. for specifying the memory block for allocation (:mem-block '.ram0-block) or special allocation for circular addressing forms (see ring buffers). Special constructor functions may also be defined and used instead of make-instance forms.

Procedural programming (as opposed to data structuring) may be based on high-level (processor independent) method definitions or on low-level code where all processor-specific efficiency features are available. The lowest level is the writing of assembly code in a flexible assembler with Lisp syntax [5,6]. New macro instructions may be defined by the **def.instruction** form, e.g.:

```
(def.instruction .add-int&float-to-dst (int float dst)
  (.float int '.r7)    ; int to float -> register R7
  (.addf float '.r7)   ; add float -> register R7
  (.stf '.r7 dst))     ; save result R7 -> dst
```

A more structured and abstract way is to write so called method instructions [6] by **def.method** such as:

```
(def.method .distance ((p1 point-2D) (p2 point-2D) value ; 1
                       &key (val '.r5) (temp '.r4)     ; 2
                        (p1ptr '.ar6) (p2ptr '.ar5))   ; 3
  (with.slots ((x1 x) (y1 y)) p1 p1ptr)                ; 4
  (with.slots ((x2 x) (y2 y)) p2 p2ptr)                ; 5
  (.ldf x1 val) (.subf x2 val) (.mpyf val val)         ; 6
  (.ldf y1 temp) (.subf y2 temp) (.mpyf temp temp)     ; 7
  (.addf temp val) (.sqrt val val)                     ; 8
  (.stf val value))                                    ; 9
```

where the comments are:
1) name, arguments p1, p2, value
2) named registers (R5, R4)
3) object address registers (AR6, AR5)
4) named references to p1 slots: x1 and y1
5) named references to p2 slots: x2 and y2
6) load x1, x1-x2, squared
7) load y2, y1-y2, squared
8) sum, square root (macro instruction)
9) save result to value

The arguments given to method instructions may be specialized as built-in data types or classes (point-2D here). Such method instructions are still assembler instructions but they provide a more direct way to refer to OOP-level objects and their slots.

High-level (processor independent) programming follows the CLOS convention, e.g.

```
(defmethod distance ((p1 point-2D) (p2 point-2D)) ; name,args
  (let ((dx (- (x p1) (x p2)))          ; x difference to local dx
        (dy (- (y p1) (y p2))))         ; y difference to local dy
    (sqrt (+ (* dx dx) (* dy dy)))))    ; Euclidian distance
```

A slightly more efficient method function can be defined by combining the low and high-level code:

```
(defmethod distance ((p1 point-2D) (p2 point-2D))
  (asm ((val float))
    (.distance (%a p1) (%a p2) (%a val))))
```

Here the **asm** form makes the interface between high-level and low-level code; (val float) is the value and type definition and forms such as (%a p1) refer to the stack frame for parameter and value passing.

The difference between QuickC30 methods and general CLOS methods is that QuickC30 requires static (compile-time) typing instead of runtime type checking and dispatching as in Lisp and CLOS. This means that every parameter must be specialized; e.g. in the form (p1 point-2D) above parameter p1 is specialized to point-2D. This is important for the efficiency of the code and in DSP applications just slightly limits the programming flexibility when compared to the full Lisp language.

Built-in data types and libraries

The QuickC30 programming environment has predefined classes for the most important standard data types, i.e., built-in types. Currently these include numbers (**integer, float, complex**), boolean values (t = true, nil = false), and arrays. Arrays are divided into two groups: static arrays and (normal) arrays. The difference is that static arrays do not have access to their size at runtime while (normal) arrays do know their size e.g. for bounds checking. Static arrays are intended to be used as slot values for composite objects that know the size of their array subparts. Among the static arrays are, e.g.:

static-float-array-1d = one-dim. array of float elements.
static-integer-array-1d = one-dim. array of integers,
static-float-ring-buffer = float array to be used as a ring buffer (circular addressing)

The QuickC30 environment includes libraries for numeric computation (both low-level macro instructions and high-level methods). There are also predefined class libraries for DSP programming, including filters, transform objects, analyzers, signal generators, AD&DA converters, etc., with related macro instructions and methods. The following examples show how these are implemented and how new constructs may be developed by the OOP approach.

EXAMPLES OF OBJECT-BASED DSP

In this chapter we show some simple but representative examples of how to use the object-oriented features of the QuickC30 in formulating typical DSP problems.

FIR filters as objects

FIR filtering is a good example of typical DSP programs where the TMS320C30 provides processor-specific features to speed up the computation [7]. First, there are parallel instructions that execute multiple primitive instructions in a single clock cycle (e.g. multiply-add). Second, the C30 has zero-overhead loops for repeated iteration of single or multiple instructions. Third, a special circular addressing mode is available for

using ring buffers (modulo n indirect incremental or decremental addressing) in such applications as FIR filtering. From the data structuring point of view there is need for proper allocation of memory for the tap coefficients and delayed sample values. In the following definition of class FIR the delayed sample values are allocated as a ring buffer so that an address register increment and decrement automatically wrap around while the coefficient buffer is a linear array so that its start address is loaded from the slot data-ptr in the beginning of each filtering step.

```
(defclass FIR ()
  ((size :type integer) ;;; filter size
   (coeff-buffer :type .static-float-array-1d)
   (data-buffer :type .static-float-ring-buffer)
   (data-ptr :type integer)))
```

For proper instantiation of an FIR filter there exists the **initialize-instance** method that accepts init keywords and fills the slots to construct an instance of FIR when the form (make-instance 'FIR {keyword-options}) is evaluated. The default behavior uses a defined default memory block for the substructures and allocate the required slot types according to the size of the filter. It is possible to pass init keywords e.g. to allocate the buffers from desired memory blocks (e.g. internal ram blocks) to optimize the performance by avoiding pipeline conflicts [7]. It is also possible to pass pre-instantiated buffer objects to the slots or to create the filter without buffers and afterwards to assign the buffers to the filter objects. Size and type compatibility of buffers is always checked.

The functionality of FIR filters is provided by proper method instructions and functions. The following example shows the principle of writing a method instruction for FIR to compute a single step of filtering:

```
(def.instruction .filter-step ((self FIR) in out
                               &key (object '.ar0)
                               (dptr '.ar4) (cbuf '.ar5))
  (with.slots (size data-ptr coeff-buffer data-buffer)
              self object)
  (.ldi size '.bk) ; size to block-size register
  (.ldi '.bk '.rc) ; size to repeat counter
  (.subi 2 '.rc) ; decrement repeat counter
  (.ldi data-ptr dptr) ; load data-buffer circulating address
  (.ldi coeff-buffer cbuf) ; load coeff-buffer start address
  (.stf in `(,dptr)) ; store input to data buffer
  (.ldf 0.0 '.r2) ; reset accumulating sum
  (.mpyf3 `(,dptr >++ 1 %) `(,cbuf >++ 1) '.r0) ; first mul
  (.rptb-macro  ; repeat macro starts
    (.mpyf3//addf3 `(,dptr >++ 1 %) `(,cbuf >++ 1) '.r0
                   '.r0 '.r2 '.r2)) ; parallel multiply-add
  (.addf3 '.r0 '.r2 out) ; last sum
  (.nop `(,dptr >-- 1 %)) ; data move (rotate in ring buffer)
  (.sti dptr data-ptr)) ; update slot data-ptr
```

When the method instruction is defined it may be used like any C30 instruction in writing new low-level code. It is also possible to define a high-level method (e.g. filter-step) by defmethod (like for **distance** above).

An important advantage of object-oriented programming is the possibility to specialize the same method (here .filter-step) to as many classes or data types as one wishes. We may for example write a similar **.filter-step** macro instruction for an IIR filter class and when using **.filter-step** later the system will automatically use a proper version according to the type of the first argument in the call. Such a group of methods sharing the same name forms a generic function.

LPC analyzer objects

As an example how to construct more complex composite objects for DSP we consider an LPC analyzer as an object class. The subunits are: windowing object with arrays (data buffers) as substructure, an autocorrelator object with its data buffers, and the lpc-solver (solving of reflection coefficients and direct-form filter coefficients). Fig. 1 depicts the principal structure of the analyzer and its subunits.

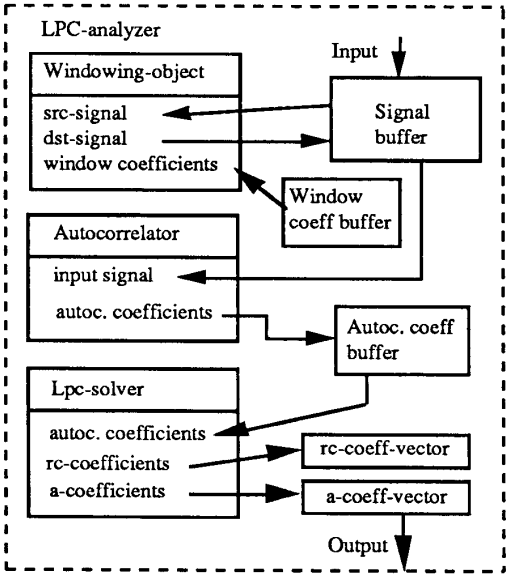

Fig. 1. *A complete LPC analyzer as an object structure.*

Each substructure of the LPC-analyzer may first be defined by defclass and finally the subunits are represented as slots in a defclass form for the LPC-analyzer. Notice the sharing of data buffers by the subobjects. An initialization method is defined for LPC-analyzer so that size parameters and optionally also some buffers can be given when instantiating a desired analyzer object. A method instruction (or method) may be called to execute the whole analysis sequence.

Other DSP applications

The QuickC30 environment supports various standard DSP components in an OOP form; A/D and D/A converters are objects, FFT analyzers objects are available, etc. QuickC30 has been used when developing several time-critical applications such as auditory modeling and speech recognition, synthesis of speech, modeling and synthesis of musical instruments (guitar and flute), and digital audio experiments.

SYSTEM FEATURES

In addition to the seamless integration of different programming levels the flexibility of the QuickC30 is based on several features that exist in the Common Lisp and CLOS environment:

* The system is fully interactive and rich in tools: interpreter and compiler, efficient debugging and inspecting facilities of Lisp are available also in QuickC30 programming.

* Incremental compilation and execution of program code (each expression separately) makes program development fast and productive.

The implementation of QuickC30 can be considered not only as a separate application program but also as an extension to Lisp and CLOS. It is important to recognize that the programming model is based on two environments: the host environment and the target (C30) environment. The host language and tools (normal Lisp and CLOS) are available for general purpose programming, e.g. to the management of C30 processors in the system. The target language is primarily for the programming of the target processor run-time code. (For clarity the symbol names in source and target languages are expressed in different Lisp packages or symbol name spaces.) One nice example of the QuickC30 system is the the definition of objects that are distributed between the host computer and the C30 so that only those slots and methods that are allocated for the C30 are needed at C30 runtime.

Among the most important features of the QuickC30 implementation are:

* Multiple TMS320C30 processor programming support. Each processor in the system (1 to 4 processors are used so far) is represented as an object. A global variable (*C30H*) keeps the selected one that is the target of host communications. By changing the value of this variable the target processor may be changed whereby the whole C30 environment (variables, objects, methods, i.e., the memory allocation) is switched so that only the selected processor and its environment are directly visible.

* Advanced memory allocation schemes. Each processor in the system has its own object allocation table. Allocation of memory is deferred as much as possible so that for variables and method functions the real allocation is carried out only when the variable or method is used for the first time. Automatic deallocation and garbage collection are not used for efficiency reasons.

* QuickC30 programs can be executed interactively and incrementally on all programming levels. Even assembly instructions are true Lisp functions that can be evoked interpretively or called from other functions. All low-level forms are assembled, down-loaded, and executed immediately when called at the speed of approximately 1000 - 5000 ordinary instructions per second. High-level methods and functions are called by passing arguments and values on the stack.

* Most of the QuickC30 software is written to be portable to new host machines and target (C30) processor boards. So far we have used Macintosh II computers and Symbolics Lisp machines as host computers and five different C30 hardware realizations (including National Instruments NB DSP2300 board for the Macintosh) as target processors. QuickC30 software supports multirate A/D and D/A conversions by proper converter hardware.

SUMMARY AND DISCUSSION

QuickC30, an object-oriented programming environment for the TMS320C30 described above has been used for programming several demanding real-time applications, including speech analysis, synthesis, recognition, neural networks, digital audio experiments, and computer music (real-time sound synthesis). The system is found to be very flexible and easily extendable, rich in programming tools, and bringing most of the OOP advantages to DSP processor programming. Although some of the most demanding parts of programs still remain processor-specific (this seems unavoidable in many cases) - data structuring and a major part of the programs are high-level code and as such directly retargetable. The OOP and integrated software approaches make it easy to combine different levels and to optimize efficiency vs. flexibility in each part of a program.

There are many challenges for future expansion of the QuickC30 programming environment. It would be beneficial to write interfaces (i.e. readers and parsers) from standard C30 assembler and C language in order to make the system compatible to these de facto standards in DSP processor programming. It would be important to study the applicability of other OOP languages, such as C++ and Smalltalk, to DSP processor programming. A major challenge is also to expand the current runtime kernel of QuickC30 to support advanced multiprocessing (especially for TMS320C40) needed in more demanding applications.

ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland. I would like to direct my thanks to all who have contributed to the results of the study, especially Matti Vartiainen, Kari Järvinen and Vesa Välimäki.

REFERENCES

[1] G. Kopec, "The Integrated Signal Processing System ISP," IEEE Trans. ASSP, vol. ASSP-32, No. 4, Aug. -84.

[2] C. Myers, "Signal Representations for Symbolic and Numerical Processing," PhD Thesis, MIT Tech. Report No. 521, Aug. 1986.

[3] M. Karjalainen, "DSP Software Integration by Object-Oriented Programming: A Case Study of QuickSig," ASSP Magazine, no 2, vol. 7, 1990.

[4] G.L. Steele, Common Lisp, The Language. Second Edition, Digital Press 1990.

[5] M. Karjalainen, "A Lisp-based High-level Programming Environment for the TMS320C30," Proc. of ICASSP-89.

[6] M. Karjalainen et. al, "A Programming and Code Generation Environment For Non-Homogeneous DSP Multiprocessors," Proc. of IEEE ISCAS-90.

[7] TMS320C3x User's Guide, Texas Instruments Inc, 1991.