A Software Control Model for Industrial Interaction

Marcelo Akira Miho
Jorge Anthony Félix Herrera
Dalton Matsuo Tavares
José Martins Júnior
Glauco Augusto De Paula Caurin
Escola de Engenharia de São Carlos
Universidade de São Paulo
P.O. Box 668 – 13560-970 – São Carlos – SP
BRAZIL
Tel: +55 16 2739399 Fax: +55 16 2739402
{akira, jfelixhe, dmatsuo, jmartins, gcaurin}@sc.usp.br

ABSTRACT

Control trends in industrial or robotic environments are currently based in encapsulated controlling systems. These systems encapsulate control and response functionality for a well-known scenario and, in the majority of cases, their configuration need to be flexible beyond the initial system specifications and requirements. Software components are being presented as a possible solution to develop reliable and reusable components in a generic controller environment. This paper explores this approach proposing a layered schema.

KEY WORDS

Keywords: Computational Control; Control Modeling; Generic Controller, System Integration

1. INTRODUCTION

Industrial automation uses a large collection of equipments and hardware [1] for control purposes. These equipments and components provide digital interface capabilities to acquire and interpret sensorial data from environment as well as activating a set of predefined responses using programmed embedded rules. Most generally, we can catalog some device types present in the industrial environment: analog input/output (I/O), digital I/O in a local scope (e.g. a serial port limited to some few meters) and network-plug ready devices with communication modules implementing sophisticated network medium access protocols such as CSMA/CD [8] to access a major control or data network.

This variety of medium-dependant interfaces introduces complexity in the data acquisition and control design projects because developers and integrators must provide specific solutions for every device or device group present in the system.

This work presents an approach to enforce the integration between control software and physical devices in an industrial environment. We propose the inclusion of an intermediate layer between physical devices and software programs to interact with the devices. This layer contains a set of components that implements software interfaces and can be accessed by the control program running in a local or remote address space [11] using a common paradigm.

2. PHYSICAL DEVICE ABSTRACTION LAYER

Device functions and environment sensorial data acquisition and response can be abstracted by using a set of rules to control I/O requests and encapsulate it into a generic component called Physical Device Abstraction Layer (PDAL). Some of the most generic hardware abstraction layers such as WinNT HAL [3] or RTLinux Kernel [4] implements basic I/O functionalities at medium access level while PDAL is designed to provide easy device interaction to the developer through a set of API (Application Program Interface) functions.

Two elements compose the PDAL: a Pluggable Device- Driver Bay (PDB) to allow specific I/O software driver component plugging. This module encapsulates all I/O operations and controls low-level data transfer. A second element is the Generic Device-Controller Library (GDL) to provide a set of functions and procedures to allow user program interaction with this layer.

We abstract a specific device by establishing a generic input/output operational semantics [5] that can be accessed via analog, digital, and network interfaces. Analog interfaces will be treated in the same way as digital interfaces in this article because the inclusion of secondary analog/digital conversion is necessary to exchange data with these devices. Latency introduced by this conversion process must be rewarded to maintain acquisition stability.
PC external interfaces includes data acquisition cards operating via Direct Memory Access (DMA), serial or parallel ports and network interfaces, thus, these three types of interfaces are good candidates to be implemented into the PDB scope (see Figure 2). Next, we examine each interface type and the necessary tasks to implement an appropriate driver for it.

For data acquisition cards, DMA interaction implies in fast memory access attempting the microsecond’s time range. Complexity in this case is introduced by requiring synchronization of the memory variables used into the control program being executed by the central processor [6]. The device driver must provide mutual exclusion mechanisms to avoid overlapping operation in the short time elapse between data arrival. Read and write operations are realized in the PC memory and data bits are extracted or deposited in regular time intervals determined by the data card timer capabilities. Latency is generally not a problem in this case because the device state is updated continuously and independently of the software interaction.

Some acquisition cards provide particular set of API functions that encapsulates variable exclusion. Semantic analysis must be considerate exclusion bridging to avoid common problems such as deadlocking. Our experience shows that manually exclusion is the most viable approach to better memory interaction.

Serial ports [7] are slow and require buffering capacity to avoid data lost, since computational cycles are too much faster that serial interaction, if multiple serial channels are present, it implies in individual maintenance context including the current state of each channel being managed. It may not be a complex task if the communication protocol is the same for each device or if all the devices are of the same type. Complexity is introduced when multiple devices with multiple serial protocols must be controlled, because each of them will require a different driver.

The adoption of a second strategy to work around the problem is necessary. The solution relays on the development of a universal serial interface to delegate protocol control to the specific drivers. Main function of this interface is the continuous data control for each multiplexed channel under management, every time that specific protocol semantics is encapsulated in the individual device driver.

Finally, devices with real networking capabilities can be considered more robust in a communication technology context. This type of device can be directly plugged in the industrial or local area network and all specific data transfer complexity is delegated to internal communication components attached to the device. The network provides natural multiplexing capabilities to this devices in a directly control scope. In this case the device driver implements a connection point to establish a “virtual” session with the device through TCP or UDP [8], for example.

Field network types supporting industrial interactivity, where the medium access control is not accessible by the Soft-PLC will require an appropriate bridge to data bypassing. This may be generally accomplished by third part equipments present between two network systems such as master PLCs or a centralized server.

3. STRUCTURE OF A PDB DEVICE DRIVER

A PDB Device Driver is composed by a specific communication protocol controller for each device type. It implements adequate end points for a given device communication protocol, buffer management and asynchronous I/O control [9].

Driver structure is composed by a global information table exposing timers granularity and other capabilities information, a device information section to inform to GDL about the nature of discrete elements under their
command (e.g. the number and type of exposed variables for a given device), an internal engine to synchronize data arrival and sending from device to physical ports and a set of functions that allows GDL to transform user API calls (see GDL Library in the paragraph 4) in adequate device start/stop/read/write/alert operations (shown in the Figure 3).

### Semantic Operation

<table>
<thead>
<tr>
<th>Semantic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>Creates a virtual representation of a communication channel between program and device.</td>
</tr>
<tr>
<td>Binding</td>
<td>Associates a variable with a current state of the device and can be single or bi-directional.</td>
</tr>
<tr>
<td>Read/Write</td>
<td>Send or receive an amount of discrete data asynchronously.</td>
</tr>
<tr>
<td>Alert sink</td>
<td>Associates a specific action with a state of the device (e.g. a sensor to angular constraint checking).</td>
</tr>
</tbody>
</table>

Table 1 - Semantic operations in the GDL

Figures 3, 4

The GDL uses these primitives to implement the semantic operations (described in the Table 1) and is responsible for I/O synchronization performing the timer granularity information obtained with the GetTimerInfo function. Every I/O interaction at this level is inherently asynchronous, since the SetIOOverlapped implementation provides a standard way to request and send information to the physical port. The internal driver engine encapsulates the device communication protocol synchronizing internal buffers with the global variables exposed in the driver.

4. **THE GENERIC DEVICE-CONTROLLER LIBRARY**

The Generic Device-Controller Library (GDL) is a procedure set that allows the realization of the following functions by the control program: establish a session with a given device, exchange data with it, define event and alert protocols and clear a current session. Exchange data with the device in a standardized way is the most important element present in this component because it allows abstraction of the control modeling.

The GDL is composed by an I/O library for data exchange tasks, an event synchronizer to receive external messages and alerts and an I/O synchronizer (see Table 2). The Control program uses these facilities to communicate with the device in a standard way and its structure is the same even if current semantics is different for each device type. Table 1 shows principal semantics and operation equivalences implemented within this component.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDeviceInfo</td>
<td>Generic device info such as manufacturer and driver version.</td>
</tr>
<tr>
<td>GetDeviceVariableList</td>
<td>Obtains a set of variable names and types statically defined by the given driver.</td>
</tr>
<tr>
<td>GetTimerInfo</td>
<td>Gets time operation capabilities for I/O ports.</td>
</tr>
<tr>
<td>SynchronizeDevice</td>
<td>Initialize I/O states for this device.</td>
</tr>
<tr>
<td>SetIOOverlapped</td>
<td>Associates device variables to program memory variables.</td>
</tr>
<tr>
<td>GetDeviceTimer</td>
<td>Obtains an internal timer reference for thread context association.</td>
</tr>
</tbody>
</table>

Table 2 - Functions

I/O library directly interacts with the PDAL “passing to” and “receiving from” data chunks through operational variables allocated within the control programs. It can be viewed as a “data bridge” and their synchronization is managed through the specific device driver implementation. The event synchronizer role is a passing alert data in an asynchronous way. The PDAL allows the definition of “connection points” to catch events from the devices or from the environment. The developer must provide specific routines following an interface contract specified by the layer. Session state for each channel is stored in a connection pool to prevent disconnection or emit an alert.
Currently, a systems integrator needs to define the application scope and manually synchronize all the operational aspects for a given device. With this approach we expect to give an improvement to the control programming tasks by centralizing the most complex task in data acquisition and device command sending.

5. API FOR PDAL INTERACTION

API functions that interact between control programs and PDAL specifies a mechanism to create a logical representation for a device connection, called “virtual device connection”. Input and output operations can be performed using this connection through a set of functions showed in the Table 3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapDevice</td>
<td>Creates a logical representation for a given device.</td>
</tr>
<tr>
<td>CreateConnection</td>
<td>Establish a virtual connection with a “mapped” device.</td>
</tr>
<tr>
<td>BindDeviceVariable</td>
<td>Maps a protocol-dependant device variable, creating an internal buffer for data transfer.</td>
</tr>
<tr>
<td>CloseConnection</td>
<td>Clear all resources associated to a device connection.</td>
</tr>
<tr>
<td>SendRawByte</td>
<td>Sends a byte via specific virtual variable addressing.</td>
</tr>
<tr>
<td>ReadRawByte</td>
<td>Read a current variable byte representation.</td>
</tr>
<tr>
<td>GetVariableMutex</td>
<td>Obtains a mutex (mutual exclusion object) for a previously bind variable.</td>
</tr>
<tr>
<td>EntryMutualSection</td>
<td>Turn on mutual exclusion for variable access.</td>
</tr>
<tr>
<td>LeaveMutualSection</td>
<td>Turn off mutual exclusion.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapDevice</td>
<td>Creates a logical representation for a given device.</td>
</tr>
<tr>
<td>CreateConnection</td>
<td>Establish a virtual connection with a “mapped” device.</td>
</tr>
<tr>
<td>BindDeviceVariable</td>
<td>Maps a protocol-dependant device variable, creating an internal buffer for data transfer.</td>
</tr>
<tr>
<td>CloseConnection</td>
<td>Clear all resources associated to a device connection.</td>
</tr>
<tr>
<td>SendRawByte</td>
<td>Sends a byte via specific virtual variable addressing.</td>
</tr>
<tr>
<td>ReadRawByte</td>
<td>Read a current variable byte representation.</td>
</tr>
<tr>
<td>GetVariableMutex</td>
<td>Obtains a mutex (mutual exclusion object) for a previously bind variable.</td>
</tr>
<tr>
<td>EntryMutualSection</td>
<td>Turn on mutual exclusion for variable access.</td>
</tr>
<tr>
<td>LeaveMutualSection</td>
<td>Turn off mutual exclusion.</td>
</tr>
</tbody>
</table>

Table 3 – GDL API Primitives

A typical program must execute the following steps to control a device: first, the device must be mapped. This means that a memory representation must be created using the MapDevice function:

```c
HDEVICE hDev = MapDevice(
    SERIAL_DEVICE,
    “SMS_PROTOCOL_V01”,
    BYTE_ALIGN, NULL);
```

This virtual connection allows the program access to the device process information, in the sample above, a serial connection via RS-232 is established. Next, it is necessary to define the device variables to create input/output connection points:

```c
static BYTE byDataTransfer;
BOOL fBinding = BindDeviceVariable(
    hDevConn,
    &byDataTransfer,
    TIME_SYNC_VARIABLE,
    READ_ISOLATED,
    “SENSOR_R3:0001”,
    100, // milliseconds
    NULL); // Callback function
```

This piece of code means “bind the variable ACTUATOR_R3 from the device-dependant model to an 8-bit memory variable and synchronize (read) the value every 100 milliseconds” according to bind semantic showed in the Table 1. The following example shows how to bind an alert variable to execute an asynchronous function if the values indicate an exception:

```c
static BYTE byDataTransfer;
BOOL fBinding = BindDeviceVariable(
    hDevConn,
    NULL, // not explicit variable
    ASYNC_VARIABLE,
    READ_ISOLATED,
    “SENSOR_R6”,
    10, // milliseconds
    TestVariable); // Callback
```

This sentence test the SENSOR_R6 variable every 10 milliseconds and evaluate it via the TestVariable callback function. SendRawByte and ReadRawByte are auxiliary functions to send and receive independent protocol 8-bit variables respectively. Both functions are useful to test device communication in earlier development stages of the control programs.

6. SYNCHRONIZATION ISSUES

Due the short time intervals needed to execute a routine to control a set of devices, the PDB creates a “device control context” for each virtual connection between the user control program and physical device. In the variable binding case, when the variable need to be updated in every few milliseconds, a secondary timing queue is created in the context thread ([10] and [11] provides a good discussion about this topic). The PDAL manages these internal structures to hide synchronization complexity to the programmer.
In the Figure 5, it is possible to observe two connection contexts (for two different devices) with two variables and one callback function associated within same user control program.

Figure 5 Variable binding and events synchronization

To avoid variable manipulation race condition, a mutual exclusion object must be requested via the GDL library (Table 3):

```c
static BYTE byDataTransfer;
BOOL fBinding = BindDeviceVariable(
    hDevConn, &byDataTransfer, ...);
DEVICE_MUTEX mutex;
GetVariableMutex(
    &byDataTransfer,
    &mutex);
// Variable modification
EntryMutualSection(&mutex);
byDataTransfer = ...;
LeaveMutualSection(&mutex);
```

This synchronization steps are not required for callback binding and the mutual exclusion is activated before and deactivated after callback function execution by the PDAL.

7. MODEL OPERATIONAL TEST

To validate this presented model, a PC-based test environment using the following configuration was build:

- Athlon XP with one 1.7 GHz processor.
- Linux operating system with Kernel 2.4.10 with real-time kernel support.
- glibc libraries version 2.2.5 with shared memory support.
- Build environment: gcc, build, make and debug utilities.

Experimental configuration was enabled using one sensor and one actuator simulation equipment connected to an controlled experimental data acquisition environment, as showed in Figure 6:

Figure 6 Experimental PC-based Environment.

Normalized variables tested in the experiment were angular speed, position and acceleration, depicted with SPEED and POS acronyms.

In the Figure 7, we can see a PC-based and an conventional controller numerical comparison for angular speed. Standard deviation for this first comparative experiment was 0.0149 rpm.

Figure 7 Nominal vs. PC-based rotational speed.

Figures 8 and 9 reflect the controlled angular position for nominal and experimental cases and numerical deviation between both systems.
Experimental results show a good approximation for nominal values provided in several simulation environments. Standard deviation rates for position controlling are in the order of 0.05 °.

8. CONCLUSIONS
We presented an implementation model to allow the control of heterogeneous devices by defining architectural patterns for physical device and program control interaction. The “virtual device connection” and “control device context” were discussed as a form of abstraction to reduce the number of semantic operations to be implemented in the Physical Device Abstraction Layer (PDAL) model. PDAL structural division tries to generalize the user interaction with the devices by establishing a standard programmatic paradigm to avoid the complexity of real-time control development.

The model concentrates efforts in the development of specific device drivers for each device type to be controlled. Synchronization issues are highly dependant of the operational system being used because kernel implementation differences and practical tests are being performed to determine the better environments for the model implementation.

Experimental results enforce the viability of PC-based control systems into industrial applications using a layered approach such as presented PDAL.

Future extensions of the model are projected to include a declarative language common to PLC programmers based in the semantics presented in the GDL Library section and a set of development patterns to help in the control programs write.

8. REFERENCES