XVII IMEKO World Congress
Metrology in the 3rd Millennium
June 22-27, 2003, Dubrovnik, Croatia

SOFTWARE DEVELOPMENT FOR ADC TESTS

Jiri Brossmann, Jaroslav Roztocil, Vojtech Ruml, Dusan Varga

Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic

Abstract - Object based universal software system for ADC testing which enables an easy programmer's access and expansion (update) is introduced in this paper. The basic idea of the object oriented design and some aspects of the software solution of the ADC testing are described.

Keywords: ADC testing, software development, object oriented programming

1. INTRODUCTION

At present, ADC tests are based on commonly accepted standards as [1] and [2]. These tests are usually performed by closed programs [3] or scripts and libraries tied together with commercial products, for instance Matlab [4].

In case, the procedures (implemented in the test described above) are updated and some devices in a measurement chain changed, the design must be completely rebuilt or rewritten in the more complicated way.

To avoid this conceptual problem, it was decided to develop a new object based kind of software for ADC dynamic testing, called DIGESTER II.

2. OBJECT ORIENTED DESIGN

The basic idea of the DIGESTER's object oriented design is to simplify a development and usage of real-world objects (see Fig. 1) in a software abstraction by the implementation of the specific tests (i.e. some abstract algorithms that describe how to provide specific tasks). There are several groups (or classes) of objects: measurement instruments, data containers, data processors etc. which might be converted into any software modules for the simulation of basic behaviour of the real objects, or into some sort of interface between software and hardware.

The real objects often overlap the unambiguous classes (a digital oscilloscope works as both, the sampling or storing device). It allows a programmer implementing the ambiguous drivers that fully describe the chosen instruments. The usage of this approach is not worth in general because of the variation of instruments for the performance of the same tasks (the different types of digital oscilloscopes require different ways to handle the stored data) that means the necessity of complete changing the test design. Basically, there are two different ways such problem to be avoided:

❖ to take into account all the possible configurations of hardware and to consider all conditions to enable the implementation of the test in all presumable situations;
❖ to increase the level of abstraction (a driver represents a general instrument).

The second approach was used in a new design of the DIGESTER. In case of the substitution of an instrument, there is no need to change the test methods. The possible changes are necessary in the setting of the instrument.

The following abstract classes were devised and used as root classes for C++ implementation of DIGESTER:

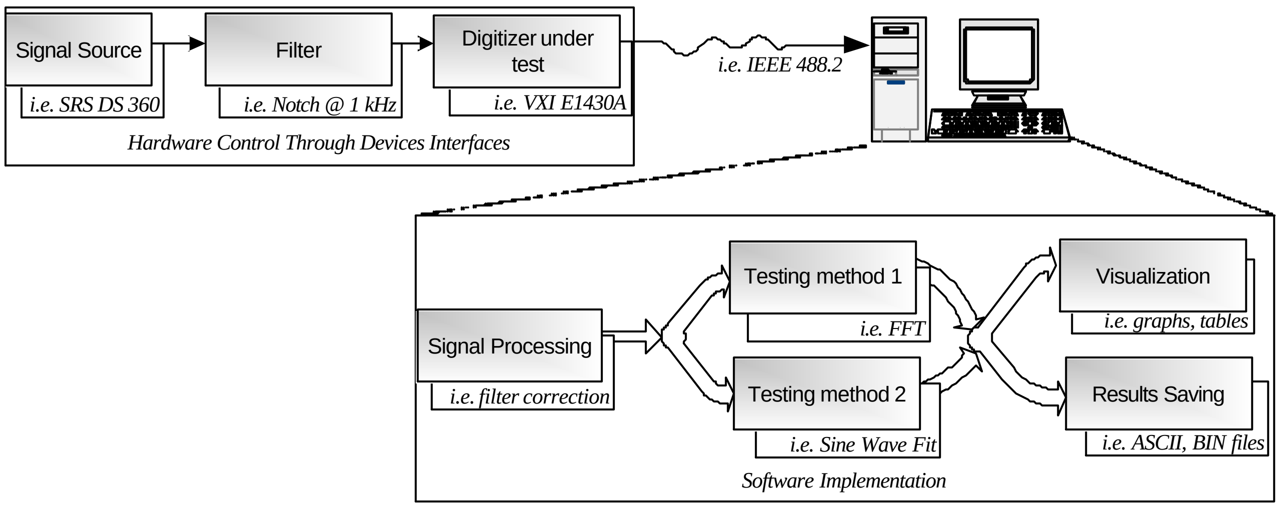

Fig. 1. Typical ADC chain testing setup

- **instrument** (a driver or simulation of a real device);
- **container** (an object intended to store the various kinds of data);
- **processor** (an object for processing of the data stored in containers);
- **summary** (an object for collecting the results of the tests and for the final test report generalization).

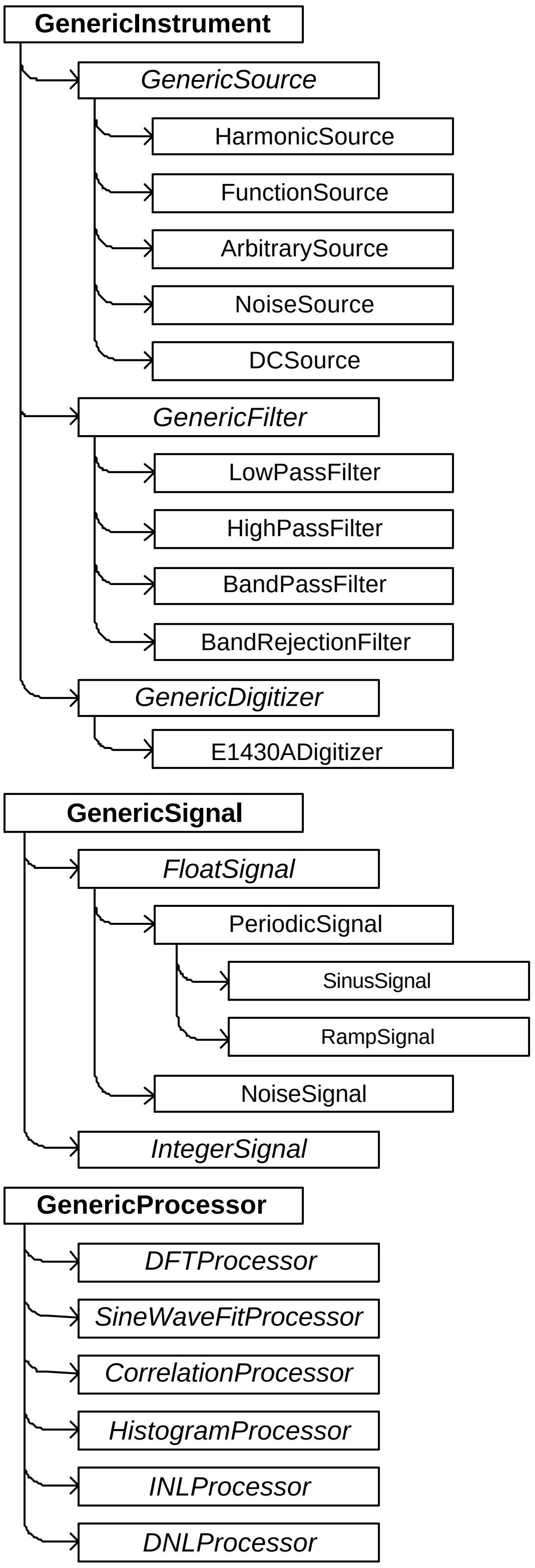

Fig. 2. DIGESTER II classes tree

The particular sub-classes are derived from these generic classes (e.g. instrument is root class for sources, and then arbitrary source is derived from source). On every level, the most common attributes and functions are investigated. When lowering the abstraction to the lowest level (i.e. to real hardware), only a few additional functions must be implemented, and some virtual methods are necessary to be rewritten to accomplish the requirements of the device.

Another important part of the implementation is a detection of the ancestors and descendants in the implementation line. To include the processor in the implementation line is useful just in case, it recognizes the ascendant – descendant relations between containers.
This detection is provided by the usage of a special component of every object, called *descriptor*. Every descriptor contains information about its type (*class*). It is referred as a string with every level of the abstraction separated by a dot (e.g. "Container.Array.Integer" or "Instrument.Source.Function.DS360").

At least, every object contains its programming interface and attributes of its ancestor. It is possible to replace the descendant with any object.

The result of this design is a coherent set of objects, which allows the user to implement tests generally, and makes its changes easily.

2.1. Design concept

- Design is based on composition of complex parts from existing simple elements.
- Elements structure is not based on algorithms but on the data.
- Elements of designed structure have object representation. There are data and set of operations allowed on data in the object.
- Object in the program should correspond to an object in the real world, like signal source, filter, etc.
- Objects in application cooperate, one object calls services of the others or the object services are called from main program.
- The program is transparent, easy to modify and consists of simple operations.

The design of the system thoroughly benefits from the formerly explained methods. For example, GenericInstrument class consists of common properties of all used devices like input and output impedance and trigger settings. The GenericSource, GenericFilter and GenericDigitizer classes present the next level of abstraction of signal sources, analog filters and digitizers from which the real instruments' classes could be derived. If there is some need to build the more complex instrument driver in the plugin form, new classes could be created, describing this instrument, or multiple inheritance could be used.

All the methods in classes are declared as *virtual*, so their functions can be redefined on any lower level. It is very easy to transfer required information between trees of objects, in the designed object model.

The numeric data are stored in objects, for example IntegerArray or FloatArray. To identify the source and destination objects, simple pointers are assigned to these objects. For example, a pointer to the IntegerArray object is given to the objects, which are derived from GenericDigitizer object, and acquired data are stored in the field inside IntegerArray object. If an operation like DFT calculation should be done, the only necessary operation is to give the IntegerArray or FloatArray pointer to the AmplitudeSpectrum object and to call its Execute method. Using the same way, the data store is performed by passing the data pointer to the DataStore object, modifying a parameter to select a required file format and call the Execute method.

Basically, there are two ways how to write the particular implementation of the objects described above.

First approach: it is necessary for the user to know completely the structure of an object before its first usage; this means that the headers and import libraries must be distributed with them, and inclusion into project is performed during compilation, in C++.

Second approach introduces only limited application programming interface (API), which defines its own, object independent access to attributes and methods.

First approach results in modules, which are much easier to implement, because the values passed into an object might be type-checked, methods return desired values directly etc. The second approach: while API is minimal and as general as it is possible, its implementation is really hard, brings the necessity to add temporary variables, limit values, which may be passed in and out of the object.

The second way has been chosen, because it allows very good automation possibilities, and creation of simple system, which handles with objects.

After analysis of requirements and object-interface tests, just only 10 API functions are used: *PICreate* (creates new instance of object), *PIDestroy* (destroys instance of object), *PIInfo* (returns object descriptor), *PIStatus* (returns last error and current status of object), *PIGetProcID* (returns ID of procedure, i.e. method), *PIGetAttrID* (returns ID of attribute), *PIProcedure* (executes procedure), *PIAttribute* (sets or reads value of attribute), *PISave* (saves object) and *PILoad* (loads object).

The API installation is performed using the Win32 functions of the *LoadLibrary* and *GetProcAddress*. The first one loads DLL library, the latter returns a pointer to the appropriate function of the plugin interface.

API is exported as standard C functions, to allow the usage of plugins in ANSI C programs.

2.2. Plugins

DIGESTER plugin is an encapsulated entity, which implements methods like data acquisition, spectrum

```
      struct PI_API
      {
       PI_API_PICreate     PICreate;
       PI_API_PIDestroy    PIDestroy;
       PI_API_PIInfo       PIInfo;
       PI_API_PIStatus     PIStatus;
       PI_API_PIGetProcID  PIGetProcID;
       PI_API_PIGetAttrID  PIGetAttrID;
       PI_API_PIProcedure  PIProcedure;
       PI_API_PIAttribute  PIAttribute;
       PI_API_PISave       PISave;
       PI_API_PILoad       PILoad;

 }

   #define PI_INIT_API(nlib, hlib, apis) \
      HINSTANCE hlib;\
      PI_API apis;\
      hlib = LoadLibrary(nlib);\
      if (hlib != NULL) {\
            apis.PICreate = (PI_API_PICreate)GetProcAddress(hlib, "PICreate");\
            apis.PIDestroy = (PI_API_PIDestroy)GetProcAddress(hlib, "PIDestroy");\
            apis.PIInfo = (PI_API_PIInfo)GetProcAddress(hlib, "PIInfo");\
            apis.PIStatus = (PI_API_PIStatus)GetProcAddress(hlib, "PIStatus");\
            apis.PIGetProcID = (PI_API_PIGetProcID)GetProcAddress(hlib, "PIGetProcID");\
            apis.PIGetAttrID = (PI_API_PIGetAttrID)GetProcAddress(hlib, "PIGetAttrID");\
            apis.PIProcedure = (PI_API_PIProcedure)GetProcAddress(hlib, "PIProcedure");\
            apis.PIAttribute = (PI_API_PIAttribute) GetProcAddress(hlib, "PIAttribute");\
            apis.PISave = (PI_API_PISave)GetProcAddress(hlib, "PISave");\
            apis.PILoad = (PI_API_PILoad)GetProcAddress(hlib, "PILoad");\

       }
```

Fig. 3. Macro PI_INIT_API

calculation, and data store. These methods operate with the internal plugin data, describing plugin state, called *attributes*. The plugin is equipped with a standard API, which allows the access to the internal methods and attributes by unified way mentioned above. The access is common for all the plugins.

DIGESTER plugin is realized as a dynamic linked library (DLL), containing the application programming interface. Input parameters of the API functions are strings representing procedures and attributes, pointers to input or output values, and handles pointing to a particular instance of the object (each plugin might contain more than one instances of its internal object, addressed by special value filled by the *PICreate* function). It is possible to apply the same operations on several different instances of the same object by only the change of one input parameter.

The macro that loads the complete API by one command facilitates the usage of the plugin (see Fig. 3). The following code loads the *GenericDigitizer* plugin to memory:

```
PI_INIT_API ("GenericDigitizer.dll", handle_library,
                    api_struct);
```

The API is very simple. When a new plugin (which does not exist in the time of application building) for a new device is added to the existing application, there is no need to pass any special information to the application – plugin is identified simple way and the user decides only about its use.

The plugin can be easily recognized and with a special request, all its methods and attributes, with their description, could be read. There is no need for any previous information about numbers and types of attributes.

The plugin enables the use of more instances of the same internal object with only one parameter change. The internal plugin object is a C++ code, which implements the plugin methods and attributes. The API implementation and implementation of methods, which provide instance management, is partially separated from this object. The support for these methods must be included in the internal plugin object; therefore their calling is done by other methods, which are not included in this object. Assigned methods are exported in dynamic link libraries. Names of these methods are the same for each plugin, so it is possible to publish them the same way for all plugin.

2.3. Handles

When the plugin API is available, we need to distinguish each instance of the plugin's internal object. The instances are dynamically created by the user (main program) requests and their identifier is the pointer (called *handle*) to them.

This handle is the first parameter of all plugin API methods and is stored in a list created for this reason. The list enables an easy verification that given handle really belongs to the plugin (i.e. to prevent illegal memory access). The user must destroy a handle of an object after disposing the object's instance (using the *PIDestroy* function). It is also possible to remove automatically all internal objects from memory if the user did not remove them explicitly.

2.4. Examles of plugins

The GenericInstrument object is the basic class that implements general attributes of an instrument. All classes of real instruments are derived from this class. It does not contain any executive code to do any settings, but only the object mechanisms for implementing setting methods and parameters in derived classes. Using inheritance, the GenericDigitizer object is created (see Fig. 2). This object implements the most important common attributes of digitizers. The E1430Adigitizer object implements the real driver for the HP E1430 VXI digitizer. Common attributes and methods are defined in the parent classes GenericInstrument and GenericDigitizer. The E1430Adigitizer source codes are written using VISA library.

3. DBASIC

As mentioned above, programming using plugin API is very inconvenient. Thus a tool allowing simplification of development process was necessary to create. After discussing graphical interface, text-based simple programming language has been chosen. *DBasic* is a very simple, but fully featured programming language, specially designed for an easy implementation of tests, using the DIGESTER's plugins. DBasic is similar to the BASIC languages. It allows the usage of complicated mathematical and string operations, IF-THEN-ELSE conditions, FOR-NEXT loops and subroutine calls. But the main strength resides in the ability to cooperate with the plugins.

4. CONCLUSION

The DIGESTER II represents new generation of the software system for ADC dynamic testing fully developed and applied at our department. DIGESTER's plugins can be used either in C/C++ development environment (in form of dynamic-link libraries with unified API) or in the DBasic environment. The DBasic was created as the scripted language to facilitate the manipulation with the plugins.

REFERENCES

[1] IEEE Std 1241-Draft, "Standard for Terminology and Test Methods for Analog-to-Digital Converters", Version May 1999.

[2] IEEE Std 1057-1994, "IEEE Standard for Digitizing Waveform Recorders". The Institute of Electrical and Electronics Engineers, New York, 1994.

[3] J. Roztocil, J. Brossmann, "Software for Dynamic Testing of Digitizers". In Proc. RTUCET'01 Riga TU, October 2001, vol. 1, pp. 95-98.

[4] I.Kollar, J.Markus, "Sinewave Test of ADC's: Means for International Comparison". In Proc. IMEKO 2000, Vienna, September 2000, Vol. X, pp. 211-216.

Authors : Ing. Jiri Brossmann, Assoc. Prof. Ing. Jaroslav Roztocil CSc., Vojtech Ruml, Ing. Dusan Varga, Faculty of Electrical Engineering, Czech Technical University, Technicka 2, CZ 166 27 Praha 6, Czech Republic. Phone: +420 224352869, fax: +420 224352876, e-mail: roztocil@fel.cvut.cz