SELECTED ENGINEERING PROBLEMS
NUMBER 6
INSTITUTE OF ENGINEERING PROCESSES AUTOMATION
AND INTEGRATED MANUFACTURING SYSTEMS

Mateusz NOWAK, Andrzej BAIER

Faculty of Automatic Control, Electronics and Computer Science,
Silesian University of Technology, Gliwice
* matnow17@interia.pl

OBJECT-ORIENTED APPLICATION FOR ELECTRIC CAR'S DRIVE SYSTEM TESTING

Abstract: Research and simulation are important phases of the process of implementing new technologies. Proper test stands and testing procedures may provide significant reduction in time consumption and overall cost of the project. In addition to a configurable physical structure, the test stand has to be controlled by a flexible and modular application providing both ease of modification and safety. In the article, the concept of object oriented programming in LabVIEW is described and the proposed architecture for an application for electric racing car drive system testing is presented.

1. Introduction

The application is being developed within the Silesian Greenpower interfaculty student research project carried out at Silesian University of Technology. The project team designs and builds electric cars that take part in the annual Greenpower Challenge racing series. Research on the efficiency of the car resulted in building the test stand for complete drive system analysis, including the engine, the drivetrain, the car's wheel and the ground. With its frame disassembled, the test stand can perform tests on fully assembled car, so that the engine cooling can also be examined. Main task for the application combined with the test stand is to simulate the car driving on the real track. The simulation is based on the data gathered from the real tracks.

The system includes:

- Ethernet RIO Controller (1)
- SEW Eurodrive MDX61B power inverter with DFE11B extension card (2)
- Three phase asynchronous motor with absolute encoder (3)
- DC motor used in the car (4)
- The shaft (5) with built-in torque sensor (6)
- The wheel of the car (7)
- Drivetrain (belt) (8)

The program communicates with the MDX61B inverter via TCP/IP protocol using Modbus. Communication is provided by the DFE11B extension card. With the switch device added to the system, remote control over the test stand can be applied.

The simplified structure of the test stand is presented below in the Figure 1. Numeric labels used in the figure correspond to the list above.

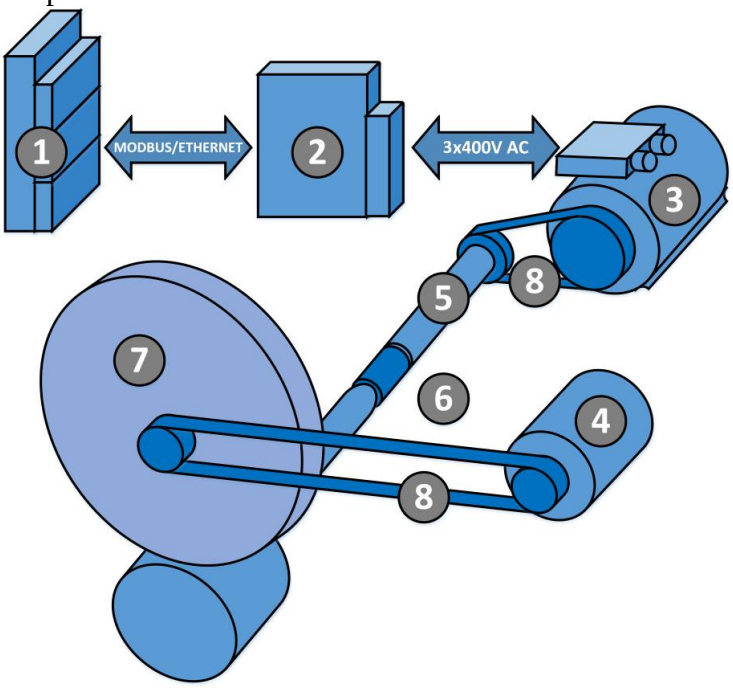


Fig. 1 Simplified structure of the test stand.

2. Application: language

The program has been implemented in LabVIEW graphical language delivered by National Instruments. LabVIEW code is generated by connecting the program blocks on the block diagram that represents the program's logic. It can be seen as an analogy for electrical circuit with wires forwarding the numeric values generated by blocks. Figure 2 presents an example code in LabVIEW.

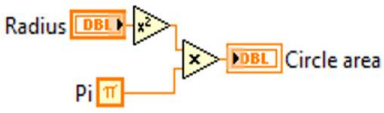


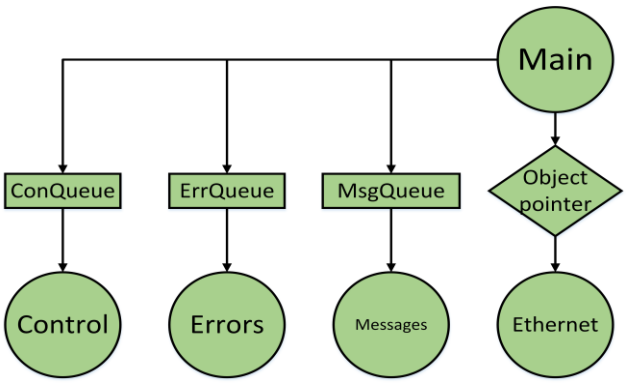
Fig. 2 LabVIEW code – circle area

2. Application: architecture

The application implements multithreaded architecture with threads separating the tasks within the program. Such approach is necessary in multitask programs that additionally

control actuators like DC motors – the program must handle user actions, data logging and visualization of the collected data while not affecting control or measurements tasks.

In the presented application, the communication between threads is realized via message queues introducing the producer-consumer programming model, as described in [1]. Commands for respective threads are send by the producer (in this case, the main thread). These commands are then interpreted by the consumer loops introducing another programming model, i.e. state machine presented in [2]. The consumer programs move within predefined operating modes where the choice of the next state depends on decisions made within the previous state. Thread termination is realized by sending the stop command to all of the threads – after receiving such command, the loop is stopped.



Fia. 3 Communication between threads

Respective tasks are divided between separate threads. Communication with the power inverter is realized in a single thread called *Ethernet*. Similarly, error handling is centralized within a single thread. All the errors are sent on a separate queue. Such approach simplifies the code assuring that a specific error is always handled in the same way and that no error is missed. Code replication is thus reduced. The relations between the threads are shown in the figure 3.

Application components generate messages which are then sent to a single thread that processes these messages and displays them to the user. All notifications generated in application are listed using dedicated front panel control and can be used as a program log. The user is informed about program actions like successful connection to the device with specified IP or successful initialization of the inverter. Implementing notification and communication routines in separate threads allows them to work without interfering with the control thread.

3. Main thread

The inverter is represented as an object – a data structure which definition will be explained in the next paragraphs. To assure failure-free operation, only one thread is allowed to modify the parameters of the object, for example to cancel the drive operation permission or to change the rotation speed setpoint. All the other threads are only allowed to read those parameters and adjust to their values.

Main thread is also responsible for handling user actions. The object associated with the controlled inverter is passed between consecutive iterations of the main thread. If the user

does not perform any action (timeout event occurs), the object is passed unchanged. If the user generates an event (like pressing a button), the object is modified and then passed forward. All the other threads will then read the modified object and adjust to the changes.

Behavior of all the other threads depends on the state of the object processed inside the main thread. Only this one particular thread should be able to act on that object to ensure that actions made within the main program loop will not be overridden. Such situation may result in damaging the test stand or hurting the operator considering the presence of actuators.

As an example the stop condition of the motor is presented. One of the object's parameter is a variable of logic type that represents permission for the drive operation. If the value equals to FALSE, the motor is allowed to run. After the operator decides to stop the motor, value of the parameter is changed to TRUE. As the main thread is the only thread allowed to perform that modification, we are sure that no other thread will accidentally allow the motor to continue operation. In the next iteration of the *Control* thread it will get the updated inverter object via the local variable mechanism and the command to stop the drive will be send to the *Ethernet* thread.

4. Safety measures

A command stopping the drive is always sent after the application startup. Controls on the front panel are locked, excluding the button stopping the drive. To unlock the controls and start the motor, user has to confirm that the test stand is safe. The purpose of such solution is to remind the user to ensure that no one is working with the test stand at the moment (test stand is equipped with a webcam monitoring the mechanical equipment). Only when user confirms that the test stand is safe to run, the controls on the front panel are unlocked.

The drive is always stopped when the program is started, the user shuts down the program or an error occurs. The user can stop the motor manually at any time using the program or the mechanical safety button.

5. Object-oriented programming

Object oriented programing is a model contrary to the procedural programming, where the program can be seen as a set of instructions and subroutines [3]. In object-oriented code, the application design is based on classes, which can be seen as an abstract structures describing particular problem or device [4]. The particular realization of the class – an object (also: an instance of the class) – represents the data on which the program operates and associated with an existing device. Object-oriented application consists of different objects interacting with each other. An example class implemented in the program is the class representing the MDX61B inverter.

In order to work, the object needs its methods – a set of subroutines that can be executed on the object of the specific class [5]. The most basic methods are those accessing data (variables of the object structure are not explicitly available in the main program). Example methods of the MDX61B class are used to initialize the device after program startup and to change the speed setpoint.

6. Class inheritance

In the described program each physical device has its own dedicated abstract class. For instance, there is a class representing a generic inverter device. Its model assumes the basic variables describing the device like device name, IP address, speed limitation, operation permission and so on. Those fields are included in the object. It also implements the fundamental methods – device initialization, data access and setpoint modifiers, as such functionality is required in this type of devices.

There is also a more specific class - the class of the MDX61B inverter. The basic idea of class inheritance is that the inheriting class includes (inherits) all the data and methods of the parent class plus extensions applied to that specific class, as presented in [4]. Thus, the code common for all objects (the parent class and inheriting classes) does not have to be replicated for all of the inheriting classes. When a specific method is called, LabVIEW checks if the current class implements this method. If it does, the method of the class is executed. If it does not, LabVIEW checks all the direct parents in the inheritance branch recursively and executes method of the first parent that implements the method. If there are many devices that need to be initialized in the same manner, startup method is implemented in parent class and then inherited by more specific classes. There is no need to define this method for all of the devices separately.

On the top level of the inheritance diagram there is always the most generic class [4]. It implements methods used by all the inheriting classes. With each level of the diagram, the classes become more specific. An example inheritance diagram is showed below in the figure 4. The green cube represents the general class. The red and the blue ones represent more specific, inheriting classes.

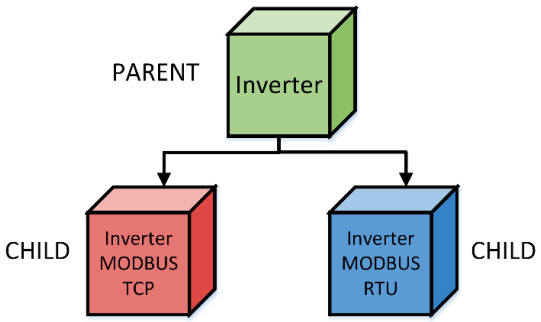


Fig. 4 Example inheritance diagram

Application functionalities are implemented as object methods, a universal set of blocks responsible for specific actions. Those blocks can be used to form an application which will use the same code to behave differently based on the currently handled class (device).

7. Advantages of object-oriented programming

Main advantage of object-oriented programing apart from efficiency improvement and better code organization is that the program can switch to work with new device model without changes in the code [3]. The flowchart is defined for a generic object (abstract inverter), and not for each device separately. Control of a specific inverter is provided through methods depending on the object passed to the input of the control loop. In the procedural

programming, a separate control loop should be created for each device and the program would switch between them. This increases the time needed to develop the program and forces to replicate the code. LabVIEW makes it possible to define an array of objects and control several devices from within a for loop, with only slight changes made to the program code.

Object-oriented approach also makes the program less dependent on the device model used on the test stand and makes it possible to exchange the device for another model with only a little work needed for the application to resume operation. Adding another devices or modifying the code shared by similar devices is far less time consuming.

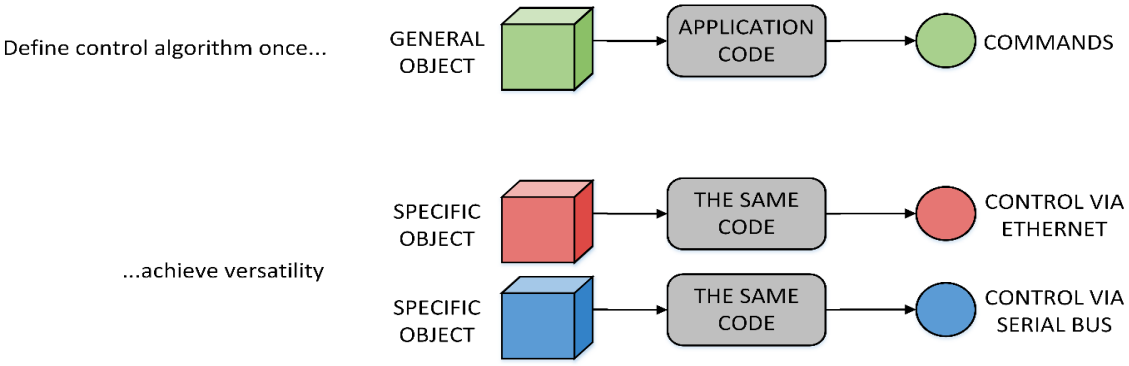


Fig. 5 Advantage of objectivity - versatility of the program

8. Conclusions

The article presents main principles and major advantages of object oriented software in research applications. Object-orientation of the code resulted in an adjustable structure of the application that is easily extendable with support for new devices. Time needed to develop the application was significantly shortened by reducing the code replication.

Proposed architecture introducing producer-consumer programming model allows to build a reliable and versatile applications capable of carrying out various tasks simultaneously. As LabVIEW queues buffer the data, there is no risk of data loss, which results in reliable operation.

Different program solutions introduced to provide safety of the program operator were also described.

References

1. National Instruments Application Design Patterns: Producer/Consumer: http://www.ni.com/white-paper/3023/en/
2. National Instruments Tutorial: State machines http://www.ni.com/tutorial/7595/en/
3. National Instruments LabVIEW Object-Oriented Programming FAQ http://www.ni.com/white-paper/3573/en/
4. National Instruments manual: Creating LabVIEW Classess http://zone.ni.com/reference/en-XX/help/371361J-01/lvconcepts/creating_classes/
5. Bitter R., Mohiuddin T., Nawrocki M.: LabView: Advanced Programming Techniques, Second Edition, CRC Press 2006, p. 449