An Object-Oriented Communication Library for Hardware-Software CoDesign

Frank Vahid
Department of Computer Science
University of California
Riverside, CA 92521
vahid@cs.ucr.edu, www.cs.ucr.edu

Linus Tauro
Quickturn Design Systems, Inc.
440 Clyde Ave.
Mountain View, CA 94043
tauro@cs.ucr.edu

Abstract

Implementing communication between hardware and software components can be a time-consuming task. Numerous communication protocols are available, differing greatly in their implementation details. Designers must spend much time focusing on those details. Even when libraries are available to encapsulate communication into C or VHDL routines, these routines are not consistent across protocols, making it difficult to switch to other protocols. In this paper, we propose an object-oriented communication library, which provides pre-implemented channel-based send/receive communication primitives, allowing easy implementation and seamless migration across protocols and components.

1 Introduction

Systems often consist of a combination of processing components, such as microprocessors, microcontrollers, and custom ASIC/FPGA processors. Ideally, application developers would be able to develop software and hardware code using abstract send/receive primitives for communication, based on a communicating sequential processes (CSP [1]) paradigm as proposed in [2], for example. A primitive's underlying implementations, such as the choice of a particular bus type and protocols, the assignment of communications to ports, and the setting and reading of interface registers and port signals, would be hidden or created later.

However, there is presently a gap between message passing primitives and their implementations, as illustrated in Figure 1. For some architecture and protocols, some abstract communication primitives do exist, though most fall far short of send/receive primitives; for example, ISA bus transfers of a PC [3] are achieved using *in* and *out* assembly instructions, and serial transfers of an Intel 8051 are achieved by configuring several registers [4]. For other architectures and protocols, direct port manipulation is necessary, such as for most transfers for an FPGA or microcontroller. This gap means that developers must spend much effort learning numerous varied communication details and testing those communications. While developers have certainly built subroutines to hide those details, to our knowledge

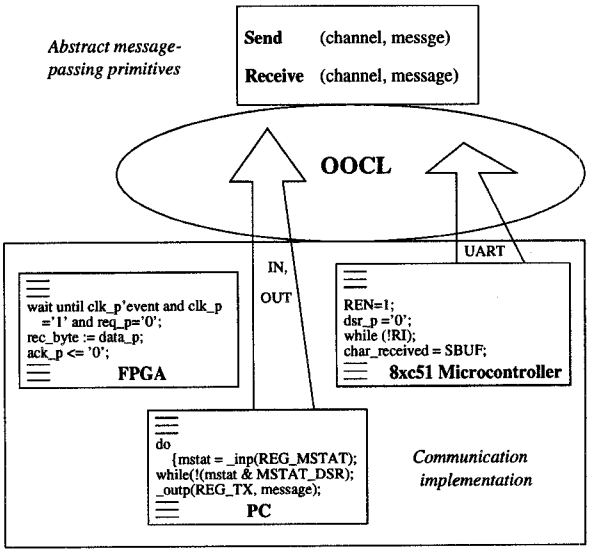

Figure 1: OOCL aims to bridge the gap between abstract message-passing primitives and communication implementations.

there is no published library of routines that supports a message passing paradigm while being consistent across architectures and platforms.

We are therefore developing an object-oriented communication library (OOCL). OOCL provides C (or C++) and VHDL send/receive communication primitives for numerous common protocols and components, with pretested underlying implementations. A designer chooses an OOCL channel supporting the desired protocol, without needing to focus on underlying implementation details. A user instantiates a communication channel object, initializes it, and then sends or receives messages over it; all access to low-level ports, registers, and communication behavior is hidden in the object's implementation. Because OOCL is a library, existing languages like C and VHDL need not be modified, and no synthesis tools are required to generate the communication behaviors (though synthesis and compilation are obviously necessary to implement those behaviors). Instead,

0-8186-7895-X/97 $10.00

OOCL needs to be ported to new architectures, just as any other library (such as a math library) must be ported.

The OOCL approach complements related codesign interface research. In [5], a solution is proposed that automates the hardware-software interface using minimum glue logic, while satisfying timing constraints. To communicate with a peripheral device, the processor generates a sequence of signals (SEQs) that read and write the device's ports. Symphony [6] defines a standard communication protocol for all processor components, using send and receive operations with a synchronous wait protocol, where the sender asserts a ready signal and then waits for the receiver to assert its own ready signal. In [2], a codesign methodology is discussed using process communication primitives that allow three types of process interaction: synchronized data transfer, unsynchronized data transfer and synchronization without data transfer. In [7], a system design methodology is discussed using abstract send/receive channels for communication. In all of these approaches, OOCL can be used to encapsulate the communication using send/receive primitives, while implementing the protocol, without any modification to the C or VHDL languages.

Operating systems often include abstractions for communication, such as remote procedure calls (RPCs), mailboxes, and inter-process communications (IPCs). However, most of these are not supported by custom processor components and microcontrollers, since such devices do not usually have an operating system.

OOCL is intended for use when at least one of the communicating components has flexible ports usage, such as a microcontroller or an FPGA, or when two fixed components use the same protocol, such as the PC RS232 serial protocol; to interface two components having fixed incompatible protocols, either an interface process [8] or interface transducer [9] must be generated.

2 Using the library

We introduce use of OOCL with the simple public-key encryption example of Figure 2; the details of the encryption algorithm are not shown. A portable data-entry device (using an 8051 microcontroller) downloads its data in encrypted form to a PC, which requires that it first receives two public encryption keys from the PC. The communication occurs between one of the PC's serial ports, which uses the RS232 protocol, and the 8051's built-in UART and its associated ports, along with additional general ports for control, as shown in Figure 2(a). Figure 2(b) shows the C code required by the 8051 to achieve its part of the communication, without the use of OOCL; complementary code would exist on the

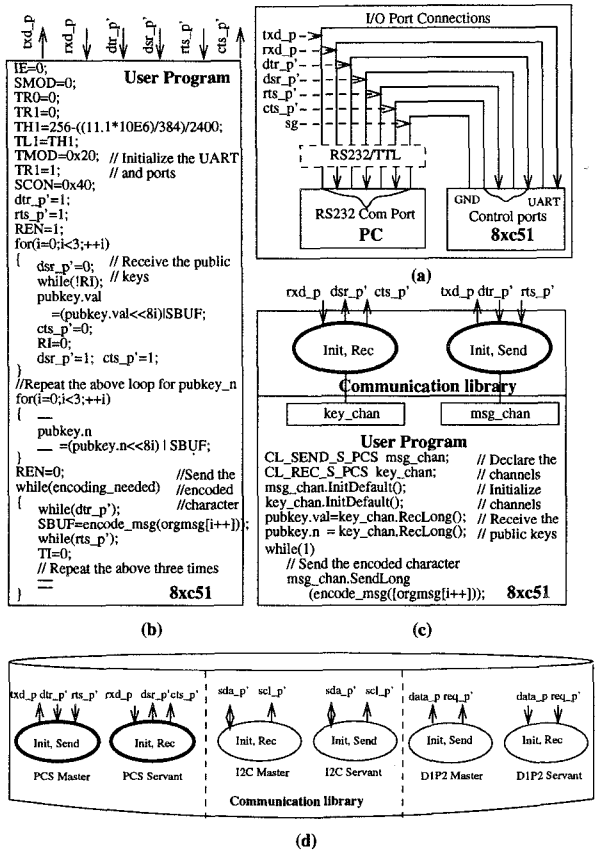

Figure 2: (a) Block diagram for the RSA system, (b) user code without and (c) with the OOCL routines using the RS-232C serial protocol, (d) library routines.

PC. Note that this communication code is non-trivial, requiring knowledge of many low-level details. The programmer must study the details of the serial protocol, referring to pin configurations, architectural details and timing diagrams. This inevitably results in the user having to iterate a few times to make the communication code error free.

Figure 2(c) shows the same code re-written using the OOCL routines. Having decided to use a PC and 8051 communicating over the PC's serial port, the user first selects the appropriate PC serial channel objects from OOCL in Figure 2(d). The user then declares a channel *key_chan* for receiving the keys, and another channel *msg_chan* for sending encoded data. The user then initializes these channels, in this case choosing the default initialization ports and speed, and then performs sends and receives over those channels. Note that the user is presented with abstract message passing, as desired,

even though underlying implementations do exist.

The reduction in code size and development time is significant, but even more significant advantages are seen when we change the communication. Suppose we choose to use the PC's parallel port instead of the serial port to speed up the data transfer. Such a change would require a complete rewrite of the code in Figure 2(b), in addition to the study of the new protocol and the testing of the implementation. However, when using OOCL, such a change merely involves selecting a new channel object from the library, i.e., replacing *PCS* in Figure 2(c) by *PCP*. Alternatively, suppose we decide to use the PC's ISA bus for the transfer for even more speed. This change requires a device faster than an 8051, so we might choose an FPGA. Once again, the communication code would only change slightly when OOCL is used, i.e., just declare a different channel type, but would require a complete rewrite without OOCL.

3 Library implementations

We now discuss the underlying implementations for two example protocols.

3.1 PC serial protocol

Information is frequently transferred to a PC, either as integral part of the design itself or to down-load information from an application. The RS-232C standard for serial interfaces is suitable for applications where lower data-transmission rates are acceptable, and the transmission is over short distances or when a limited number of I/O pins are available for communication. This protocol is especially suitable for slower devices like the 8051 which cannot keep up with the faster transfer rates of the ISA or PCI bus. The TTL logic level signals need to be converted to the non-TTL interface needed by the RS-232C standard.

The receiving device controls the flow of data from a transmitting device. Hardware handshaking is used for developing the PCSerial communication library routines. This type of handshaking uses dedicated circuits to control the transmission of data. Figure 2(a) showed the block diagram of a PC and an 8051 microcontroller connected via a serial interface. On the PC side, the user can use any one of the available RS-232C serial ports. The RS-232C/TTL drivers/receivers interface the RS-232C and TTL voltage levels.

The transmit data (*txd_p*) and receive data (*rxd_p*) lines run from the PC to the UART data ports, which are P3.1 and P3.0 respectively, on the 8xc51. *sg* is the RS-232C signal ground. The control handshake signals, data terminal ready (*dtr_p'*) and request to send (*rts_p'*) run from the PC to the 8xc51, while the data set ready (*dsr_p'*) and the clear to send (*cts_p'*) are the control signals from the 8xc51 to the PC. General purpose 8xc51 I/O ports can be used for the control lines.

We now describe the naming conventions used by OOCL. Active low logic is indicated by appending a '. The communication ports are indicated by appending "_p". For example, *ack_p'* indicates a port that is active low. Declarations that are part of the library are indicated in upper-case and are prefixed with "CL" e.g., *CL_PARITY_NONE*. Other declarations are indicated in upper-case and are not prefixed with "CL". Variables are indicated in lower case, with an underscore if needed. The following fields are used in a channel name and are separated by an underscore:

- A channel name is prefixed with "CL" to indicate that is a communication library channel.
- The next field is "SEND" or "REC" to indicate whether the channel will be used to send or receive messages.
- This is followed by "M" or "S" to describe whether it is associated with a master or servant process. The master process initiates the transfer.
- An abbreviation for the protocol used is next, which may include additional information if needed.
- Finally "A" can be appended to distinguish the addressed mode of communication from the point-to-point mode.

Figures 3 and 4 show the OOCL routines, which use the PCSerial protocol for the PC and 8xc51 respectively. The figures show the PC send and the 8xc51 receive channels. The complementary channels are similar. Each channel type's library entry consists of declarations, initializations, and send/receive transfers.

Declarations: The PC send channel is declared as *CL_SEND_M_PCS* while the 8xc51 receive channel is called *CL_REC_S_PCS*, conforming to the CL naming convention. The data members for the PC send channel are *txd_p* for the data, *dsr_p'* and *cts_p'* for the control, and *port_ads* to store the serial port address.

The 8xc51 receive channel includes *rxd_p* for the data, and *dsr_p'* and *cts_p'* for the control lines. The channel member functions for accessing the ports (*_inp()* and *_outp()* for the PC, and *SetPortBit()* and *GetPortbit()* for the 8xc51) are accessible only to the OOCL routines.

Initializations: While using the OOCL routines for the PC, the user can call the default initialization. In this case, the communication parameters are set to COM1, which is usually the 9-pin serial port, 2400 baud, no parity, 8 data-bits, and 1 stop-bit. Otherwise the user could call the non-default initialization by specifying the communication parameters. In both these cases, the port addresses are fixed registers. Similarly, when

Declare Channel

```
class CL_SEND_M_PCS
{
    uint  txd_p;            // transmit data line
    uchar dsr_p', cts_p'; // handshake signals
    uint port_ads;          // serial port address
    // Set the baudrate
    void baudset(uint port, long baud_rate);
    // Set the parity, data bits, stop bits
    void comparmset(uint data_bits, uint stop_bits,
    // Read, write to an I/O port          uint parity);
    uint  _inp(uchar addr);
    void  _ outp(uint addr, uchar value);
public:
    // Initialize ports to default values
    void InitDefault();
    // Initialize ports to user specified values
    void Init(uint port, long baud_rate,  uint parity,
                        uint data_bits, uint stop_bits);
    // Send the message (character)
    void Send(uchar message);
};
```

Initialize channel (default)

```
void CL_SEND_M_PCS::InitDefault()
{
    dsr_p' = CL_MSTAT_DSR;
    cts_p' = CL_MSTAT_CTS;
    // Default initialization is port-COM1, 9600
    // no parity, 8 data bits and 1 stop bit
    port_ads = CL_PCS_PORT_COM1;
    txd_p = CL_REG_TX;
    baudset(port_ads, 9600);
     comparm(CL_PCS_PARITY_NONE, CL_PCS_
            STOPBITS_1,  CL_PCS_DATABITS_8);
    // Turn off handshaking
    _outp(CL_REG_MCONT, 0);
}
```

Send message (character)

```
void CL_SEND_M_PCS:: Send(uchar message)
{
    uint mstat = 0;        // to test modem status reg
    // Device asserts DSR when it is ready to receive
    do  { mstat = _inp(CL_REG_MSTAT);}
    while(!(mstat & dsr_p'));
    mstat = 0;
    // Send the message
    _outp(txd_p, message);
    // Device asserts CTS after receiving
    mstat = 0;
    do  { mstat = _inp(CL_REG_MSTAT);}
    while(!(mstat & cts_p '));
    mstat = 0;
    // Wait for both DSR and CTS to go low
    do
       { mstat = _inp(CL_REG_MSTAT);}
    while(mstat & dsr_p ');
    mstat = 0;
    do{ mstat = _inp(CL_REG_MSTAT);}
    while(mstat & cts_p' );
}
```

Figure 3: OOCL code extract for the PC serial protocol: PC send.

Declare Channel

```
class CL_REC_S_PCS
{
    uchar rxd_p             // receive data line
    uchar dsr_p', cts_p';  // handshake signals
    // Write to a port bit
    void SetPortBit(uchar port, bit value);
    // Set the specified baud rate
    void baudset(uint baud_rate, float freq);
public:
    // Initialize ports to default values
    void InitDefault(float freq);
    // Initialize ports to user specified values
    void Init(uchar dsr_p_addr,  uchar cts_p_addr,
                        uint baud_rate, float freq);
    // Receive the message (character)
    uchar Receive();
};
```

Initialize channel (default)

```
void CL_REC_S_PCS::InitDefault(float freq)
{
    rxd_p = P3^0;
    dsr_p' = _CL_DEFAULT_DSR_P;
    cts_p' = _CL_DEFAULT_CTS_P;
    IE=0;                   // Disable interrupts
    SMOD=0;                 // To set baud rate
    // Both timers are turned off initially
    TR0 = 0;
    TR1 = 0;
    baudset (2400, freq);
    // Initialize ports to high impedance
    SetPortBit(dsr_p', 1);
    SetPortBit(cts_p', 1);
}
```

Receive the message (character)

```
uchar CL_REC_S_PCS::Receive()
{
    uchar char_received;
    REN = 1;                // Enable reception
    // Set the primary handshake
    SetPortBit(dsr_p',  0);
    // Read the character once received,  from the
    // UART register
    while (! RI);
    char_received = SBUF;
    // Next set the secondary handshake
    SetPortBit(cts_p', 0);
    // Reset RI and turn off handshake signals
    RI = 0;
    SetPortBit(dsr_p', 1);
    SetPortBit(cts_p', 1);
    REN = 0;                // Turn off reception
    return char_received;
}
```

Figure 4: OOCL code extract for the PC serial protocol: 8xc51 receive.

using the 8xc51 OOCL routines, the user can call the default initialization. In this case, the crystal frequency is specified and the 8xc51 timer registers are loaded appropriately to set the baud rate at 2400 baud. The I/O port addresses for *dsr_p'* and *cts_p'* are pre-assigned in the library. Alternately, the user can call the non-default initialization function and pass the baud rate and the I/O port addresses for *dsr_p'* and *cts_p'* as parameters. *rxd_p* is always port 3.0, which is the UART receive register for the 8xc51. **Send/Receive:** Figures 3 and 4 show the PC send and 8xc51 receive routines for a byte transfer. The protocol has been discussed earlier. The RI flag on the 8xc51 is set once a character has been been received by the UART and has to be cleared to enable the reception of the next byte.

3.2 Custom protocols

Many components, such as microcontrollers and FPGA's, have ports which are flexible, not having a pre-defined use. In other words, the component's program or circuit can set and read these ports in any manner desired. Flexible ports are in contrast to fixed ports, like those associated with the ISA bus, which have a specific purpose (e.g., request, acknowledge, interrupt, data, or address). When two components can communicate using flexible ports, custom protocols are required.

There are several characteristics that a custom protocol behavior may possess. We can taxonomize these characteristics as follows:

- Master versus Servant: A master behavior initiates a send or a receive data transfers, using a request port. The servant must respond to such

requests by either receiving or sending the data.

- Sender versus Receiver: Each channel may be either for sending or receiving.
- Data size: Data must be encoded into some number of bits in order to be transferred.
- Data bus size: The data bus uses some number of ports, which may be less than the data size, in which case a number of "chunks" are sequentially transferred.
- Four-phase versus two-phase handshaking: In two-phase handshaking, the master asserts the request signal, waits for a specified time, and then deasserts the signal and proceeds. In four-phase handshaking, the master asserts the request signal, but then waits for the servant to assert and acknowledge signal, only then proceeding. Four-phase communication achieves synchronized data transfers, while two-phase is faster when synchronization can be ensured by other means.
- Point-to-point versus Addressed mode: A master may have many servants, in which case the master must provide an address along with a request. If there's only one servant (point-to-point), then no address is required.
- Address bus size: If an address is used, its size must be known.

We point out that multiple channels may share the same ports. The actual binding of abstract channels to ports is done during the initialization of the channel; in the same manner, address and data may also share ports.

Figure 5 shows the block diagram for a two-phase addressed data transfer between two 8xc51s, a liquid crystal display (LCD) and an FPGA. At a given time, only one of these devices can act as the master on the bus, the figure shows this to be an 8xc51. All the devices on the bus share an 8-bit bidirectional data port. An 8-bit address port runs from the master to each servant on the bus. In addition, an address request line (*addr_req_p'*) and a data request line (*req_p'*) also goes from the master to each servant. General purpose 8xc51 I/O ports are used for the address, data and control lines.

Figures 6 and 7 show the OOCL routines for the master and servant 8xc51s respectively, which use the two-phase addressed protocol. The figures show the master send channel and the servant receive channel. The complementary channels are similar. Each channel type's library entry consists of declarations, initializations, and send/receive transfers, consistent with the OOCL methodology. Only one servant device is addressed at a time and that device responds within the two-phase delay.

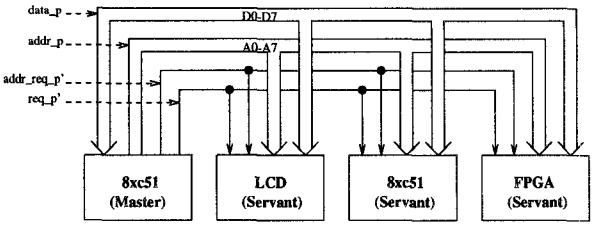

Figure 5: Bus connections for the two-phase addressed protocol using an eight-bit data port.

A number of OOCL routines have been developed to support custom protocols; we omit details here. Because of the large number of combinations of protocol characteristics, there are a large number of custom routines. For this reason, OOCL currently supports a subset of possible characteristic values: data size is 8, data bus size may be 8, 4, or 1, and address size is equal the the data bus size.

3.3 Other protocols

We have developed an initial library for a number of protocols. For PC's, we have C routines for the serial port and for the ISA bus, and preliminary routines for the PCI bus. For 8051 microcontrollers, we have C routines for custom protocols, the PC serial protocol, and the Inter-Integrated Circuit (I^2C) serial bus. For custom hardware (FPGAs and ASICs), we have VHDL routines for custom protocols, ISA and PCI.

4 Conclusions

OOCL bridges the gap between the abstract message passing primitives desired by the user, and the underlying communication implementation. It provides a consistency across diverse protocols and hardware-software components. Because OOCL is based on libraries, no additional tools are needed to generate the interface, and no language extensions are necessary. In addition to supporting the initialization, send and receive methods, the OOCL servant channels have been enhanced with methods that determine if the channel is ready to send or receive data, and with other routines that then proceed to complete the transfer. This enables the OOCL methodology to be applied to multiple processing in software using interrupts or a RTOS, or in hardware using a bus controller.

Future directions include expanding the library to include other common protocols, and investigating automated generation of the library routine implementa-

Declare Channel

```
class CL_SEND_M_D8P2_A
{
  uchar data_p, req_p', addr_p, addr_req_p';
  // Data, function members for 2 phase delay
  uint  wait_time, oneus_loop;
  void WaitFor(uint wait_time, oneus_loop);
  // Write to a port bit
  void SetPortBit(uchar addr, bit value);
  // Write a character to an 8 bit port
  void SetPortByte(uchar addr, uchar value);
public:
  // Initialize ports to default values
  void InitDefault();
  // Initialize ports to user specified values
  void Init(uchar data_p_base_addr, uchar req_p_addr,
    uchar addr_p_base_addr, uchar addr_req_p_addr,
                uint wait_time, uint oneus_loop);
  // Send message (character)
  void Send(uchar addr, uchar message);
};
```

Initialize channel (default)

```
void CL_SEND_M_D8P2_A::InitDefault()
{
  data_p = CL_DEFAULT_DATA8_BASE_ADDR;
  req_p' = CL_DEFAULT_REQ_P;
  addr_p = CL_DEFAULT_ADDR8_BASE_ADDR;
  addr_req_p' = CL_DEFAULT_ADDR_REQ_P;
  wait_time = CL_DEFAULT_WAIT_TIME;
  oneus_loop = CL_DEFAULT_ONEUS_LOOP;
  // Set data, address, handshaking to high impdenace
  SetPortBit(req_p',1);
  SetPortBit(addr_req_p',1);
  SetPortByte(data_p,0xff);
  SetPortByte(addr_p,0xff);
}
```

Send message (character)

```
void CL_SEND_M_D8P2_A::Send(uchar addr,
                            uchar message);
{
  SetPortByte(addr_p, addr);  // Put address on port
  SetPortBit(addr_req_p',0);  // Receivers : check address
  // Wait as per the 2 phase delay
  WaitFor(wait_time,oneus_loop);
  SetPortBit(addr_req_p',1);  // Finish 2 phase handshake
  // Wait as per the 2 phase delay
  WaitFor(wait_time,oneus_loop);
  // Put data on the port
  SetPortByte(data_p, message);
  SetPortBit(req_p',0);       // Receivers : read data
  // Wait as per the 2 phase delay
  WaitFor(wait_time,oneus_loop);
  SetPortBit(req_p',1);       // Finish 2 phase handshake
  // Wait as per the 2 phase delay
  WaitFor(wait_time,oneus_loop);

  SetPortByte(data_p, 0xff);  // Release the data bus
  SetPortByte(addr_p, 0xff);  // Release the address bus
}
```

Figure 6: OOCL code extract for the two-phase addressed protocol using an eight-bit data port: master send

Declare Channel

```
class CL_REC_S_D8P2_A
{
  uchar data_p, req_p', addr_p, addr_req_p';
  uchar addr;                 // Address of this receiver
  // Read/write from/to a port bit
  bit GetPortBit(uchar addr);
  void SetPortBit(uchar addr, bit value);
  // Read/write a character from/to an 8 bit port
  uchar GetPortByte(uchar addr);
  void SetPortByte(uchar addr, uchar value);
public:
  // Initialize ports to default values
  void InitDefault(uchar addr);
  // Initialize ports to user specified values
  void Init(uchar data_p_base_addr, uchar req_p_addr,
    uchar addr_p_base_addr, uchar addr_req_p_addr,
                                    uchar addr);
  // Receive message (character)
  uchar Receive ();
};
```

Initialize channel (default)

```
void CL_REC_S_D8P2_A::InitDefault(uchar addr)
{
  data_p = CL_DEFAULT_DATA8_BASE_ADDR;
  req_p' = CL_DEFAULT_REQ_P;
  addr_p = CL_DEFAULT_ADDR8_BASE_ADDR;
  addr_req_p' = CL_DEFAULT_ADDR_REQ_P;
  addr = addr;
  // Set data, address, handshaking to high impdenace
  SetPortBit(req_p',1);
  SetPortBit(addr_req_p',1);
  SetPortByte(data_p,0xff);
  SetPortByte(addr_p,0xff);
}
```

Receive message (character)

```
uchar CL_REC_S_D8P2_A::Receive()
{
  uchar data message, bus_addr;
  // Wait until receiver address is on the bus
  while(GetPortBit(addr_req_p'));
  bus_addr = GetPortByte(addr_p);
  while(bus_addr != addr)
  {
    while(GetPortBit(addr_p));
    bus_addr = GetPortByte(addr_p);
  }
  // Finish 2 phase handshake
  while(!GetPortBit(addr_req_p'));

  while(GetPortBit(req_p'));  // Read data when ready
  message = GetPortByte(data_p);
  // Finish 2 phase handshake
  while(!GetPortBit(req_p'));

  return messsage;
}
```

Figure 7: OOCL code extract for the two-phase addressed protocol using an eight-bit data port: servant receive.

tions to simplify library development and porting to new architectures.

References

[1] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[2] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," in *IEEE Design & Test of Computers*, pp. 6–15, 1993.

[3] L. S. Eggebrecht, *Interfacing to the IBM Personal Computer*. Sams, second ed., 1990.

[4] Philips, *80C51-Based 8-Bit Microcontrollers Data Handbook*, February 1995.

[5] P. Chou, R. Ortega, and G. Borriello, "Interface cosynthesis techniques for embedded systems," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 280–287, 1995.

[6] S. Vercauteren, B. Lin, and H. D. Man, "Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications," in *Proceedings of the Design Automation Conference*, 1996.

[7] D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, and P. Fung, "System design methodologies: Aiming at the 100 h design cycle," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 4, no. 1, pp. 70–82, 1996.

[8] S. Narayan and D. Gajski, "Interfacing incompatible protocols using interface process generation," in *Proceedings of the 32nd Design Automation Conference*, pp. 468–473, 1995.

[9] G. Borriello, *A New Interface Specification Methodology and its Applications to Transducer Synthesis*. PhD thesis, University of California, Berkeley, May 1988.