An Object-Oriented Layered Approach to Interfaces for Hardware/Software Codesign of Embedded Systems

W. B. Gardner and M. Serra
VLSI Design & Test Group, Dept. of Computer Science, Univ. of Victoria, Victoria, B.C., Canada
wgardner@csr.uvic.ca, mserra@csr.uvic.ca

Abstract
The Canadian Microelectronics Corporation\(^1\) has developed and distributed a Rapid Prototyping Board (RPB) to facilitate research in Hardware/Software (HW/SW) Codesign, case studies, applications and prototyping of projects in embedded systems. This research develops a series of layers between hardware and software, exploiting the dynamically reconfigurable hardware of the RPB and creating the connection to host processes and software layers in general, both on the board itself, and between a HW/SW system downloaded to the board and its host workstation. In this paper we describe a new approach which uses object oriented technology as the basis for the system design methodology, the specifications and the implementation, providing a flexible and dynamic foundation, lending itself to further expansion and research in HW/SW codesign.

1. Introduction

Canadian universities have recently received a new tool for research in configurable computing, the Rapid-Prototyping Board (RPB) [2]. The board features an MC68306 16-bit microprocessor (MPU) with 2Mb of RAM, and two Xilinx XC4010 reprogrammable FPGAs, each connected to 128Kb of RAM and a variety of off-board connectors (see Figure 1). Uses envisioned for this research tool...
include prototyping of embedded system designs, experiments in hardware/software codesign, and automated testing of FPGA-based circuits. Since the MPU has large onboard RAM (expandable to 16Mb), dynamic reprogramming of the FPGAs from a number of stored configurations is also possible.

The RPB, however, is not yet easy to use. Aside from the hardware synthesis tools (we use Mentor Graphics SDS for high-level design entry and Synopsys FPGA Compiler for VHDL synthesis), the GCC cross-compiler, and a primitive ROM-based monitor/ debugger, there is nothing that can be called “configware” which could aid an application in managing the interface between its software partition and the configurable hardware. Therefore, we have set out to build up layers of software that can form the foundation for more elaborate projects.

After giving a brief background on codesign, this paper describes the first two layers in our anticipated suite of configware for the RPB:

- a driver for the FPGA which covers the hardware with an object-oriented software interface; and
- a distributed communication gateway that transparently exports the above interface to software running on external hosts.

We close with some discussion of the next layer that we would like to build for our codesign research.

2. Background

The new area of hardware/software codesign encompasses a number of concepts and tasks [6] [8]. We look at the cooperative design of hardware and software components of a system and at a possible unification of the currently separate hardware and software paths. Also we often need to explore the movement of functionality from hardware to software and vice versa, with platforms to support research and simulation. The main goal remains to meet system-level objectives by exploiting the synergism of hardware and software through their concurrent design.

In codesign, designers consider trade-offs in the way hardware and software components of the system work together to exhibit a specified behavior, given a set of goals and an implementation technology. This methodology demands a more flexible design strategy, in which software and hardware designs proceed in parallel, with feedback and interaction between the two.

Although such problems have been tackled before, some recent developments have stimulated interest in this topic. First, systems now deliver much higher performance, requiring architectural support for operating systems or particular hardware features to expedite the application software. At the same time, cost has brought us to exploit the programmability of software components of digital systems to support product evolution. Secondly, new architectures based on programmable hardware circuits can accelerate the execution of specific computations, or emulate new hardware designs. Configuring and executing programs on these architectures exploits the synergy between hardware and software. Finally, recent progress in synthesis and simulation tools for hardware circuits has paved the way for integrating CAD environments for codesign. Our research project falls in this last category.

Embedded systems are application specific systems which contain hardware and software tailored for a particular task, generally part of a larger system, and based on a microprocessor [3]. Embedded systems often fall under the category of reactive systems, which continuously interact with humans and their environment, with real-time constraints usually forming a main consideration. As the complexity of these systems increases, the issue of providing design approaches that scale up is essential; yet current tools, while quite sufficient for hardware (e.g. CAD in general) or software (e.g. some CASE tools), are not so for codesign.

The architectural template of a typical rapid prototyping system is a distributed structure consisting of a single-board computer and custom boards, where a real-time operating system is run. Rapid prototyping attempts to shorten the design path from a system specification to a working prototype using automated synthesis techniques, but these techniques do not answer basic functionality questions as to the optimal partitioning. How this decision is made is an interesting research topic, which needs to be supported by a smooth framework, with flexibility and ease of use. This leads to an active new area of research labeled “configurable computing” or configware [7]. Configware is based on a combination of programmable hardware devices and high performance microprocessors, and all the interactions and interfaces between them and the application software or external operating system.

In this research project we construct the basic framework for more advanced research in prototyping embedded systems through a codesign methodology, by providing an object-oriented approach which is extensible to a wide set of environments.

3. IntraGiz: an object-oriented FPGA driver

The RPB was developed by the Department of Electrical and Computer Engineering, University of Toronto, under the auspices of CMC. The MPU monitor prompt is “Gizmo2>”, the name Gizmo being a holdover from a 68000-based processor board used in their computer archi-
architecture courses. This explains the “Giz” in our names Intra-Giz and Gizgate below.

In designing the board, the engineers had to make choices about the means of connecting the FPGAs and the MPU, and these choices—being hardwired—have implications for configware. Figure 1 shows that the FPGAs are connected to the MPU essentially in series. This has the effect of maximizing the number of I/O pins on FPGA1 that can be directly controlled by the software, either via the 16-bit MPU bus (plus 4 address lines and 2 chip select lines) or 14 of the parallel port lines, as shown in Figure 2 and Figure 3. In addition, FPGA1 can interrupt the MPU.

![Figure 2. MPU-to-FPGA bus connection](image)

**Figure 2. MPU-to-FPGA bus connection**

![Figure 3. MPU-to-FPGA parallel port connection](image)

**Figure 3. MPU-to-FPGA parallel port connection**
FPGA1 can, if desired, be configured to simply pass some MPU signals through to FPGA2, or a large circuit can be partitioned between the two FPGAs, with FPGA1 acting as the MPU interface.

From the software standpoint, then, the first issue to face is how to conveniently manipulate the MPU’s bus-related lines and parallel port lines so as to achieve reliable communication with FPGA1. This can be done directly in C or 68000 assembly code by means of memory-mapped I/O; however, this approach is fraught with snares for the unwary. Apart from the fuss of bit-configuring Chip Select Registers (i.e., the 68000 memory map and DTACK regime), Port Direction Registers, the System Register, and the Interrupt Configuration Register, some of the I/O lines (chip select and address) must be set indirectly by calculating offsets for C pointers. Mistakes in addressing cause the MPU to hang.

The situation with the parallel ports is similar. Masking and shifting are required in the port registers to set/read specific lines that match the FPGA pinout. Input vs. output direction can be configured on a bit-by-bit basis, and mistakes can damage the hardware (though the RPB developers reduced this risk by placing buffers between the MPU and FPGA pins). Furthermore, since some FPGA pins are potentially multiple use (see Figure 3), it is possible to make errors in parallel port pin allocation for I/O use that conflict with JTAG port or FPGA programming use. This is also the case when using parallel port lines for interrupts—the same lines can be used for normal input.

Such pitfalls are no encouragement to developing projects for the RPB, and it is not reasonable to expect a user to grub about at this level. These considerations thus echo the usual motivations for creating a layer of abstraction in software, in effect, an FPGA driver. The structure and use of our driver will now be described.

### 3.1. IntraGiz objects

Taking an OO approach, we appropriated objects that occur naturally in the problem domain: the bus, parallel port pins, and interrupts. The C++ class hierarchy of IntraGiz is shown in Figure 4. The specific public classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Function</th>
<th>Syntax (obj = class instance)</th>
<th>Arguments / Comments</th>
</tr>
</thead>
</table>
| FPGAbus    | Output   | `obj(s,a) << data;`          | chip select (CS1/CS2): s = \{0,1\}  
address: a = \{0..15\}  
16-bit data  
location loc |
|            | Input    | `obj(s,a) >> loc;`           |                      |
|            | Set DTACK regime | `obj.extDTACK(w);`  
`obj.autoDTACK();` | w wait states |

*italics* = protected base class

**Figure 4. IntraGiz class hierarchy**

(which the programmer would find in file “IntraGiz.h”) are detailed in Table 1 through Table 4, with their methods grouped according to functionality.

### Table 1. IntraGiz bus I/O classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Function</th>
<th>Syntax (obj = class instance)</th>
<th>Arguments / Comments</th>
</tr>
</thead>
</table>
| FPGAbus    | Output   | `obj(s,a) << data;`          | chip select (CS1/CS2): s = \{0,1\}  
address: a = \{0..15\}  
16-bit data  
location loc |
|            | Input    | `obj(s,a) >> loc;`           |                      |
|            | Set DTACK regime | `obj.extDTACK(w);`  
`obj.autoDTACK();` | w wait states |

### Table 2. IntraGiz parallel port I/O classes

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Function</th>
<th>Syntax (obj = class instance)</th>
<th>Arguments / Comments</th>
</tr>
</thead>
</table>
| InputPin   | Constructor | `InputPin(pin);` | pin = \{PA0, PA1, PA4, PA5, PB4, PB5,  
PB6, PB7\} |
|            | Input    | `obj.level()` | returns pin level 0 or 1 |
Note that all I/O directions are specified relative to the software’s viewpoint. That is, “output” means from the MPU to the FPGA. Regarding port pin names, MPU nomenclature is used for general I/O pins. This matches the table of “Hardwired Microprocessor Connections” in the RPB Users Guide [2], which must be referred to when
assigning circuit ports to FPGA pads for synthesis under Synopsys. But when specialized pins are referenced in the TestPins, ProgPins, and FPGArupt constructors, the corresponding functional names are used (e.g., IRQ3 rather than PB5, or TDI instead of PB0).

In order to employ any of the FPGA connections represented by these classes, a programmer simply instantiates the appropriate object, and the constructor code will perform the needed initialization of MPU registers. The FPGApin base class contains the global pin allocation table. Its protected methods are invoked by subclass constructors and destructors to allocate and deallocate pins. The public object methods are then used to transfer data between C++ variables and the FPGAs.

In thinking about what classes and methods to create, we had to consider what styles of I/O the software partition might wish to implement with the FPGAs. Because FPGAs are “soft” and can potentially represent any hardware, this is a different problem not encountered in typical device driver design, say for a printer or disk drive. We decided that, rather than limit the user’s options by imposing predetermined semantics with regard to FPGA I/O, we would provide primitive operations that are close to the hardware. The user can then choose between direct use of the primitives alone, or “layered use” based on creating higher level abstractions which in turn use the direct methods. Examples of both uses are given in Section 3.2 and Section 3.3.

Below, then, we list the basic capabilities provided by IntraGiz.

### 3.1.1. 16-bit bus cycle transfer (FPGAbus class)
This mode requires the FPGA’s cooperation, in that it must contain a circuit capable of handshaking with the 68000 bus cycle when a chip select signal is asserted. An example of its use would be to pass numeric arguments to a coprocessor and retrieve the results. A further 5 bits of data are transferred on the 4 address lines and 2 chip selects (since they are mutually exclusive, they only count as one data bit), allowing up to 20 data bits per cycle.

#### 3.1.2. Single-pin and multi-pin parallel port I/O (InputPin[s] & OutputPin[s] classes)
To use these classes, the user must instantiate pins explicitly by name (e.g., PA0, PB4) as either input or output.2 The constructors validate pin direction and check for overlapping allocations. The methods take care of masking/shifting in the port registers. The “plural” classes, InputPins and OutputPins, provide a bundling service so that a group of pins, which need not be contiguous in the port registers, may be conveniently used to transfer an n-bit quantity, such as a function code or completion code. These classes will likely find their use for flags, clocks, or as primary I/Os when the FPGA is being treated as a block of combinational logic.

#### 3.1.3. Interrupts (FPGArupt class)
Instantiating this class converts the input pin corresponding to the specified IRQ level over to interrupt use. The constructor establishes an autovector that invokes the supplied user function when the FPGA interrupts the MPU.

#### 3.1.4. Special pins (TestPins & ProgPins classes)
The pins controlling special functions—the JTAG test port3 and reprogramming—are guarded by these classes. Their constructors check to make sure these pins have not already been allocated for I/O use.

#### 3.1.5. Error handling
All the above facilities are accessed using the GCC cross-compiler. The C++ exception mechanism (g++ 2.7.2 or higher) is used to deal with software usage errors: invalid pin direction, pin or IRQ level in use, arguments out of range. This helps to ensure correct use of the interface by software.

A sample of code used for debugging purposes might look like this:

```cpp
try{
  FPGAbus bus;
  OutputPin flag( PA3 );
  InputPins code( PB7 | PB6 | PB5 | PB4 );
  bus(select, addr) << value;
  flag.set();
  code >> newCode;
}
catch( FPGAerror err ) {
  print( "\nCaught FPGAerror! " );
  print( err.explain() );
  print( "\nBailing out\n" );
}
```

In this sample, the FPGAerror exception would be thrown if any of the named pins were invalid (e.g., PA3 is not wired for output), or if the arguments select, addr, and value were too large.

### 3.2. Example of direct use
In this simple scenario, the user has synthesized a VHDL program (a traffic light controller) on the FPGA, and wishes to use the MPU to drive the circuit with test vectors. Output is observed on the red and green LEDs wired to FPGA2. The source code in Figure 5 shows how several OutputPin objects are instantiated, and then used to drive the circuit’s primary inputs while toggling the clock.

---

2Even though Xilinx FPGAs support bidirectional I/O, the buffers on the RPB effectively hardwire the directions of the pins.

3Use of the FPGA JTAG port requires placing special attributes in the synthesis step, otherwise these pins function as normal I/O regardless of the use of the TestPins class.
The original version of this code which formed part of an RPB training course [1], written without IntraGiz, used bus I/O even though no handshaking circuits were created in FPGA1, likely because parallel port I/O (the natural choice for simply driving primary inputs) was so troublesome to program. It relied on bit operations with memory-mapped I/O through C pointers, was therefore less intuitive to understand, and in fact configured the Chip Select Registers with an erroneous (nonfatal) value. In contrast, we consider the IntraGiz version superior both in principle (i.e., using the parallel port) and for readability, and it leaves the programmer no opportunity to make mistakes, given only that the correct output pins are designated.

3.3. Example of layered use

More complex hardware configurations may be advantageously wrapped in a higher-level functional abstraction, to purposely hide the FPGA I/O and/or to ensure that the protocol is correctly carried out. In that case, we can encapsulate IntraGiz objects inside another class, either by membership or inheritance. This example shows how one could define a class to represent a hardware accelerator configuration that implements the CORDIC algorithm. In the source code (see Figure 6), the class definition would be provided by the creator of the hardware configuration. The user instantiates the CORDIC class and uses its trigonometric methods in a natural way without being concerned about the FPGA I/O that is going on. The CORDIC class methods in turn carry out a multiphase protocol involving shipping the argument and a function code to the FPGA (bus transfers), looping until a completion flag is set (port pin PB0), then retrieving the result on the bus.

3.4. Discussion

Here we comment on the efficiency of using the IntraGiz layer for I/O, and the semantics of multiple instances of IntraGiz objects.

3.4.1. Efficiency. Apart from the constructors and destructors, the actual I/O methods are just a short line of C++ code, representing a handful of machine instructions. Multiple-pin methods have a packing or unpacking loop.

3.4.2. Multiple object instances. A basic assumption is that the FPGAs can only implement a single hardware configuration at any one time. This is reflected in the design by the convention that IntraGiz constructors allocate hardware resources and the destructors release them. This means that having, for example, two InputPin(PB0) objects in existence concurrently is not meaningful. On the contrary, IntraGiz raises an exception to highlight the danger that conflicting software protocols may be trying to control the same hardware, port pin PB0, simultaneously.
In terms of layered-use objects, however, multiple user instances may be desirable. In the CORDIC example, this could be allowed simply by declaring the private FPGAbus and InputPin members as static. Furthermore, a sophisticated configure-on-demand system could be managed by having the CORDIC class download the corresponding FPGA program (from an MPU RAM library of Xilinx configurations) if necessary, and dynamically allocate the associated FPGAbus and InputPin objects.

The simple tools available in IntraGiz are adequate to give software running in the RPB’s microprocessor access to its configurable hardware, within the constraints of the hardwired interconnections. But how about software external to the board? Gizgate was created as a means of exporting the IntraGiz classes outside of the RPB, thus enabling convenient access to its FPGAs by programs running on network-connected hosts. Currently being refined and implemented by master’s student Dave Sharp, its design is sketched in the following sections.

4. Gizgate: a distributed FPGA gateway

The conceptual model for Gizgate is shown in Figure 7, with the IntraGiz-only model on the left. On the right side, a layer of communication software has been inserted between the user software and the IntraGiz methods on the RPB, so that the user software can be rehosted elsewhere. This is done in a transparent fashion, leaving the “virtual execution path” intact.

To be specific, an off-board user includes the same “IntraGiz.h” file and writes precisely the same program that would work directly on the board. The program is compiled with GCC for the host computer, and linked with Gizgate, rather than IntraGiz, methods. Gizgate translates
each call to an IntraGiz constructor or method into an inter-
process message which it transmits to its board-side server
module. That server is linked with the regular IntraGiz
library and carries out the request. Since IntraGiz is essen-
tially a master/slave protocol, no activities originate from
the FPGA except enabled interrupts. These are conveyed
back to the host computer by Gizgate and result in calls to
the specified handler function.

4.1. Distributed Architecture

Gizgate is actually more complex than Figure 7 may
imply. It consists of two autonomous programs and a set of
stubs (i.e., the “Gizgate Methods” in Figure 7). To under-
stand this architecture one must first appreciate the con-
straints imposed by the RPB’s means of external
communication. This is via the two serial lines shown in
Figure 1. In practice, these are plugged into two serial ports
(TTY A and TTY B) on the same SPARCstation.

The line marked “ASCII terminal” is provided for a
command line interface to the FBUG monitor/debugger,
and could just as well be plugged into a dumb terminal.
However, it is more convenient to handle it via UNIX “tip”
terminal interface program) running in a shell window.

The other line is used by FBUG to log into a host that
possesses disk storage for the purpose of obtaining down-
loads. There is a “transparent mode” in FBUG that allows
one to type from the tip window, through FBUG, and into
the host, and this is how the login is performed. After the
user’s command shell is running on the login host (so the
stdin and stdout file descriptors are connected to FBUG),
transparent mode is turned off. Now, when the user types
(in the tip window), say:

```
lo 20000 ;sload myprog.srec
```

FBUG strips off the characters to the right of the colon;
and sends them to the host command shell, where presum-
ably the “sload” program will begin outputting the S-
records of the file “myprog.srec” on stdout. These will be
received by FBUG’s “lo” command, and loaded into RAM
at address 20000.

It is evident that these arrangements are exceedingly
cumbersome and prone to user error, but this is how the
RPB comes “out of the box.” Nevertheless, the host serial
line is there, connected to an external UNIX process, and
that is the route which we seize to carry Gizgate communi-
cations onto and off of the board.

The architectural model of Gizgate is shown in Figure 8.
Its three components will be described from left to right.

```
4.1.1. Gizgate methods. This is the portion of Gizgate
that is linked with the configure user’s application. A call
to an IntraGiz constructor results in opening a socket con-
nection to the Gizgate server (next section). A method
invocation results in marshalling of arguments and a coded
message being sent to the server. When the server replies,
the method returns to its caller.

4.1.2. Gizgate server (host side). The portion of the Giz-
gate server running in the UNIX shell has two purposes,
one obvious and one obscure:

1. The obvious purpose is to relay stub messages between
the sockets and the serial link (which appears to the
server program as stdin and stdout).

2. The obscure purpose is to pretend to FBUG that Gizgate
is not there, should FBUG get control of the MPU and
want to issue UNIX commands. In that case, Gizgate
will realize that it is not talking to its board-side partner
(next section), and will simply execute FBUG’s com-
mand in a subshell. This way, the terminal user can
order fresh downloads without stopping and restarting
Gizgate.
```
Once the Gizgate server is running on a UNIX host, any user on the network wishing to utilize the RPB can do so, regardless of location, just by executing a program containing Gizgate stubs.

### 4.1.3. Gizgate server (board side)

The third portion of Gizgate is an MPU program that must be downloaded into RAM and given control. It takes over the host serial link from FBUG and establishes contact with its host-side partner. It then goes into a loop, waiting for messages. These are decoded, the appropriate IntraGiz objects are constructed, and their methods are invoked. In effect, the IntraGiz objects instantiated by this program are the “real” ones, capable of interacting with the FPGAs directly; the corresponding objects in the remote user’s software are dummies.

### 4.2. Benefits

Since with Gizgate we have inserted a layer of complex communications between the software and the configurable hardware, it will be evident that “speed” is not the motivation. Indeed, the communications overhead will, for most applications, swamp any acceleration that could be obtained by calculating in hardware. However, there are other reasons why Gizgate constitutes an attractive component in our configware.

First, it allows the application developer to get access to debugging facilities not available in the primitive RPB environment. It would be helpful to get the software working first on a UNIX host, and then move it to the MPU.

Second, some applications may need access to disk files or other operating system services that only exist on the UNIX host. Gizgate allows them to have it for the price of a speed penalty.

Third, Gizgate facilitates timesharing of a scarce resource—the RPB—among multiple users, and can offer them access to popular hardware configurations.

Furthermore, the worst part of the time penalty is incurred from the existing 14.4 Kbaud serial line connecting the RPB and its host. With the proper software in the MPU, the board’s external connectors could be used to attach it directly to the network or to a workstation bus extender.

### 5. Future work: an IPC layer for codesign

One of our projects under the heading of codesign is the synthesis of C++ source code from CSP (Communicating Sequential Processes) [5] for the purpose of realizing executable specifications. This software is called CSP++ [4]. We want to carry this work further and also produce synthesizable VHDL from CSP. The next step would be to partition CSP specs into hardware and software portions, on a process by process basis, and automatically generate the interprocess communication.

The RPB is an excellent platform for these experiments, and we intend to build the IPC primitives on top of IntraGiz. For example, a hardware process could be represented in the software partition as a wrapper object (as in Section 3.3). Process objects could then be invoked on the software side in a uniform way regardless of their location, with the IPC to FPGA-resident processes going through IntraGiz.

### 6. Conclusion

The CMC Rapid-Prototyping Board presents rich capabilities for research in configurable computing, but these have been difficult to utilize. With IntraGiz and Gizgate, we have taken two steps toward better exploiting this resource and we expect to find our configware welcomed in other Canadian universities.

### 7. References


