# Active Serial Port: A Component for JCSP.net Embedded Systems

Sarah CLAYTON and Jon KERRIDGE
*School of Computing, Napier University, Edinburgh, Scotland*

**Abstract**. The `javax.comm` package provides basic low-level access between Java programs and external input-output devices, in particular, serial devices. Such communications are handled using event listener technology similar to that used in the AWT package. Using the JCSP implementation of active AWT components as a model, we have constructed an active serial port (ASP), using `javax.comm`, that gives a channel interface that is more easily incorporated into a distributed JCSP.net collection of processes. The ASP has been tested in a real-time embedded system used to collect data from infrared detectors used to monitor the movement of pedestrians. The collected data is transferred across an Ethernet from the serial port process to the data manipulation processes. The performance of the JCSP.net based system has been compared with that supplied by the manufacturer of the detector and we conclude by showing how a complete monitoring system could be constructed in a scalable manner.

## 1. Introduction and Motivation

For some time research has been pursued to investigate the use of low-cost infrared detectors to monitor the path of pedestrians as they move through a space [1]. Of particular interest is the microscopic changes of direction people make as they interact with each other in a confined space. Such data is important when designers of spaces are considering the layout to ensure that movement is as efficient, pleasant and as free from conflict as possible.

The detectors we have investigated, manufactured by a British company IRISYS Ltd [2, 3], have a maximum field of view of 4m square and thus, to monitor a large area, a number of such detectors are required. Each detector generates a serial output data stream, which contains image data, counts of pedestrians crossing user defined datum lines and the instantaneous location of all the pedestrians in the field of view. The detector comprises a 16 by 16 pixel array and has an associated Digital Signal Processor, which undertakes image processing functions to fit an ellipse to each pedestrian that is then used to determine counts of pedestrians across datum lines and the sub-pixel location of each person.

The data output by the detector can be restricted to a subset of the available data. For the particular application we require only the location of each target in the field of view. The detector outputs serial data, with no flow control, equivalent to a frame rate of 30 frames per second. Each target in the field of view generates 34 bytes of data. Each frame of data requires a further 23 framing bytes. Thus the total data transfer for each frame of data comprises 23 + (n * 34) bytes where n is the number of people in the field of view of the detector. For normal spaces a suitable maximum value for the number of people, n, would be 10. Thus the total data transfer required for a single detector would be 10860

bytes per second. In more normal situations, we have rarely seen more than 6 people in the field of view at any one time, which gives a data rate of 6780 bytes per second. The serial communication port on the detector operates at 115k baud and thus the serial port should operate at about half its capacity.

## 2. Process Structure of the Data Collection System

Previous work undertaken by Kerridge had shown that it was possible to build a multi-process system using the JCSP Network Edition (JCSP.net) [4] that was capable of tracking the path of pedestrians as they move through a corridor, monitored by three detectors [5]. In this case, the data was read from files of data that had been captured by the detector manufacturer's software systems. The data was then read from the files at the equivalent of real-time using a JCSP `CSTimer` built into the system. This paper will not concern the design of that part of the system, except to confirm that we could deal with three detectors operating at an equivalent speed of 30 frames per second on an 850MHz processor. The system had been designed so that processes that accessed the detectors directly could replace the file reading processes. At the simplest level therefore the process architecture associated with a single detector is shown in Figure 1. This is broken down into two distinct parts. The first part, called the Detector Part, is concerned with receiving data from the detectors, parsing this into data objects that can then be transmitted over TCP/IP using JCSP.net. The second part is the Process Target Data (PTD) part. This receives the data sent across the Ethernet and then passes it on to the software that extract pedestrian trajectories from the data.

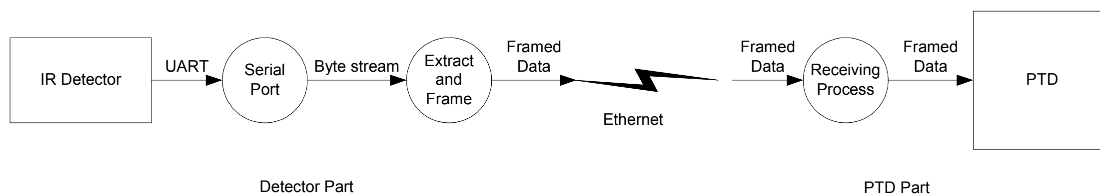


**Figure 1:** Initial System Design

The structure shown in Figure 1 assumes that there is a direct connection between each detector and the process that is going to analyse the target information. In practice, this would be a very limiting design, as each detector would need its own processor. Further, when a pedestrian leaves the field of view of a detector the path taken (as a series of [x,y] co-ordinates) has to be passed to an adjoining process where the pedestrian is likely to be observed next. For this to be efficient, it is better to have several Process Target Data processes in the same processor. If we have a large number of detectors it is also difficult to connect many serial ports to the same processor. In addition there is a limit to the length of a serial cable, which will limit the placement of the detectors and the processing system. A revised design was therefore proposed in which a relatively small number of detectors would be connected to a simple single board computer, which then used wireless technology to transfer packets of data to the processor dealing with the Process Target Data processes as shown in Figure 2.

The system comprises a number of Single Board Systems each connected to a number of detectors mounted physically close to the detectors. Data is read from the detector by means of the Active Serial Port process and a stream of data values are passed to the Extract and Frame process, which processes the data stream to extract the individual numbers that make up the data frame for each target (pedestrian) in the field of view of the

detector. This data for each target is placed into a data frame object. The data frame from each of the detectors is packed into a single object for those detectors connected to the same single board system. It is this packed data that is sent across the network. Once the data has been received it is unpacked and each data frame is then communicated to the appropriate process that processes the target data from that detector.

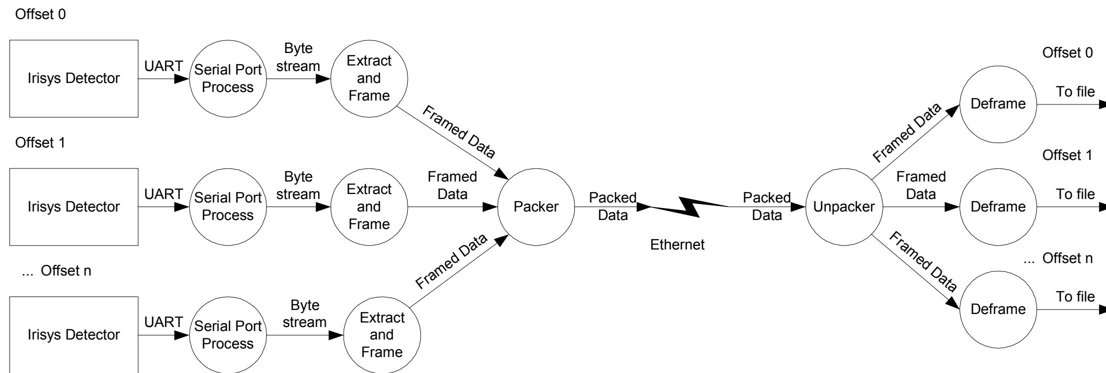


**Figure 2:** Overall System Design

## 3. Design and Implementation of Active Serial Port

The `javax.comm` API, released by Sun Microsystems and available as a download, provides Java with serial and parallel port functionality. Although this is often considered as an unfinished API by Java developers, it still provides all the necessary functionality for proper serial communications. In order to make the API portable across platforms, the API defines an abstract `SerialPort` class. This class is then subclassed and platform specific functionality is implemented in the subclassed object. For example on the Microsoft Windows platform, the class that implements the functionality is called `Win32SerialPort`. This concrete class then interacts with a Dynamic Link Library (DLL) file through the Java Native Interface (JNI). The applications programmer need have no knowledge of platform specific issues in managing serial communications, as these are provided through the concrete implementation of the abstract `SerialPort` class. Once a `SerialPort` object has been created, communications through the physical port are conducted through standard `InputStream` and `OutputStream` objects. These streams send and receive information as bytes, integers or arrays of bytes.

The `SerialPort` listener, the `SerialPortEventListener`, communicates 10 possible serial port events specified by the `SerialPortEvent` class.

According to Niemeyer and Knudsen [6]:

> *"Swing and AWT events are multicast; every event is associated with a single source but can be delivered to any number of receivers. When an event is fired, it is delivered individually to each listener on the list."*

This is not true of the classes in the `javax.comm` API. The `SerialPort` object is limited to only one listener, of one type, `SerialPortEventListener`. As Niemeyer and Knudsen [6] continue:

> *"If an event source can support only one event listener (unicast delivery), the add listener method can throw the `java.util.TooManyListenersException`."*

JCSP does not approve interaction between *processes* (i.e. *active* objects running in their own threads) other than through its various channel mechanisms (or other CSP-based synchronization primitives, such as multiway events). This is at odds with standard practices in OOP. In an *event-driven* context, the listener mechanism, or more strictly implicit invocation by the Java Event Thread, is the standard method for separate objects to respond to events. The architecture of Java is built entirely on these principles. Therefore, there is an architectural mismatch between the *process-oriented* JCSP and standard Java. This leaves the question of how to leverage the wealth of existing Java classes so that they can work as processes in a JCSP design. An answer is given by the JCSP AWT classes, provided with the JCSP API. These extend the Java AWT classes, allowing them to be run as processes, providing User Interface elements to JCSP applications. Insight into how the developers of JCSP overcame these problems was gained through decompilation of the JCSP AWT classes and from discussion with Welch [7]. The design patterns used have then been applied in the implementation of the serial port process.

To convert event-driven Java objects for use in JCSP, replace the listener and configure mechanisms with channel communications. For example, in JCSP AWT classes:

- the listener interface is implemented in a (hidden) event handler class, which communicates events through the JCSP AWT class' (published) output channel. This event handler class is added as a listener to the JCSP AWT class' superclass.
- a class based mechanism for configuring the JCSP AWT class at run time is provided via its (published) configure channel.

One of the concerns about JCSP was that, once a process had started, there was no obvious way of configuring that process; it would simply execute its `run()` method until it terminated. However, the designers of JCSP provided their JCSP AWT classes with inner interfaces called `Configure`: to configure, simply send them a `Configure`-implementing object along their configure channels.

The ActiveSerialPort (ASP) process cannot be implemented in quite the same way as the JCSP AWT [7] classes because `SerialPort` is an abstract class, thus it cannot be instantiated directly or derived from. It therefore has to be added to ASP as an attribute.

The need for the propagation of events in this context is far more straightforward than in a User Interface setting. Although the fact that `SerialPort` only supports one listener is often a matter of complaint among programmers using the `javax.comm` API, for our purposes there is no need for these events to be propagated beyond the ASP class itself. Therefore ASP implements the `SerialPortEventListener` interface itself, in addition to `CSProcess`, rather than this being delegated to a concrete event handler as in JCSP AWT. The UML class diagram for ASP is shown in figure 3.

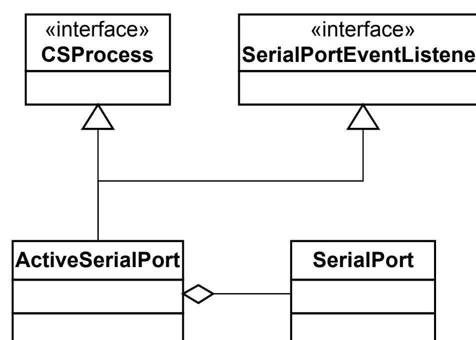


**Figure 3:** UML Diagram for ASP

The JCSP AWT classes, in general, had a single input channel and a single output channel. The situation in this case is more complicated however. In the final design, ASP has three channel interfaces: configure channel (`configure`), input channel (`input`) and an output channel (`output`). These are shown in the process diagram given in Figure 4.

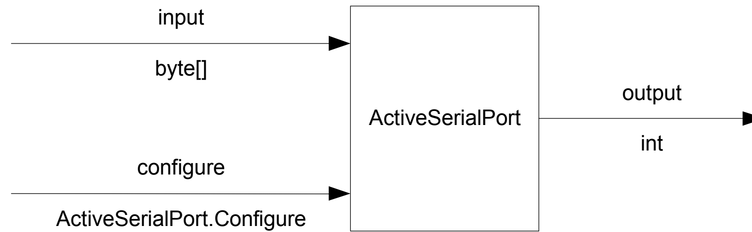


**Figure 4:** Process Diagram for ASP

The `configure` channel of an `ActiveSerialPort` is a `One2OneChannel` that expects an `ActiveSerialPort.Configure` object. It was decided early in the design process that configuration information and data input to ASP should be sent along separate channels. This is in line with the standard practice of separating control information and data. The design pattern, of specifying a `Configure` interface for the process, similar to that used in `jcsp.awt` classes, is implemented here. The `SerialPort` class requires a combination of settings that set flow control, baud rate, parity and stop bits. These need to be set up correctly for communication to proceed. It was decided, in order to avoid the returning of erroneous data from ASP, that no data should be sent from it until the object had been configured once.

On its own, ASP does not represent an API. Its workings as a process are summarized in Figure 4; it has a pure channel interface. However, as stated above, serial ports require a great deal of configuration information to work correctly. In addition to this, there are many separate notification options available to the programmer. Although in our application these are not used, the Irisys detectors employ a 'naïve' communications protocol with no flow control, the `java.comm` API lists some nine different notification events. These range from notify on ring indicator to notify on framing error, and encompass a number of events relevant to flow control. In order to make these available when using ASP, an `ASPConfigure` object was created, that inherits from `ActiveSerialPort.Configure`. This takes flow control, baud rate, parity, stop bit and notification information as parameters in its constructor. Once initialized, it can be communicated to the ASP's `configure` channel. As shown in the listing below, it is relatively simple to implement and to use.

```
public class ASPConfigure implements ActiveSerialPort.Configure
{

  int notify;
  int baud;
  int databits;
  int stopbits;
  int parity;

  public ASPConfigure(int baud, int databits, int stopbits, int parity)
  {
    this (baud, databits, stopbits, parity, 0);
  }
```

```
  public ASPConfigure (int baud, int databits, int stopbits,
                       int parity, int notify)
  {
    this.baud = baud;
    this.databits = databits;
    this.stopbits = stopbits;
    this.parity = parity;
    this.notify = notify;
  }

  public void configure (ActiveSerialPort s)
  {
    s.setPortParameters (baud, databits, stopbits, parity);
    if (notify != 0)
      s.setNotify (notify);
  }

}
```

The input channel of an `ActiveSerialPort` is also a `One2OneChannel` that expects data sent along it to be an array of bytes. `SerialPort`'s `OutputStream` can then write out that array of bytes to the port using one method call.

The output channel is `One2OneChannelInt`. The reason for using an `int` carrying channel requires some explanation, which will be given here. JCSP channels, other than the `int` channels, are typed so that only `Objects` may be sent along them. All Java classes, including arrays, are derived from `java.lang.Object`. However, as the serial port, potentially, produces an unlimited amount of new data, the manner in which this is returned requires some thought. According to Lindholm and Yellin [8]:

> *"There are three kinds or reference types: the class types, the interface types, and the array types. An object is a dynamically created class instance or an array. [...] An object is created in the Java heap, and is garbage collected after there are no more references to it. Objects are never reclaimed or freed by explicit Java language directives."*

The `SerialPort`'s underlying `InputStream` is able to read data from the port either as an array of bytes, or one byte at a time. While it is possible to declare new arrays at each read, this can very quickly use up system resources. As stated above, the reclamation of these resources is not within the control of the Java programmer. It is governed entirely by the Java Virtual Machine's (JVM) garbage collector. The garbage collector is a low priority thread that from time to time is activated and searches the Java heap for objects that no longer have any references to them within scope. Object creation involves the allocation of resources on the heap, and the creation of a reference on the stack. In Java, objects are often referenced in more than one place. If, however, there are no references to the object within scope, the garbage collector reclaims these resources when it is run.

C++, in comparison, does not have this problem because class destructors can be specified, the `delete` keyword will invoke its operand's destructor to reclaim resources. However, this must be done explicitly, and any oversight in this regard leads to memory leaks, where orphaned objects stay on the heap without any reference to them on the stack. The aim of the Java garbage collector, a system that has been adopted for other languages such as C#, is to remove responsibility from the programmer for object deletion and resource reclamation.

A simple solution to this dilemma is to completely avoid object creation at the outset. This is particularly important in this situation. The target platform for this software is an embedded machine, with limited resources in terms of either processor time or memory. Venners [9] also states:

> *"Heap fragmentation occurs through the course of normal program execution. [...] On an embedded system with low memory, fragmentation could cause the virtual machine to 'run out of memory' unnecessarily."*

The `One2OneChannelInt` interface provides an immediate solution to this problem. Unlike all other *JCSP* channels, the `Int` channels send only integer values. These are sent by value, not by reference. Primitive data types, such as `byte`, `int`, `float`, `double` and so forth, are created and allocated resources on the stack. These resources are automatically reclaimed when they pass out of scope. The heap is entirely unaffected.

Testing showed the software ran sufficiently quickly that, even when the serial port was running at 115200 baud, that there was no blocking when using simple integers rather than arrays of bytes to send data from ASP.

Only the input channels are dealt with entirely in ASP's `run()` method, as is consistent with *JCSP* programming principles. They form the guards in a `pri.select()` of an `Alternative` as shown in the following code snippet:

```
public void run ()
{

  Guard[] guards = {configure, input};
  final int CONFIGURE = 0;
  final int INPUT = 1;
  Alternative alt = new Alternative (guards);

  while (true)
  {
    switch (alt.priSelect())
    {
      case CONFIGURE:
        handleConfigure ();
        break;
      case INPUT:
        handleSerialWrite ();
        break;
    }
  }

}
```

The output channel is not handled directly in the `run()` method, however. It is only used when the `readFromSerial()` method is invoked, which only happens when the `DATA_AVAILABLE SerialPortEvent` is generated. The `SerialPort` object, running in the Java Event Thread, calls this method implicitly through the listener mechanism, whenever data is available. A criticism often levelled against Java's thread model, by Welch[10] and Hansen[11] amongst others is that threads can access an object's methods and data irrespective of whether or not it is ready to service that request. Objects become vulnerable to arbitrary requests at any time, irrespective of their state. For our purposes however, this makes ASP extremely efficient at reading from the serial port. The code is shown in the following snippets.

```
public void serialEvent (SerialPortEvent event)
{
  switch (event.getEventType ())
  {
    case SerialPortEvent.BI:
    case SerialPortEvent.OE:
    case SerialPortEvent.FE:
    case SerialPortEvent.PE:
    case SerialPortEvent.CD:
    case SerialPortEvent.CTS:
    case SerialPortEvent.DSR:
    case SerialPortEvent.RI:
    case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
      break;
    case SerialPortEvent.DATA_AVAILABLE:
      readFromSerial ();
      break;
  }
}
```

The `readFromSerial()` method shows how the data is read from the serial stream into an integer variable and only written to the output channel if the `SerialPort` has been configured. The `output` channel is connected to the Extract and Frame process.

```
private void readFromSerial ()
{
  int i = 0;
  while (inputStream.available () > 0)
  {
    i = inputStream.read ();
    if (configured)
      output.write (i);
  }
}
```

## 4. Extract and Frame Process

The Extract and Frame process (EFP) is specific to this application but has a generally applicable process structure shown in Figure 5.

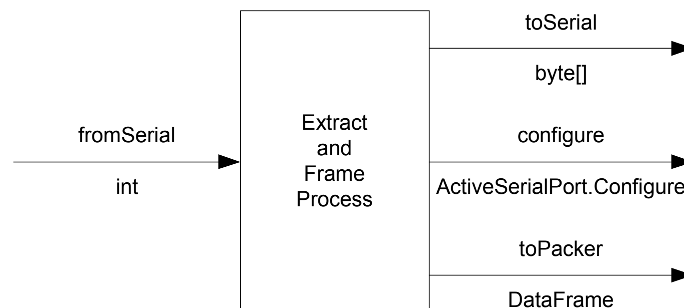


**Figure 5:** Process Diagram for Extract and Frame Process

EFP has one input channel, a `One2OneChannelInt`, that is unbuffered. It receives integers from ASP, as discussed in the previous section, that carry the bytes returned from the serial port.

EFP has three output channels that are all unbuffered `One2OneChannels`. They are:

- `toSerial`: this sends an array of bytes to be written to the serial port by ASP;
- `configure`: this sends an `ActiveSerialPort.Configure` object to the `ActiveSerialPort` process, which configures the ports settings;
- `toPacker`: this sends the parsed `DataFrame` object to the packer process.

The data sent by the sensor is limited to `shorts` and `floats`. The internal DSP uses the IEEE 754 Floating Point Format. Extracting the data values from the input byte stream was simply achieved by byte shifting to accommodate *endian* differences between Java and the DSP and, for floating point values, a call to the `Float.intBitsToFloat(int)` method to convert the resulting 32-bit pattern into a `float`. It was possible to do comparison testing of the data by capturing the byte stream passing through the serial port of a machine running Irisys' proprietary software using a serial port logging application. The bytes captured were then sent through the parser and the values returned were compared against the results given by the Irisys software. The results showed that these were entirely in agreement with each other and that the assumptions made about byte ordering and data formats were correct.

As discussed previously, this software is intended for use on an embedded machine, and therefore there is a need for reducing the amount of object creation to an absolute minimum. EFP sends `DataFrame` objects, containing an array of `TargetData` objects, through its output channel. The `TargetData` objects contain the parsed target data for each target (if any) in the parsed packet. Potentially, this could involve a great deal of object creation. As stated by Welch [10], *JCSP* channels (other than the `Int` channels) do not transmit *copies* of classes, they transmit *references* to the classes. Consequently, the sending and the receiving process can manipulate the same object at the same time and thus two processes can then, potentially, use that object's data and methods at the same time while running concurrently. This is known as *aliasing*, which is an endemic problem in Java, and in this situation would be a race hazard.

However, with careful thought and consideration, it can also be used to our advantage, and it is used in EFP to implement a *double buffer* between EFP and the `Pack` process. This is done as follows:

- an array of two `DataFrame` objects is declared and initialized in EFP;
- an index `counter` is also declared and initialized;
- before parsing, the active `DataFrame` is indexed from the array using the index counter;
- the active `DataFrame` is cleared of all data, as are its constituent `TargetData` objects;
- once the parsing is complete, the `DataFrame` is written out to the channel connecting EFP to the `Pack` process;
- the index `counter` is changed to either 0 or 1, depending on its value. The next parse then occurs using the other `DataFrame` object in the array.

To avoid aliasing race hazards downstream, we *copy* the received `DataFrame` in the `Pack` process. For this, we have chosen to use static functions that have the prototype:

```
SomeClass.copy (SomeClass src, SomeClass dest);
```

There are a certain number of advantages to using static functions in this way. Static functions are defined at the class rather than the instance level. They do not require the programmer to have a handle on the object to be called. Where data is being received through channel communication, as in this case, it removes the need to declare and initialise a reference pointer, and cast the input value to the correct type. This helps to make the code more readable, as the example shown below shows. We read from each sensor by means of a parallel read using JCSP's `ProcessRead`. The values obtained are copied into a local `DataFrames` vector and then added to the `packedData` object.

```
ProcessRead[] processReads = new ProcessRead[toPacker.length];

for (int i = 0; i < toPacker.length; i++)
  processReads[i] = new ProcessRead (toPacker[i].in());

CSProcess p = new Parallel (processReads);

while (running)
{
  p.run ();
  for (int i = 0; i < toPacker.length; i++)
  {
    DataFrame.copy((DataFrame) processReads[i].value, dataframes[i]);
    packedData.addFrame (dataframes[i]);
  }
  toNet.write (packedData);
}
```

This `Pack` process simply outputs the packed `DataFrames` saved in `packedData` over the Ethernet connection to an `Unpack` process in the main processing system, thence to be passed to the subsystem that processes target data.

Although effective, this does not represent a solution to the problem intrinsic to JCSP, in that two processes are *potentially* free to use references to the same object at the same time. For our design, that freedom is curtailed by the synchronisations forced by JCSP channel communications to make it safe. All EFP processes have two (`DataFrame)` buffers, which they use alternately. Having sent one to this `Pack` process, they work on their other buffer. When filled, they commit to send it to this process, but that blocks until this process takes them (`p.run`). That won't happen until this process has copied, packed, forwarded and finished with the first buffers from the EFPs, and looped around. In each cycle, `Pack` and the EPF processes work on different sets of `DataFrame` buffers – with the switch-over safely coordinated by the `p.run` communications.

## 5. Performance Evaluation of ASP

In order to test the software adequately, a number of outputs were added to the processes in the application. Figure 6 shows the outputs from each component.

The EFP process has two outputs: a binary output, which records every byte received from the ASP process, and a parsed data file, which records the result of the parsing operation. `Pack` records the metrics of three operations: parallel reading from all its channels, the time taken to copy the data into the `PackedData` object, and the time taken to write the `PackedData` object to the `One2NetChannel`. The `Deframe` process merely streams out to file all the data it receives.

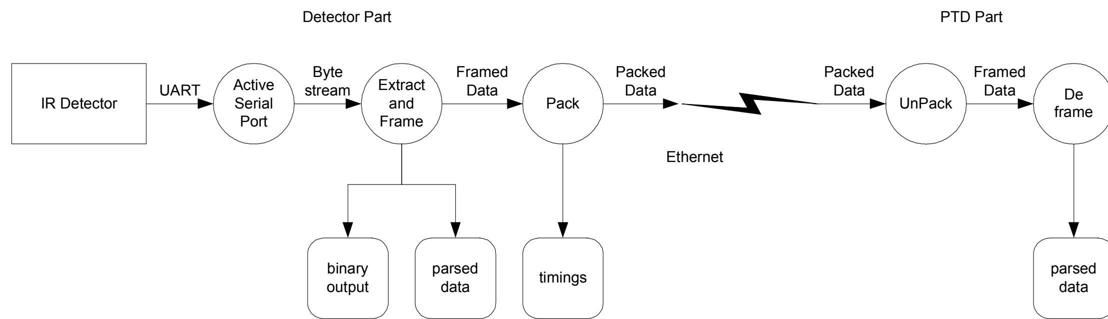


**Figure 6:** Test Outputs from the Software

If the system is working correctly, the parsed data output by the EFP process should be exactly the same as the data recorded by its analogous `Deframe` process. In this way, the correctness of the Ethernet channel communications can be tested. This does not test the parsing process itself, which was proved to be correct through comparison testing with Irisys' own software. For the sake of clarity, figure 6 shows only one set of ASP, EFP and `Deframe` processes. The intended system will have an arbitrary number of these running in parallel. As such, the tests defined in Table 1 all involve reading from between 3 to 4 serial ports.

**Table 1:** Allocation of Processes and Detectors to the Available Processors

| | 2 Ghz Athlon | 400Mhz Pentium | Detector 1 | Detector 2 |
|---|---|---|---|---|
| **Test 1** | DetectorPart<br>PTDPart | - | 4 ports | - |
| **Test 2** | DetectorPart<br>PTDPart | - | 2 ports | 2 ports |
| **Test 3** | DetectorPart | PTDPart | 4 ports | - |
| **Test 4** | DetectorPart | PTDPart | 2 ports | 2 ports |
| **Test 5** | PTDPart | DetectorPart | 3 ports | - |
| **Test 6** | PTDPart | DetectorPart | 2 ports | 1 port |

The machines used for testing are a 2 Ghz Athlon with 1Gb RAM running Windows XP Pro and a 400MHz Pentium with 128Mb RAM running Windows 2000. Serial cable splitters were used, so that one detector could potentially be connected up to four serial ports. Although the Athlon could be upgraded to four serial ports, the Pentium could only be upgraded to three. This is reflected in Table 1, which gives an overview of the tests that were carried out. Two separate Irisys detectors of the same model were used. Splitting the signal from one detector to two or more ports has the effect of stressing the serial port and parsing processes, as they will all receive data at the same time. At the same time, this is also a good test of parallelism in action. Processing the outputs from two or more detectors is a good test of the efficacy of the `Pack` process, as this must handle inputs from three to four EFP processes in parallel. The binary output file, that recorded every input from the serial port process, permitted comparison between ports that received data from the same detector. Where parse errors occurred, it was possible to check the binary output and rule out errors in the parser itself.

Each test was run for two minutes. The metrics collected are presented here as average times, in milliseconds, for each packet to be parsed and each `PackedData` object to be written out to the Ethernet. The parsed data output file generated by each EFP process could be directly compared, byte for byte, with the file generated by its analogous `Deframe` process. In all of the tests, the files were exactly identical. The sharing of buffers between the EFP processes and the `Pack` process might allow for some data to be corrupted, if that sharing were not properly synchronised by the channel communication of their references. As all the output data files were identical, this shows that this never happened during any of the tests. Parse errors were only detected in the last test, where two detectors were connected to the slower 400Mhz Pentium. No other parse errors were reported.

Metrics were calculated for the parsing of each target in the byte stream, for each port. They were also calculated for the operations of the `Pack` process. Tables 2 and 3 shows the average time in milliseconds for each of the operations. There is a striking similarity in the network performance between the slower Pentium and the Athlon. However, as expected, the average parse operations were significantly slower.

**Table 2:** Average times for the Extract and Frame Process in milliseconds

| | Port 1 | Port 2 | Port 3 | Port 4 |
|---|---|---|---|---|
| **Test 1** | 29.217 | 29.217 | 29.213 | 29.213 |
| **Test 2** | 30.531 | 30.516 | 29.362 | 29.362 |
| **Test 3** | 31.947 | 31.947 | 31.947 | 31.947 |
| **Test 4** | 31.510 | 31.509 | 32.579 | 32.579 |
| **Test 5** | 34.693 | 34.707 | 34.696 | - |
| **Test 6** | 35.683 | 35.729 | 35.706 | - |

In Test 1 and Test 2, the time taken to read from all the channels in parallel was significantly higher than all the other tests. This is explained by the fact that, as these tests were conducted using the loopback address, network operations were only on average 3.77 milliseconds. Therefore, the process would have been waiting on input from its channels. As Table 3 shows, the times taken by the `Pack` process were overall very similar. From this we can conclude that the determining factor is the speed at which the detector sends its data, rather than the overhead of the network operation itself.

In Test 4 and Test 6, where two detectors were attached through serial cable splitters to each machine, the resulting time taken was exactly the same on the Athlon as the Pentium. This suggests that the difference between detector 1 and detector 2 sending their data accounts for the greater time taken in packing the data for transmission across the Ethernet.

**Table 3:** Average times for the `Pack` process in milliseconds

| | Parallel read | Pack Data | Write to Ethernet | Total |
|---|---|---|---|---|
| **Test 1** | 28.447 | 0.0339 | 3.841 | 32.322 |
| **Test 2** | 28.515 | 0.0376 | 3.718 | 32.271 |
| **Test 3** | 11.630 | 0.0294 | 20.679 | 32.338 |
| **Test 4** | 15.279 | 0.0870 | 19.785 | 35.151 |
| **Test 5** | 14.298 | 0.0711 | 19.663 | 34.032 |
| **Test 6** | 15.279 | 0.0870 | 19.785 | 35.151 |

## 6. Building a Scalable System

The test results given above demonstrate that a system with multiple detectors can be constructed and that the data can be transmitted over an Ethernet to a processing subsystem. The design is thus inherently scalable. The limit to scalability will in fact be determined by the number of people in the field of view of the detectors and the effect this has on the accuracy of the system in terms of the number of frames that will be lost and the effect this has on the performance of the system in tracking individuals from one detector's field of view to that of an adjacent one. Calculations suggest that if there are more than 10 people in the field of view then the time taken to transfer the data over the serial link is longer than the time available when the detector is working at 30 frames per second. From the evidence above, this would seem to be a more pressing limit to scalability than the performance of the underlying processing system.

The likely scenario would be a number of sensors connected, by wire, to an embedded system that undertakes initial data capture and immediate processing and then sends the data across a wireless network. The increasing availability of plug in wireless components that can be attached to the Ethernet port of such embedded Java based systems means that the size of the sensor layout is limited only by the coverage of the wireless network in the area to be monitored.

## 7. Conclusions and Further Work

The ASP described in this paper has been designed specifically for the infrared sensor application using PCs rather than embedded systems. For it to be used in a genuine embedded system some modifications would be needed because reporting errors on `System.out` is not feasible. However the modification is in fact quite simple. An additional output channel from ASP is introduced to the application process connected to it. This channel is used to send error values, which can then be interpreted by the application process. The package `javax.comm` defines most of the required error values and all that would be required are some additional error values pertaining to initialization and configuration.

This paper has demonstrated that it is possible to construct real-time systems using JCSP.net, which operate under constraints imposed by the devices that are connected to the system. Furthermore, the system is inherently scalable and with advances in modern embedded systems technology and the use of wireless network technologies the applicability of the approach is not limited to applications that rely on wired connections between components.

The next stage in the development is to use a number of sensors working together to monitor a larger area. This however, is not a problem for the input of the data from the sensors because we have demonstrated its feasibility. The challenge in the next phase is to track a person as they move from one field of view to that of another detector. We have achieved this in a corridor application, where the movements of pedestrians are confined, essentially to one direction [5]. When monitoring a rectangular area the complexity of processing increases and yet again we would expect to deploy parallel processing techniques to solve this problem

## Acknowledgements

IRISYS Ltd, the manufacturers of the infrared detector, have given us access to confidential information concerning the internal operation of their detector, which we gratefully acknowledge. Sarah Clayton acknowledges the funding provided by the Student Awards Agency for Scotland, who paid her tuition fees for the undergraduate degree of which, the work reported in this paper formed her final year project. Discussions with colleagues, Alistair Armitage, David Binnie, Frank Greig and Tim Chamberlain are gratefully acknowledged. This research has been, in part, supported by a grant from the UK Department of Transport in the LINK programme Future Integrated Transport with the project PERMEATE (GR/N33706).

## References

[1] A. Armitage, T.D. Binnie, J.M. Kerridge and L. Lei, "Measuring Pedestrian Trajectories with Low Cost Infrared Detectors: Preliminary Results", Pedestrian Evacuation and Dynamics – 2003, Galea, E.R. (ed), University of Greenwich, London, UK. 2003.

[2] M.V. Mansi, S.G. Porter, J.L. Galloway and N. Sumpter, "Very low cost infrared array based detection and imaging systems" (SPIE) Aerosense 2001, Orlando, Florida USA, 17-19 April 2001

[3] N. Stogdale, S. Hollock, N. Johnson and N. Sumpter (2003), "Array based infrared detection: an enabling technology for people counting, sensing, tracking and intelligent detection", SPIE, USE 3, 5071-94

[4] Quickstone Ltd. "An Introduction to the JCSP Network Edition". Retrieved November 20, 2003 from: http://www.quickstone.com/xcsp/jcspnetworkedition/ . 2003.

[5] J.M. Kerridge, A. Armitage, T.D. Binnie and L. Lei, "Monitoring the Movement of Pedestrians Using Low-cost Infrared Detectors: Initial Findings". 2004 Transportation Research Board, Washington, January 2004, paper 2185. 2004.

[6] P. Niemeyer and J. Knudsen, J. "Learning Java (2nd ed.)". Sebastopol: O'Reilly. 2002.

[7] P.H. Welch, private communication, January 2004, by email, concerning the structure of JCSP AWT components confirming a proposition sent to him.

[8] T. Lindholm and F. Yellin. "The Java Virtual Machine Specification". Reading: Addison-Wesley. 1997.

[9] B. Venners. "Inside the Java Virtual Machine". London: McGraw-Hill. 1998.

[10] P.H. Welch. "Process Oriented Design for Java: Concurrency for All". Retrieved, October 2003, from: http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp.ppt . 2002.

[11] P. Brinch-Hansen. "Java's Insecure Parallelism", ACM SIGPLAN Notices, Volume 34, Issue 4, 1999, Pages: 38 – 45. 1999.