To Stream or not to Stream

Jeffrey Sickel

Corpus Callosum Corporation
Evanston, IL 60202 USA
jas@corpus-callosum.com

ABSTRACT

This paper describes a technique to manage multiple serial devices that switch data transmission modes between request-response and streaming protocols. It utilizes the ideas of coroutines and communicating sequential processes to build concurrent input and output processing routines for each device. The example program leverages Limbo's buffered channels to concurrently queue and process data from multiple inputs in soft real-time.

1. Introduction

Scores of researchers have promoted various structured programming techniques to manage concurrency over the years. Conway [1], Dijkstra [2], Hoare [3,4], and Kahn and MacQueen [5] have all contributed to the body of knowledge that frames how concurrent programs are written today. Yet, even with this history, research, and practice of programming, the processing of input and output (I/O) continues to be reduced to routines that must run to completion before another task can continue. This inherent single-tasked nature of handling I/O means that user programs, especially with graphical front ends, need to investigate concurrent approaches to managing multiple I/O interfaces. For example, reading data from a slow remote device while at the same time needing to service other input tasks requires managing system state successfully to prevent blocking conditions from interrupting the logic flow.

Coroutines, coined by Conway and expanded by later researchers, are a proven way to handle various issues surrounding the logical segmentation of code. The definition, as published by Conway [1:396], elucidated *separability,* or modularity:

> [The coroutine] may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.

Leveraging this modularity helps in designing optimal input- and output-handling routines based on machine or process state at any given time. Though the logic is modularized, state changes stored by the coroutines do not fully model a concurrent or parallel system as the join required for two coroutines to exchange data will cause at least one to wait, or block, until the other is able to send or receive data to the peer (see Knuth [6] for examples).

In order to achieve greater concurrency a mechanism is required to leverage the I/O handling of coroutines into a more approachable framework. The seminal paper by Hoare [3], *Communicating Sequential Processes* (CSP), provides such a framework and influenced the design of the Limbo programming language. CSP is the mechanism used by Limbo channels to provide bi-directional communication between processes. Limbo processes are able to run concurrently, in parallel if the underlying hardware supports it, by leveraging constructs provided by CSP. Ritchie [7] describes the simplified use of Limbo channels to handle reading data from a single device. Updates to Inferno and Limbo since Ritchie's document now include buffered channels, a

language extension that allows for up to *n* buffer size of values to be sent without blocking. This paper contributes to the available documentation on using Limbo channels to manage concurrent I/O tasks.

The final development stage of a new aero-acoustic levitator (AAL) [8] required a program to interface, control, and display data from various serial devices. Some of these devices use a simple request-response protocol, whereas others switch from request-response communication to a streaming protocol. All of the devices operate in modes where sending a request and then blocking to wait for a response will not suffice, as there are certain conditions where a message will be sent from the device out of order from a sequential request-response loop initiated by the user control program. In order to manage these message states, this implementation uses multiple asynchronous processes as coroutines to handle all of the data management and machine control. The user is presented with a seamless interface isolated from the underlying serial communication tasks.

The following sections present *aal/pyro* (Figure 1), a Limbo program that uses CSP programming techniques to manage the I/O from multiple devices. The system combines a remote pyrometer with a linear XY translator, three serial devices in total. The program plots, and optionally logs, the stream of temperature measurements sent from the pyrometer. Position of the pyrometer is controlled using the XY translators through numeric key entry and a graphical view that accepts coordinates converted from mouse input. These tasks are accomplished by building program structures to handle I/O from the various serial interfaces. The implementation leverages Limbo's buffered channels for inter-process communication to control parallel data feeds.

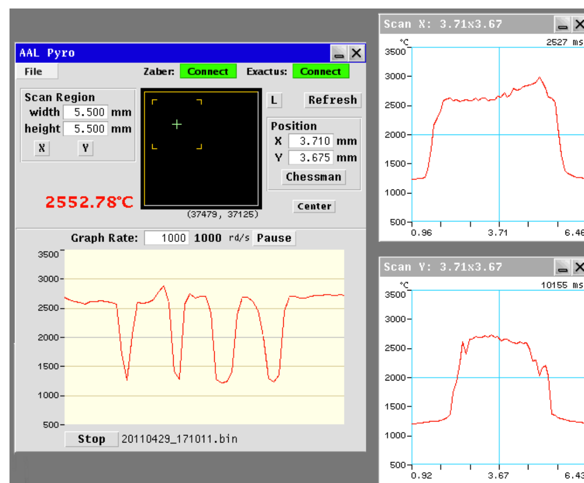

Figure 1 aal/pyro application

2. Design

The *aal/pyro* program is a graphics interface that manages I/O from multiple physical devices. Different serial communications protocols are used to process data from each device. Blocking conditions common in I/O routines are eliminated from the control flow by separating each task into independent processes (thought of as coroutines). The program manages these coroutines by communicating through Limbo channels. By isolating the blocking functions into separate coroutines, the program is able to run seamlessly without interrupting the interactive program flow. The Limbo *alt* (alteration) statement [9] is used to manage concurrent communications between each of the key coroutines that depend on data processing. The *alt* is like a *case* or *switch* statement but specific for handling multiple communications channels. Thus the program logic can be declared in simple synchronous terms even while handling asynchronous events.

On startup, the *pyro* process (Figure 2) spawns off a single *timer* process as a mechanism to

signal the *pyro* process to flush any out-of-sync communications from the XY translator (Zaber). The *timer* does nothing until a connection is made to remote devices. The main *alt* event loop in the *pyro* process handles user interaction and drawing routines representing the coordinate space of the Zaber devices. Plotting and other drawing updates are managed using separate processes started after connection to the Exactus pyrometer is made.

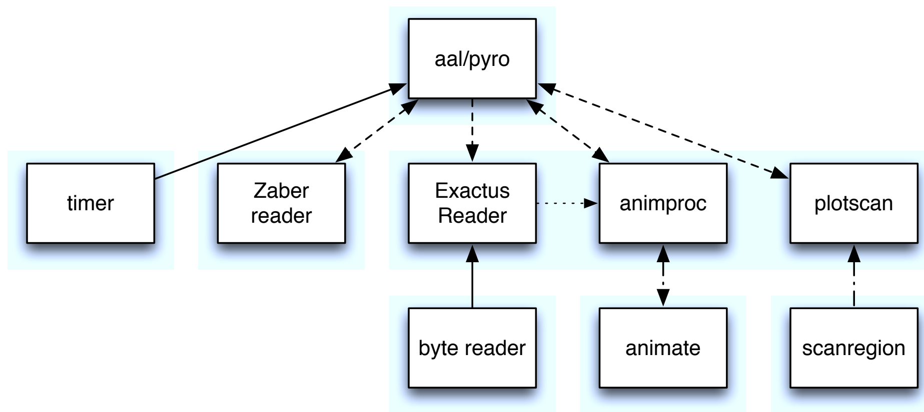

Figure 2 aal/pyro processes

All serial devices have a minimal data structure referred to as a *port*. For simplicity, each port can be opened from a serial device file using *sys->open()* or via *sys->dial()* to connect to a remote service. The connection routine is itself spawned off from the *pyro* process so that the blocking *open()* or *dial()* calls do not inhibit other user interface interaction. A message will be sent over a channel to the *pyro* process notifying whether a successful connection was made. A valid connection will enable additional UI elements and spawn a dedicated process to manage reading from the device.

Connecting to the Exactus pyrometer starts up a series of other processes. The first is the reader used to analyze the different input data from the device. The reader spawns off a blocking byte reader that simply receives all data from the pyrometer serial interface and sends each byte back over a buffered channel, decoupling the analysis from the physical device. A separate process group is used for on-screen drawing. The *animproc* process is spawned after connection to the pyrometer and plots temperature data using a coordinated animate process. A separate *plotscan* process is created by a user-initiated event to plot temperature data in relation to the axial position of the pyrometer. Each of the interfaces is declared in its own module as defined in the following sections.

2.1. Zaber

The Zaber XY translators are interfaced via a single RS232 link that communicates with both the X and the Y linear stage. The serial protocol provides a clean, fixed byte-length message format for both sending and receiving data. Though the protocol works like a sequential transmit-response operation, not all commands to the device return a response message. Additionally, there is no guaranteed order to the responses and there are certain cases where the device issues state data interleaved between other response messages.

The connection to the device creates an instance of a Zaber port:

```
Port: adt
{
    pid:    int;
    local:  string;
    ctl:    ref Sys->FD;
    data:   ref Sys->FD;
    rdlock: ref Lock->Semaphore;
```

```
        wrlock: ref Lock->Semaphore;
        buf:    array of byte;

        write:  fn(c: self ref Port, b: array of byte): int;
    };
```

After successfully connecting to and opening the device, the *pid* is set to the process id of the spawned off *reader()*. A buffer of received bytes is stored in the *port* structure. The semaphore *rdlock* is used as the bytes buffered in the Zaber *reader* process are validated and consumed in the *pyro* process.

Spawning the *reader* off into its own process allows the blocking *sys−>read()* call to continually read data without interfering with the rest of the application event management. Data received from the device is stored in a buffer until a subsequent routine polls the buffered data.

```
reader(p: ref Port, pidc: chan of int)
{
    pidc <-= sys->pctl(0, nil);

    buf := array[1] of byte;
    for(;;) {
        while((n := sys->read(p.data, buf, len buf)) > 0) {
            p.rdlock.obtain();
            if(len p.avail < Sys->ATOMICIO) {
                na := array[len p.avail + n] of byte;
                na[0:] = p.avail[0:];
                na[len p.avail:] = buf[0:n];
                p.avail = na;
            }
            p.rdlock.release();
        }
        # error, attempt reconnect and try again
        openport(p);
    }
}
```

The parent *pyro* process checks for responses using ad hoc interleaves with directed timeouts. This is accomplished by calling *readreply()* whenever the parent processes needs to poll data from the device. Each *readreply()* call passes a millisecond time out parameter to enable logic flow to continue if a response has not be received.

The reply may be *nil* due to a timeout. If the response is not returned, it does not matter to the rest of the system state as no dependency is based on actual responses unless specifically requested in another routine. Scanning processes, discussed later, require exact position details and thus explicitly wait until the proper return has been received.

```
readreply(p: ref Port, ms: int): ref Instruction
{
    if(p == nil) return nil;
    if(ms < 0) ms = 60000;      # arbitrary maximum of 60s

    r : ref Instruction;
    for(start := sys->millisec(); sys->millisec() <= start+ms;) {
        a := getreply(p, 1);
        if(len a == 0) {
            sys->sleep(1);
            continue;
        }
        return a[0];
    }

    return r;
}
```

Readreply() calls the function *getreply()* to scan the buffer and check for valid instructions.

```
getreply(p: ref Port, n: int): array of ref Instruction
{
    if(p==nil || n <= 0)
        return nil;

    b : array of byte;
    p.rdlock.obtain();
    if(len p.avail >= 6) {
        if((n*6) > len p.avail)
            n = len p.avail / 6;
        b = p.avail[0:(n*6)];
        p.avail = p.avail[(n*6):];
    }
    p.rdlock.release();

    a : array of ref Instruction;
    if(len b) {
        a = array[n] of { * => ref Instruction};
        for(j:=0; j<n; j++) {
            i := a[j];
            i.id = int(b[(j*6)]);
            i.cmd = int(b[(j*6)+1]);
            i.data = b[(j*6)+2:(j*6)+6];
        }
    }
    return a;
}
```

Access to the *port* available data buffer is coordinated between the Zaber *reader()* process and any calls to the *getreply()* function through the semaphore *p.rdlock.* The semaphore is used to insure that the byte reading and message interpretation do not conflict while accessing the buffer. The *getreply()* routine obtains the semaphore lock, then checks the length to determine if a valid message is contained in the data. If the buffer contains enough bytes for *n* messages, it will store those bytes and trim the buffer. The semaphore is then released to allow the Zaber *reader* process to continue filling the buffer as new bytes are received from the device. This coordinated hand-off using the *readreply()* loop means that timeouts can be used while polling for new messages from the XY translators without blocking the *pyro* process.

2.2. Exactus

Making the Exactus pyrometer control transparent to the end user is the primary task of the *pyro* program. The device is connected through an RS232 or RS422 interface. The AAL connects to the pyrometer over TCP/IP using a Perle IOLAN SDS as a proxy for transferring the raw bytes to and from the device. The Exactus uses two serial protocols for normal operation: a request-response protocol known as Modbus, and a raw byte stream of data termed the legacy Exactus mode. When in the streaming mode, it is capable of transmitting up to one measurement per millisecond. Though this rate is not fast by modern computing standards, the mode switching over the same serial interface defines the way we have to implement the byte reader.

The pyrometer streaming mode leverages buffered channels to send decoded data frames to the *animproc* process that is dedicated to converting those data into numerical forms presented in a graphical view and optionally to a log file. On the one hand, this is no different than storing the data in a buffer and having another process continually poll for new entries in the same way one would let the request-response loop manage synchronous communications. But in this case the reader process isolates the mode switching and continually monitors the input bytes for proper message structure for both states. The resulting data streams are typed as they are read and subsequently handed off to the respective end points.

2.2.1. Modbus

On startup, the pyrometer communicates over its serial link using the Modbus RTU protocol. Modbus is a communications protocol used by many industrial devices and comes in three implementations: *RTU, ASCII,* and *TCP/IP.* The Exactus uses a subset of the RTU protocol to read and write coil and register values on the device. Each message is framed by 3.5-character times of silence (at 115.2kbps, a 303.819μs break between messages). All messages must be initiated by the *aal/pyro* program, as it is the master node in the serial configuration. The primary user interaction that utilizes the Modbus mode is to change the sampling rate, known as graph rate, and switch back into the Exactus streaming mode during actual data collection.

The Limbo Modbus module has been modeled on the 9P or Styx modules due to its transmit and response message structure. There are 19 function codes and an error type declared within the data structure. Unlike the incremental pairing of the T and R types in 9P, Modbus uses the same function code values when declaring both the transmit and response structures. For example, the TMmsg structure:

```
TMmsg: adt {
    frame: int;
    addr: int;              # 1 or 2 bytes
    check: int;             # 0 or 2 bytes
    pick {
    Readerror =>
        error: string;
    Error =>
        fcode: byte;
        ecode: byte;
    Readcoils =>
        offset: int;        # 2 bytes, 0x0000 to 0xFFFF
        quantity: int;      # 2 bytes, 0x0001 to 0x07D0
    Readdiscreteinputs =>
        offset: int;
        quantity: int;
    Readholdingregisters =>
        offset: int;
        quantity: int;      # 2 bytes, 0x0001 to 0x007D
...
    read:   fn(fd: ref Sys->FD, msglim: int): ref TMmsg;
    packedsize: fn(nil: self ref TMmsg): int;
    pack:   fn(nil: self ref TMmsg): array of byte;
    unpack: fn(b: array of byte, h: int): (int, ref TMmsg);
    mtype:  fn(nil: self ref TMmsg): int;
};
```

is nearly identical to the RMmsg structure for received data. As the TMmsg is requesting data from a certain register or coil, the return RMmsg message would include the actual results, as in:

```
    Readholdingregisters =>
        count: int;
        data: array of byte;    # registers, N (of N/2 words)
```

After a process writes a TMmsg, it will block waiting for the reply by calling *readreply()*:

```
EPort.readreply(p: self ref EPort, ms: int): (ref ERmsg, array of byte, string)
{
    if(p == nil)
        return (nil, nil, "No valid port");

    limit := 60000;         # arbitrary maximum of 60s
    r : ref ERmsg;
    b : array of byte;
    err : string;
    for(start := sys->millisec(); sys->millisec() <= start+ms;) {
        (r, b, err) = p.getreply();
        if(r == nil) {
```

```
            if(limit--) {
                sys->sleep(5);
                continue;
            }
            break;
        } else
            break;
    }

    return (r, b, err);
}
```

Note the similarity to the *readreply()* used to access the Zaber devices. Both of these use one function in a loop to poll the buffer through the *getreply()* function. By doing so, the developer can schedule events as needed and continue execution with simple recovery in the case of an error.

2.2.2. Streaming

The Exactus legacy streaming mode is a setting where the device sends out a continuous stream of messages at a set rate. The four message types are *temperature, current, dual* (temperature and current), and internal *device* temperatures. The stream contains packets of data messages of variable length that consist of a header byte defining the type, followed by one or more 32-bit IEEE 754 binary floating point values. The portion of the packet encompassing the floating point bytes may include escape codes masking the type and other reserved bytes within the message, thus creating a variable length packet.

The *temperature* and *current* types have a variable packet size of 5-9 bytes, determined by the escape sequences used in packing the message. The *dual* and *device* types packet size is 9-17 bytes in length. The lack of a length attribute in the Exactus message protocol mandates that each byte received be scanned and evaluated to test for message completion. The data structure used in Limbo to represent an Exactus message is:

```
Emsg: adt {
    pick {
    Temperature =>
        degrees:    real;
    Current =>
        amps:       real;
    Dual =>
        degrees:    real;
        amps:       real;
    Device =>
        edegrees:   real;
        cdegrees:   real;
    Version =>
        mode:       byte;
        appid:      byte;
        vermajor:   int;
        verminor:   int;
        build:      int;
    Acknowledge =>
        c:      byte;
    }

    unpack: fn(b: array of byte): (int, ref Emsg);

    temperature:    fn(m: self ref Emsg): real;
    current:    fn(m: self ref Emsg): real;
    dual:       fn(m: self ref Emsg): (real, real);
    device:     fn(m: self ref Emsg): (real, real);
    acknowledge:    fn(m: self ref Emsg): byte;
    text:   fn(m: self ref Emsg): string;
};
```

There is no hinting for the sampling time from the device; the receiver must calculate the timing interval based on the receipt of bytes. The timing accuracy depends not only on the resolution of *sys−>millisec()* but also on any latency in the receipt of the bytes from the device. Though latency can be an issue at higher transmission rates, the maximum 1kHz sample rate from the pyrometer is successfully handled.

2.2.3. Byte stream processing

Switching states between Modbus and Exactus modes requires a slightly more complex process structure for validating bytes received from the device than defined for the Zaber interface. *Aal/pyro* creates a reference *EPort* to store all of the connection elements:

```
EPort: adt
{
    mode:   int;                # Exactus or Modbus
    maddr:  int;                # Modbus address
    temp:   real;               # Last measured temperature
    rate:   int;                # Graph rate
    path:   string;
    ctl:    ref Sys->FD;
    data:   ref Sys->FD;
    wdata:  ref Sys->FD;
    rdlock: ref Lock->Semaphore;
    wrlock: ref Lock->Semaphore;
    buffer: array of byte;      # bytes from reader
    pids:   list of int;
    tchan:  chan of ref Exactus->Trecord;
    ms: int;                    # ms start of last packet

    write:     fn(p: self ref EPort, b: array of byte): int;
    getreply:  fn(p: self ref EPort): (ref ERmsg, array of byte, string);
    readreply: fn(p: self ref EPort, ms: int):
            (ref ERmsg, array of byte, string);
};
```

When a process is spawned off to connect to the device, the *path* is stored and *sys−>dial()* is called. After successfully establishing a connection, the members *ctl, data,* and *wdate* are populated using *sys−>open().* The blocking calls *sys−>dial()* and *sys−>open()* mean that it is important for the main loop to have started this connection routine concurrently as to not block any other elements of the interface or data handling of other I/O components. On successful initialization, the connection process sends a command back over a channel to the *pyro* process and then exits. The notification that the *Eport* has been initialized updates the interface and spawns off the *animproc* process used to plot any data from the pyrometer.

The interesting members of the data structure are the *mode, temp, rate, pids,* and *tchan*, as they are updated based on user interaction controlling the state of the device. Any commands that read or write value changes must first set the device to Modbus mode before making any further requests. The device will be switched back to Exactus streaming mode after the sampling rate is set and data collection is started. The *temp* variable is used as storage and a lookup mechanism for the most recently received temperature message from the pyrometer. *Rate* is a hint field set when the user changes the graphing and sampling rate of the device. The use of *pids* provides a list of subprocess ids to the parent process in case they need to be terminated.

The channel *tchan* is used when the device is in the streaming mode. When *tchan* is not *nil*, then temperature data will be sent from the reading process to another process that acts as a listener. This enables the *animproc* graphing process to be started independently from the *reader.* When a frame of data from the stream is available, it is sent over the *tchan* channel to the *animproc* process for handling within the graphics system.

The two processes that manage the Exactus serial communications are a *reader()* spawned by the *pyro* process, and the blocking *bytereader()*, spawned off by the reader to pick off bytes from the data stream:

```
bytereader(p: ref EPort, c: chan of (int, byte), e: chan of int)
{
    p.pids = sys->pctl(0, nil) :: p.pids;
    buf := array[1] of byte;
    while(sys->read(p.data, buf, len buf) > 0) {
        c <-= (sys->millisec(), buf[0]);
    }
    e <-= 0;
}
```

The channel *chan of (int, byte)* is a buffered channel created in the *reader* process that decouples the blocking reader from the actual decoder used to validate the data stream. The insertion of the *sys−>millisec()* in the tuple is used to mark the receipt time of the first byte that begins a message. Latency may offset the accuracy, but it does provide a mechanism to represent time between data messages from the pyrometer.

```
reader(p: ref EPort)
{
  p.pids = sys->pctl(0, nil) :: p.pids;
  c := chan[BUFSZ] of (int, byte);
  e := chan of int;
  spawn bytereader(p, c, e);

  for(;;) alt {
  (ms, b) := <- c =>
    p.rdlock.obtain();
    n := len p.buffer;
    if(n == 0) {
      p.ms = ms;        # used in Trecord, track first received
      l : list of byte;
      if(p.mode == ModeModbus) l = SMBYTES;
      else l = SEBYTES;
      if(!ismember(b, l)) { # frame error
        p.rdlock.release();
        continue;
      }
    }
    na := array[n + 1] of byte;
    if(n) na[0:] = p.buffer[0:n];
    na[n] = b;
    if(p.mode == ModeExactus && p.tchan != nil) {
      (i, m) := Emsg.unpack(na);
      if(m != nil) {
        t := ref Trecord(p.ms, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 1.0);
        pick x := m {
        Temperature =>  t.temp0 = p.temp = x.degrees;
        Current =>      t.current1 = x.amps;
        Dual =>
          t.temp0 = p.temp = x.degrees;
          t.current1 = x.amps;
        Device =>
          t.etemp1 = x.edegrees;
          t.etemp2 = x.cdegrees;
        * =>
          t = nil;
        }
        if(t != nil) {
          p.tchan <-= t;
          if(n > i) na = na[i:];
          else na = nil;
        }
      }
    }
    p.buffer = na;
```

```
      p.rdlock.release();
    <-e =>        # bytereader exited, try again
      openport(p);
      spawn bytereader(p, c, e);
    }
}
```

3. Graphical interface

The application has two primary graphical components: a representation of the XY position of the pyrometer and a temperature plot of data received. The *pyro* window provides a consolidated interface into the concurrent processes used to coordinate all of the I/O from the attached devices. The user can initiate an additional view that combines the position and temperature data into a consolidated scan plot. By providing a simplified view on top of the coordinated coroutines, the *pyro* process is able to synthesize the translator control and pyrometer data acquisition into a concise view that hides the multiple processes from the user.

3.1. Translator control

The Zaber XY translators create a 13x13mm region where the pyrometer can be focused. The data that are returned by the device are in micro-steps and are converted to millimeters for user viewing and numerically entered changes. There is an additional graphical panel that presents the position as a reticle that can be moved by a click in the view. The graphical interaction is managed completely within the *pyro* main alt loop using the *zcmd* channel:

```
c := <-zcmd =>
    if(dflag) sys->fprint(stderr, "zcmd: '%s'0, c);
    if(!plot.lock) {    # max microstep: 131327
        (nil, toks) := sys->tokenize(c, " ");
        pnt := Point(int hd tl toks, int hd tl tl toks);
        ms := real MAXMICROSTEP / real plot.bimg.r.dx();
        x1 := real pnt.x * ms;
        y1 := real pnt.y * ms;
        zsend(Instruction.newwithval(1, Zaber->Cmoveabsolute, int(x1)));
        zsend(Instruction.newwithval(2, Zaber->Cmoveabsolute, int(y1)));
    }
```

The *zcmd* is a string channel named for use within the Tk graphics system. An on-screen Tk panel sends X and Y coordinates over the channel. If the panel *plot* has not been locked by the user to ignore the commands, then the coordinates will be converted into the micro-steps required by the translator and written to the device. The function *zsend()* encodes the instruction into an array of bytes and writes them to the device in order to move to the assigned absolute position. A *zsend()* call is addressed to each translator as they are moved independently. The Zaber devices will not confirm an instruction until after the physical move has completed. The return values from the device may be received out of sequence to the calling convention as the time of travel between positions is the determining factor. There is no requirement to wait for the return result due to the use of a *timer* process checking for new queued return values before updating the display.

The *timer* process sends a message over *tchan* once per second. The *pyro* main alt loop receives the timeout message over the *tchan* and processes the event:

```
<-tchan =>
    if(!scanning) {
        if(zport != nil)
            while((r := zaber->readreply(zport, 1)) != nil)
                processzaber(r);
        if(epid > 0) ecmdc <-= PyroPlot->SAMPLE;
        else if(eport != nil)
            updatedegrees(exactus->temperature(eport));
    }
```

The *scanning* check ensures that the *pyro* process will only poll the Zaber buffer when the *plotscan* process is not actively running. Zaber translator messages are processed before attempting to update an on-screen temperature readout. The channel *ecmdc* is used to message the *animproc* process requesting a new temperature measurement. A response communication would then update the on-screen temperature. If *animproc* is terminated, then the *epid* is set to zero and a blocking call to the pyrometer is made; this requires the main loop to wait for a return from the pyrometer before updating the display and continuing to the next instruction.

3.2. Temperature plot

Graphic plotting of temperature data is managed through the *animproc* process spawned after successfully connecting to the Exactus device. Commands controlling logging, sampling rate, and whether or not to plot the data are sent over a channel by the *pyro* process. The buffered channel *recc*, of Exactus Trecord type, is used to receive data processed by the Exactus *reader* while operating in streaming mode:

```
Trecord: adt {
    time:        int;
    temp0:       real;
    temp1:       real;
    temp2:       real;
    current1:    real;
    current2:    real;
    etemp1:      real;
    etemp2:      real;
    emissivity: real;
    pack:   fn(nil: self ref Trecord): array of byte;
    unpack: fn(b: array of byte): (int, ref Trecord);
};
```

The *Trecord* is created by parsing values sent from the Exactus stream data. The *time* field is the milliseconds from the beginning of a log of the data. Logging to disk will use the *pack()* function to create the binary data written out to a filesystem.

Starting *animproc* will in turn spawn off another process to manage the actual drawing routines. This new process, *animate*, receives real values over another buffered channel:

```
animate(top: ref Tk->Toplevel, p: ref Plotter, c: chan of array of real)
{
    for(;;) {
        data := <-c;
        if(!p.paused)
            p.mavg = update(top, p, data);
    }
}
```

All screen drawing is buffered in a fixed-length array of real values before being sent to the *animate* process. The effect of this buffering is to allow the graphical plot to always present at least one minute of historical data. The generated plot point is an average of all the buffered data points; this works well for the full spectrum of graphing rates available from the Exactus pyrometer.

3.3. Scanning

During the course of a levitation experiment, it is important to verify that the pyrometer is focused on the sample in order to acquire the best temperature reading possible. In order to accomplish the optimal focusing of the pyrometer, the XY translators are used to scan a region in one dimension while simultaneously collecting temperature and position data. The scanning routine requires all of the prior I/O related functionality in order to accomplish its task in the *pyro* application.

Scanning is handled by spawning off a dedicated *plotscan* process to create and control a new

window where a plot is drawn showing position on the X axis and temperature on the Y axis. The creation of a *plotscan* process sets up the window and spawns off a separate short-lived process to move the XY translator and return temperature data for graphing:

```
c := chan[8] of (int, int, real);
comp := chan of int;
spawn scanregion(rect, c, comp);
```

The *scanregion* process calculates new positions to move the translator and sends command messages in a loop to make the move occur. At each position a temperature measurement is made and the resulting data is sent over a buffered channel, *c*, back to the *plotscan* process where an plot will be rendered on the display. Once the scan completes, a final message will be sent to move the pyrometer back to the starting position. The *scanregion* process will then send a message over the *comp* channel and promptly exit.

All other *aal/pyro* processes continue to run and update their graphical components while the scanning is taking place. Once the *scanregion* process has exited, it is possible for the user to click on the graphic temperature plot to move the pyrometer to a better centered location. The user may then repeat the process to verify optimal pyrometer placement during the experiment.

4. Conclusion

System development can be difficult enough without having to worry about I/O blocking a process or threaded programs causing a deadlock. This example detailed how coroutine and CSP models can be used to successfully manage multiple devices by isolating the I/O handlers. The decoupling of the blocking *sys−>read()* call, when managed with Limbo channels, can be a useful tool for separating out components of a program to process I/O.

Learning to leverage Limbo channels for inter-process communication may be a foreign idea when coming from other programming languages. Though channels behave like pipes in Unix, the ability to create typed data and easily pass it between processes enables the model to work quite well for concurrent programs. The implementation uses this feature to create independent byte stream readers that gracefully handle serial protocol changes while continually consuming input from external devices.

There are areas where this model could be improved. For one, the reallocation of the buffer array used to store bytes from the input stream can be optimized. Implementing a new data structure to eliminate the semaphore locking could facilitate programming logic simplification. For now, with the constraint of the serial line transmission speeds available to the remote devices, the system performs well enough to capture transmissions from the pyrometer at its maximum rate of 1kHz.

The Limbo source is available upon request from the author.

```
# wc results:

    1103    3215    21536 exactus/exactus.b
     196     578     4055 exactus/exactus.m

    1143    3775    27058 modbus/modbus.b
     248     787     5897 modbus/modbus.m

     424    1276     8449 zaber/zaber.b
     104     248     1944 zaber/zaber.m

    1515    5351    41132 aal/appl/pyro/pyro.b
     365    1035     9056 aal/appl/pyro/pyroplot.b
      29      72      556 aal/module/pyroplot.m

    5127   16337   119683 total
```

5. Acknowledgements

The author wishes to thank Belma Hadziselimovic and Omer Hadziselimovic for providing advice on the proper use of the English language, and Jason Bubolz for providing a critical review.

The original work was performed under contract to Physical Property Measurements, Inc. for RWTH Aachen University Institute of Mineral Engineering. Funding for this project was provided by the German Research Association number DFG: INST 222/779-1 FUGG and the Federal state of North Rhine-Westphalia number NRW: 121/4.06.05.08.-566.

6. References

[1] Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (July 1963), 396-408. DOI=10.1145/366663.366704 http://doi.acm.org/10.1145/366663.366704

[2] E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (September 1965), 569-. DOI=10.1145/365559.365617 http://doi.acm.org/10.1145/365559.365617

[3] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (August 1978), 666-677. DOI=10.1145/359576.359585 http://doi.acm.org/10.1145/359576.359585

[4] C. A. R. Hoare. 2004. *Communicating Sequential Processes*, current edition published on-line at http://www.usingcsp.com/.

[5] G. Kahn and D. B. MacQueen. 1977. Coroutines and networks of parallel processes, Information Processing. In *Proceedings of IFIP Congress,* 77:993-998. http://hal.archives-ouvertes.fr/inria-00306565/.

[6] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms,* pp. 193-231. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[7] Dennis M. Ritchie. The Limbo Programming Language, In *Inferno Programmer's Manual, Volume Two*, pp. 91-100. Vita Nuova Holdings Ltd., York, 2000.

[8] Jeff Sickel, and Paul C. Nordine. 2010. Effective Resonant Frequency Tracking With Inferno, In *Proceedings of IWP9*, 2010: 42-52.

[9] Sean Dorward, Rob Pike, and Phil Winterbottom. 1997. Programming in Limbo. In *Proceedings of the 42nd IEEE International Computer Conference* (COMPCON '97). IEEE Computer Society, Washington, DC, USA, 245-.