1999

Designing an Integrated Environment for Artificial Intelligence

Andrew B. Ritger '99
Illinois Wesleyan University

Recommended Citation

Designing an Integrated Environment for Artificial Intelligence

Andrew B. Ritger and Dr. Lionel R. Shapiro
Department of Computer Science and Mathematics
Illinois Wesleyan University

April 26, 1999

Contents

List of Figures

Abstract

The SHELLEY RESEARCH GROUP (part of the Illinois Wesleyan Intelligence Network on Knowledge - IWINK) has been in existence for several years, and has benefited immensely from various student contributors who have added such components as robotic arm control, cross platform networking, an artificially intelligent tic-tac-toe player, and an interactive teaching tool demonstrating the functionality of artificial neural networks. What is lacking, however, amidst these undergraduate contributions to the SHELLEY Project, is an effective means of integrating existing components into a single cohesive functional unit, let alone any easy means of making further contributions within a simple unified context.

The focus of this research has been to design an all-encompassing structure for incorporating the different components of SHELLEY (both existing and future). Because we must operate under the assumption that we cannot predict what future contributions will be made to SHELLEY, nor how these components will be used, this integrated environment must be both flexible and expandable in such a way as to not confine future projects.

The approach to artificial intelligence that the SHELLEY RESEARCH GROUP has taken relies heavily upon interaction with the surrounding environment. For this reason, many of the existing components are devices for receiving input from SHELLEY's surroundings (such as vision cameras) or acting upon the surroundings (such as robotic arms). Thus, we can assume that future contributions will fall under two primary categories: additional devices (either cognitive modules, such as neural networks, or interactive devices, such as cameras or arms), or intelligent agents (such as tic-tac-toe players, or navigation systems) that will use these devices. The environment must then be flexible in two manners – allowing for the addition of further devices, and providing a task management mechanism for accessing these devices. The solution is to use a modern operating system model where the devices that SHELLEY uses to interact with her environment correspond to computer hardware devices and their drivers, the intelligent agents are analogous to processes that run on the system and use the devices, and the administrator, which coordinates these agents and their usage of devices, can be compared to the kernel of the modern operating system.

iii

1 Motivations

The primary purpose of this research has been to ease the implementation process for future students making contributions to the SHELLEY Project.[†] It is currently difficult for students to make contributions because each external robotic device has its own unique communication protocol. Additionally, even in the case where there exists a program that communicates with an external device (for example, a robotic arm), there is no system in place to facilitate other programs reusing that same portion of program code. Finally, a third obstacle hindering progress is intimidation felt by potential contributors. Students often examine the prospects of implementing a task for SHELLEY such as playing tic-tac-toe or chess, but may feel that the project is too daunting because "that's so complicated" or "I don't know anything about robotics." By providing a simple interface to an all-encompassing structure for incorporating the different components of SHELLEY, these difficulties can be alleviated.

2 The Requirements of an Integrated Environment for Artificial Intelligence

An *agent* is an entity that perceives characteristics of its environment and acts upon that environment. An *intelligent* agent acts upon its environment in ways humans consider appropriate to the characteristics that the agent perceives. It uses its *sensors* – means of perceiving its surroundings – to collect information that it then uses to make intelligent decisions. The agent then acts upon its surroundings through its *effectors* [1]. It is in this decision making, or *mapping* of input from sensors to output actions through the effectors, that the intelligence of the agent lies. This mechanism for agent intelligence, however, is not the focus of this research; there are many different approaches and techniques for making an agent intelligent, which encompasses several major paradigms and philosophies. It is also not the place of this research to make a judgment as to which paradigm is most appropriate for SHELLEY, but rather to design an environment that can facilitate different approaches to building an intelligent agent so that future students can explore the many options without feeling confined or restrained to one predefined paradigm.

The distinction can be made between pure software agents, whose world consists entirely of entities internal to the computer upon which the agent resides (*soft agents*), and agents whose world extends beyond the confines of a computer and encompasses the physical parameters of its surroundings [1]. In the former, the sensors and effectors are much more easily implemented, while the latter requires special hardware that introduces all the complications already discussed (see **Section 1 Motivations**).

SHELLEY is a robotic entity – her environment is the physical world. Therefore, while some tasks may be handled sufficiently through soft agents, others must be addressed by agents who make use of SHELLEY's special hardware peripherals in order to interact with the "real world" [2].

[†]The SHELLEY Project takes its name from *Frankenstein* author Mary Shelley

In light of the aforementioned difficulties associated with interfacing to these peripherals, we require some mechanism through which agents can easily access specialized external hardware in order to accomplish their tasks. Thus, SHELLEY necessitates an *integrated environment* that can provide an effective and flexible system for integrating both existing and future peripherals such that these devices can be shared and adequately managed. Additionally, this integrated environment must provide a simple method of programming with and using these peripherals. Ideally, this can exist in the form of function calls which can be made directly from within researchers' program code; however, the function calls must be structured such that they can easily accommodate new and different types of devices, as well as be used from any one of numerous programming languages.

3 The Operating System Analogy

In many ways the integrated environment that SHELLEY requires is analogous to a modern operating system. An operating system serves as an interface between the user-level software on a computer and the computer's hardware [3]. It has two primary tasks: provide convenience for the programs running on the computer and do so efficiently. The modern operating system can also be considered a control program that manages system resources (memory and processing time, as well as system Input/Output devices) by resolving conflicting resource requests and guaranteeing effective use of these limited resources. The operating system also controls application processes and ensures that the system is properly used by these processes [3].

One model of operating system is the one program running at all times on the computer – generally called the *kernel*. Following this, all other processes are applications that provide some functionality, either for the kernel, or for the user. It is the kernel that processes all system calls, handles all sharing of central processing unit (CPU) time and random access memory (RAM) between competing processes, and performs the handling of peripherals. To access peripherals, processes must do so through the kernel (see **Figure 1**) [4]. Another crucial part of an operating system is the capacity for multiprocess scheduling and management – a significant aspect of the modern operating system, and one that is crucial for our use of the model as an environment for artificial intelligence.

Given this formal model of an operating system, it can be used as a point of departure to construct our own model of an integrated environment for artificial intelligence. There are several differences, though, which merit attention before proceeding. The first of these differences is that when addressing the potential paradox of convenience and efficiency, operating system design has historically favored efficiency over convenience when the two have contradicted each other. In our model, however, though we emphasize both, convenience receives precedence when conflicts require resolution. As stated earlier, the primary goal of this work is to make more convenient the work of future users of this environment.

The second major difference is actually an issue of implementation that will be addressed later. For now, let it suffice to say that we employ a client-server model to facilitate communication, rather than use a method similar to that of system calls to the kernel. In a client-server model, we have two classifications of programs: clients, which make requests, and servers, which service the clients [5]. The client-server model may not be quite as efficient, but it certainly increases the convenience of our model. Thus, we see the influence of our favoritism for convenience over
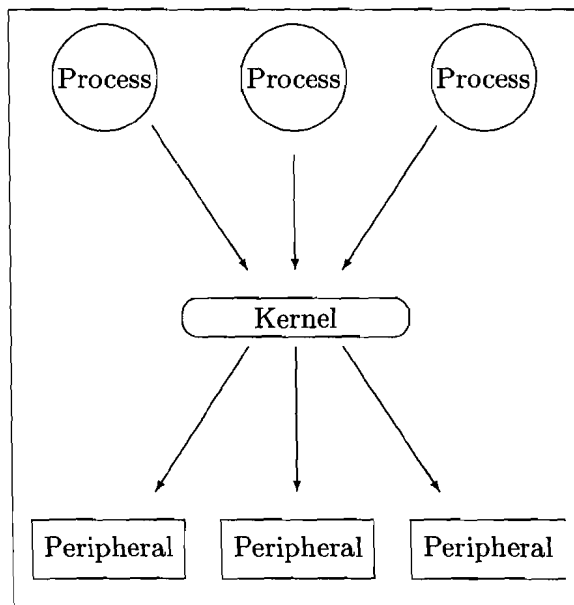
Figure 1: PROCESS/PERIPHERAL COMMUNICATION VIA THE KERNEL [4]

efficiency. The client-server model allows for network-ability, easier implementation, and allows this integrated environment to be built on top of, not replace, the existing operating system.

4 The SHELLEY Integrated Environment (SIE)

The operating system model discussed above (see **Figure 1**) offers the additional benefit of being completely modular: distinct functional units are separated into disjoint and independent components. These components then can be used to construct more complex structures, which in turn can be used as the components to build even more complex and powerful entities. This modularity provides a very simple and convenient way for developers to construct powerful systems by using the work of previous developers as building blocks. Additional advantages are that this form is very flexible and expandable, and that it facilitates program code reuse. It is with this model in mind that we have designed the SHELLEY INTEGRATED ENVIRONMENT (SIE).

4.1 The Agent/Administrator/Device Model

We cannot, however, have complete freedom in the modularity of SIE; there must be some constraints to define the relationship between these modules. Thus, we employ the operating system model as the basis for a similar structure to govern SIE: the agent/administrator/device model.

In this modular all-encompassing structure, there are three primary types of components: *agents*, which are programs for a specific task, *devices* which the agents use to accomplish these tasks, and the *administrator*, which intercedes between the other two modules, facilitating communication and regulating agent access to devices.

3

4.1.1 Intelligent Agents

As discussed earlier, *intelligent agents* are objects designed for a specific artificial intelligence task, such as navigating through a maze, playing chess, or performing speech recognition. Thus, to build an artificially intelligent entity, multiple agents would be run to accomplish each different behavior desired. In SIE, the intelligent agent is considered to be a software application written to accomplish a particular goal, following the general definition presented earlier of an agent as a mechanism mapping input from sensors to behavior through effectors. Agents in SIE are analogous to application processes in the operating system model.

4.1.2 Devices Modules

The agents, however, should not have to know the details of the resources (sensors and effectors) that they necessarily must use. Therefore, we call upon the *device module* to act as an interface between the agent requests and the physical robotic hardware. Device modules within the context of SIE encapsulate individual functional units. They usually control external hardware peripherals such as arms and cameras, though a device can just as easily contain a cognitive module such as an artificial neural network. Through device modules we are able to extract the implementation details of these functional units from the role of the agent programmer. These device modules are equivalent to an operating system's device drivers in that both the SIE device modules and the operating system's device drivers are software applications which facilitate the use of specific hardware by other software applications.

4.1.3 The Administrator

The challenge remaining is to integrate the agents and devices into a cohesive whole, allowing intelligent agents to use devices while still maintaining a relative degree of simplicity for the individual implementation of an agent. The solution is the *administrator* module which serves as a mediator between the agents and the devices that the agents use, much like the operating system kernel serves as a mediator between software applications and the hardware they must use. The administrator resolves all conflicts between multiple agents trying to control the same device. For example, if multiple agents require access to a robotic arm, but wish to move the arm in differing directions, it is up to the administrator to resolve this dispute. When an agent requires data from a device, the agent sends the request to the administrator who passes the request on to the device if the administrator deems the request admissible. When the device sends back data, the data is streamed to the administrator who channels it to the appropriate agent. In this way the complications of resource management are extracted from the agents and devices, and handled only by the administrator (see **Figure 2**).

4.2 Agent Ownership of Devices

Some devices such as one which interfaces with motor-driven wheels should only be controlled by one agent at any one time. Other devices such as one which acquires single frames of video from a camera may be used by multiple agents, but only one agent should have permission to make status changes to the device, for example, change the resolution or filtering mode in the case of a video frame acquisition device module. Given these stipulations, the administrator employs a mechanism for read/write permissions similar to that of a Unix operating system. If a device is designated as "sharable" then it can have an unlimited number of agents using it, though it can have at most one owner with full read/write permission at any one time (ownership equals write

4
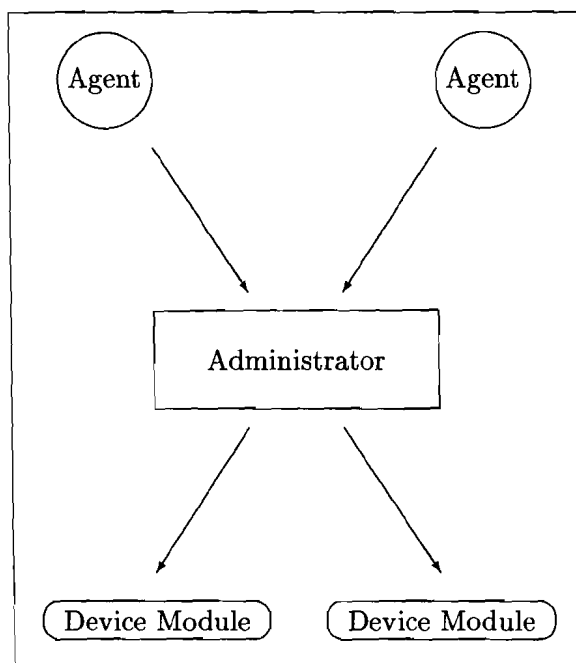
FIGURE 2: THE SIE AGENT/ADMINISTRATOR/DEVICE MODEL

permission); all other agents must use the device in read-only mode. The requests of a device which are considered read-only and those which require write permission must be explicitly made known to the administrator through a .conf file for each device (see **Section 4.4 The device_list File**).

This permission system introduces the additional complexity of determining which agent owns (has write permission for) a device. The administrator grants owner privileges to an agent for a specific device if the agent requests the device and no other currently connected agents own the device – either no other connected agents have requested the device, and therefore it is not yet connected, or other agent(s) are using the device, but the previous owner has relinquished ownership and no other agent in the interim has requested ownership.

Devices may also be specified to allow multiple instances. This mostly like would occur with cognitive modules, or at least devices which operate completely at a software level and do not interface with external peripherals. If an agent requests a device of this type, then each request will result in a new device of that type to be run. Therefore, an agent who requests a device of this type is guaranteed to be granted owner permissions because it is the only agent using that instance of the device.

To further facilitate sharing of devices and inter-agent cooperation, an agent can query the current ownership status of a device and receive a response of either (a) the querying agent owns the device, (b) another agent owns the device, (c) no agent owns the device, or (d) the device is not known. The agent can also make a request to the administrator to claim ownership of a device, receiving either a confirmation or rejection. In the case of a rejection, the administrator sets a flag which indicates that there are agents without ownership who desire ownership. Agents can query the status of this flag to know if other agents have been requesting ownership. Additionally, the agent

can relinquish ownership of a device. Through this system, multiple agents can effectively share a device by yielding ownership when they are able to operate in read-only mode, and by requesting ownership only when it is absolutely necessary to have write permission. Of course, handling of ownership issues is not necessary for an agent implementation; if an agent is not intended for use alongside other agents, then behaving in a "device-greedy" manner is completely acceptable.

4.3 The Flow of SIE

Just as the kernel runs the entire time the operating system is running on a computer, our administrator module runs as a background process whenever we are using SIE. If no intelligent agents are running, then the administrator simply waits, listening for agents. When an agent is run, it connects to the administrator, informing it of what devices are needed. After each device request, the agent listens for a response from the administrator, who processes the device request and makes one of the following responses if an error occurs:

- The requested device is currently in use by another agent, and the device is not sharable. The administrator gives the agent the choice of either exiting or continuing without the device.

- The requested device is currently in use by another agent, but the device is sharable. Thus, the agent can access the device, but will not have ownership permissions. The administrator gives the agent the choice of either exiting or continuing without ownership of the device.

- The requested device is unknown to the administrator. The administrator gives the agent the choice of either exiting or continuing without the device.

If a device is successfully connected to the administrator with the requesting agent as owner, then the administrator sends a confirmation to the agent. After each confirmation by the administrator (or acceptance of restrictions by the agent) the administrator sends the agent a unique identification number which the agent and administrator then use to refer to the device in all future communications. Note that this device identification number is not the same as the identification number listed in the device_list file. This identification number serves the purpose of allowing the agent and administrator to refer to a specific device, distinguishing between instances of the same device type. If an instance of a device is shared between multiple agents, then the same number is used by all agents to refer to this device. For example, if there is a video camera device module, an agent may require two instances of this module (one for each of SHELLEY's two cameras). Thus each instance of the module would be referred to by a different identification number so that the agent and administrator can distinguish between the two. Also, if multiple agents are sharing the same video camera module, then both agents use the same number to refer to the same device.

Such different situations may appear to make initialization of an agent overly complicated, but an agent could be written very simply by connecting to the administrator, asking for certain devices, and immediately failing if confirmation is not received. Depending on the context in which this agent will be run, this approach may be sufficient. However, in cases where the programmer wishes to maximize the stability of the agent and build support for inter-agent cooperation, handling of the above situations is necessary, along with the ownership issues discussed previously.

When an agent disconnects from the administrator, any devices used exclusively by that agent are also disconnected by the administrator. If the device is being used by other agents, then the device will remain until all agents accessing it disconnect.

6

4.4 The device_list File

When the administrator is started, it reads a device_list file that lists all the device modules that will be supported by the administrator. The device_list file must adhere to the following syntax: lines that begin with white space or pound signs ("#") will be ignored; lines which define a device must have the following information in order (separated by white space):

- A positive integer, to be used as a unique identification number for the device. This number is used both internally by the administrator and by agents referring to the device at time of request.

- The device executable name (with full path).

- The host computer on which the device should be run. This currently has no effect – all device modules are run on the host of the administrator (see **Section 7 Future Work** for a discussion of running devices on remote computers).

- Either **share** or **no-share** to designate if the device can be shared by multiple agents.

- An integer to designate the maximum number of agents which can share the device. If the number is 0, then there is no fixed maximum. If the device is designated as not shareable, then this value must still be here, but it serves no functional purpose.

- Either **multiple** or **no-multiple** to designate if the device can have only one instance or multiple instances.

- An integer to designate the maximum number of instances which can exist for that device. If the number is 0, then there is no fixed maximum. If the device is designated as **no-multiple**, then this value must still be here, though it is meaningless.

Note that **multiple** overrides **share**; if a device is marked to be both **multiple** and **share**, a separate instance of the device will be created at each request – the device will never be shared until the maximum number of instances for that device is achieved, at which point the device will be shared.

Below is a sample device_list file. The lines with pound signs are comments. Each other line specifies a device by executable name and unique id number by which both administrator and agents will refer to the device.

```
# Andy Ritger
# 4-12-99
# Research Honors
# sample device list file
5 /opt/local/shelley/devices/frame_grabber localhost share 0 no-multiple 0
1 /opt/local/shelley/devices/neural_net localhost no-share 0 multiple 0
17 /export/home/aritger/temp/mobot_wheels localhost no-share 0 no-multiple 0
```

The administrator also requires a file associated with the device in the same directory as the executable called <executable name>.conf (for example: frame_grabber.conf or neural_net.conf). These .conf files specify which device requests, if any, are considered to require write permission. If the administrator cannot find the file, it will produce a warning and proceed under the assumption that all requests require ownership to be performed.

4.5 Network Protocols and the Details of Inter-Module Communication in SIE

4.5.1 Choice of Communication Medium

The mechanism for inter-module communication within SIE is the *sockets Application Programmer's Interface* (API), following a client-server model where the administrator functions as the server, and the agents and devices function as clients. Sockets were chosen over other forms of interprocess communication (IPC), such as shared memory, pipes, and signals [6], because they are built on top of TCP/IP, and therefore facilitate the possibility of networking and distributing module execution over the different computers that comprise SHELLEY.[†] Sockets are a sequenced, reliable, fast, bidirectional means of interprocess communication through variable length streams [4]. The sockets are of type SOCK_STREAM and domain AF_INET, which allows the client-server to connect and communicate anywhere on the Internet [7].

To ease future implementation, several C wrapper functions are provided for simple socket creation and use (shelley_sockets.h and shelley_sockets.c; see **Appendix C** and **Appendix D**, respectively). Future contributors are not bound to use the SHELLEY_SOCKETS mini-library, but the basic facilities are provided and a complete – though rudimentary – client-server structure can be built solely with SHELLEY_SOCKETS function calls. **Figure 3** gives a detailed chart of the layers of protocol used in SIE.

4.5.2 Defining a Communications Protocol

In his 1997 paper, Douglas Gage discusses the obstacles he has encountered in networking mobile robot systems [7]. His approach is primarily for defense purposes using a wireless RF networking system, which differs from this project in that we do not have the same constraints of long bandwidth-delay, error prone links, and of being mission critical. Nevertheless, his discussion on communication protocol is still very much applicable. Gage defines protocol as: "a language used by two entities to exchange information over a communications channel – it represents a shared understanding or agreement of how each entity will interpret the signals it receives from the other" [7]. To facilitate communication between the three types of modules in SIE, we must therefore intricately define the signals that will be sent between the modules.

SIE's communications protocol is as follows: once the socket connection has been established, 1-byte messages are sent across the connection, always initiating at the agent in agent-administrator communication, and at the administrator in administrator-device module communication.[‡] This

---

[†]While SHELLEY currently consists of one Sun Ultra 1 Workstation and three Intel-based personal computers, there is no reason why that could not change in the future, and we can maintain the flexibility emphasis in our design by allowing for any number and type of computers.

[‡]SIE employs its own defined types int8 (one byte, unsigned) and int32 (four bytes, unsigned) for all its communication. This is important for portability. If SIE is ported to a platform with a different sized integer (a different number of bits), the only change which needs to occur is the definition of SIE's int8 and int32 [8]. The int8 and int32 type definitions are in the sie_protocol.h file; see **Appendix A**.
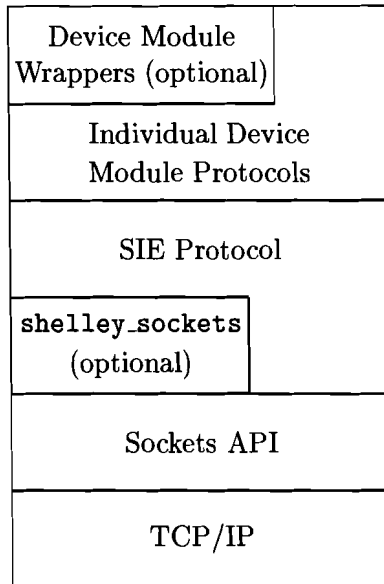
Figure 3: THE LAYERS OF INTER-MODULE COMMUNICATION PROTOCOL USED IN SIE

forces the condition that the administrator cannot directly broadcast information to agents; if something changes at the administrator, such as another agent relinquishing device ownership, agents can only find out this information by explicitly requesting it. Similarly, at the connection between the administrator and a device, state changes at the device can only be known by the administrator if it explicitly queries the device. This may at first appear confining, but it greatly simplifies the protocol between any two devices, because both will always know which is expected to send the next message.

When an agent connects to the administrator, there is a sequence of startup information passed back and forth in the following format: the agent sends the AGENT_CONNECT message[†], to which the administrator then replies with the ADMIN_ACKNOWLEDGE_AGENT_CONNECT message. Next, for each device the agent requires, it sends AGENT_DEVICE_REQUEST followed by an int32 integer which is the device identification number specified in the file device_list. The administrator then responds with ADMIN_CONFIRM_DEVICE_REQUEST if the device was successfully connected, and owner privileges granted to the requesting agent, otherwise, one of the following errors is sent by the administrator: ADMIN_DEVICE_UNKNOWN, ADMIN_DEVICE_ALREADY_OWNED, or ADMIN_DEVICE_NOT_AVAILABLE. In all three error cases the agent has the option of accepting the error (AGENT_ACCEPT) and the stipulations which that implies (see **Section 4.3 The Flow of SIE**) or failing (AGENT_FAIL), in which case there is no further communication between the administrator and the agent; the administrator disconnects the agent and frees all resources used exclusively by that agent. After the administrator has sent the ADMIN_CONFIRM_DEVICE_REQUEST, the administrator also sends a 32-bit integer which is a unique number which the agent should then use whenever referring to the device. The number is also sent after the agent sends the AGENT_ACCEPT message. When all devices have been requested and either confirmed, or errors accepted, the agent sends AGENT_DEVICE_REQUEST_DONE, indicating the end of startup communication between the agent and the administrator.

[†]All messages are declared as constants through C #define statements in the sie_protocol.h header file, see **Appendix A**.

9

When a device is requested, the administrator uses the device id number given by the agent to lookup the device executable (this information is stored in the device list file) and run it, following the convention "<device executable>␣<computer hosting the administrator>␣<port number on which the administrator is listening for devices>." For example:

```
/opt/local/shelley/devices/frame_grabber localhost 4096
```

The administrator then waits for the device to connect to it, sending an acknowledgement upon connection, ADMIN_QUERY_DEVICE, to which the device responds with either DEVICE_READY or DEVICE_FAILED if the device module experienced some internal error and was not able to acquire all its needed resources. On a DEVICE_FAILED, the administrator sends an ADMIN_DISCONNECT_DEVICE message to the device allowing it to exit cleanly, and informs the requesting agent that the device is unavailable.

After these initial exchanges of startup information between agents and the administrator, and devices and the administrator, the specific protocol for a device must be explicitly defined for every device type. The administrator examines the device's .conf file to know which device requests require write permissions (requests not listed in the .conf file are assumed to only require read permission). However, beyond knowledge of what requests can only be issued by the owner, the administrator does not need to know any more specifics of the device protocol, and merely channels allowable requests through from agent to intended device, and from device to appropriate agent.

For an agent to send a command to a device, the agent sends the message AGENT_SEND_DEVICE followed by two int32 numbers: the unique identifying integer to specify the device, and the length (in bytes) of what is being sent to the device. There is the further stipulation that the first byte of the message for the device must be the request code. If the agent is not owner, the administrator compares this request with the requests listed in the .conf file for the device in question, and determines if the message can be sent to the device. If the administrator determines that the message can be sent to the device, the message (stripped of the AGENT_SEND_DEVICE, the int32 device id number, and the int32 message length). The administrator also notes which agent sent the message, so that when the device responds, the message can be channelled to the correct agent. The two requirements we place on device protocol in SIE are: (1) devices must always send some response back after receiving a request, and (2) the device must prepend this response with an int32 indicating the size (in bytes) of the response.

Finally, the agent can send the AGENT_DISCONNECT message which tells the administrator that the agent is quitting. At this point the administrator assumes that it will receive no more communication from the agent, and frees any resources that had been allocated for the agent. If the disconnecting agent is the only agent using any devices, those devices are sent the ADMIN_DISCONNECT_DEVICE message.

5 How SIE Can Support Different Paradigms of Artificial Intelligence

The agent module is defined no further than the communication protocols through which an agent talks to devices in order to sense and react to its surroundings. Thus, flexibility is built into the foundation of the SIE structure, allowing support for any approach to the agent design. The

classical approach to robotics and artificial intelligence is to construct an internal model of the world upon which we have our agents make decisions. All sensory inputs are gathered together and information conflicts are resolved to construct a consistent world view [9]. The advantage to this approach is that all information about the world is centralized and there is one single entity which is fed all the information and can therefore make the most complete and well-informed decision about how to react. This paradigm is very well supported by our integrated environment. The simplest implementation would be to have a single agent that requested from the administrator all the necessary devices to build a world model. The agent could then make calls to all the devices to request data, receive that data, construct a model of the world, and make a decision about how to behave.

Rodney Brooks, of the MIT Artificial Intelligence Laboratory, however, condemns this traditional approach because it is slow (computationally intensive to build a world model) and large (much memory is required to store the internal model). Brooks instead advocates a *subsumption archi-tecture* wherein the decisions are not made by a single agent, but is distributed over an organized hierarchy of behavior modules that directly map perception to action [10]. These separate behavior modules do not directly communicate with each other in making decisions, but rather inhibit other modules when they are active. For example, there may be a behavior that tells a mobile robot to continuously move forward, but there may be another behavior that tells the robot to stop if there is an obstacle in its path. If, in our hierarchy, we defined that the stopping behavior inhibits the forward behavior, our robot will travel forward (the stopping behavior has no reason to be active, and therefore lies dormant and does not inhibit the forward behavior) until the robot encounters an obstacle. When an obstacle is encountered, the stopping behavior is made active, which inhibits the forward behavior – the robot comes to a stop. By defining a hierarchy of behaviors and by defining how these behaviors interact and inhibit each other, we can construct a system that does not need a central intelligence, but can behaved based on a series of smaller intelligences. SIE can easily support this approach to designing intelligent agents by building a separate agent for each "behavior module" and having them access the needed devices through the administrator. Agents – in this case functioning as behavioral modules – can inhibit each other in the way Brooks prescribes using the SHELLEY_SOCKETS mini-library to produce direct inter-agent communication.

6 An Example of SIE Applied: Identifying a User

A simple example of SIE in use is to address the task of recognizing the person sitting at SHEL-LEY's console. The task is accomplished by constructing one agent that accesses two devices: a frame grabber module that upon request returns a frame of video from the Sun video cameras used by SHELLEY, and an artificial neural network module that encapsulates all the functionality and data structures of a neural network, complete with facilities for training using the backpropogation algorithm [11]. This presents an example of a device module which does not interface with any external physical hardware. Instead, the artificial neural network device module provides a com-pletely self-contained functional unit. It is still valid, however, to consider this as a device module because all any device module does is provide some function which should be separate and distinct from the role of the agent.

The user-identifier agent connects to the administrator, and requests both the frame grabber and artificial neural network devices, failing if owner permission cannot be granted for both. After initialization, the agent presents to the current user a menu with options: "identify user," "cap-

ture frames to pgm," "train," or "quit." Through this user interface, we can acquire a series of video frames and save them as pgm image files, use the image files to train the network, and then test the network with live video from one of SHELLEY's Sun video cameras. This functionality demonstrates the use of multiple devices by a single agent, all communicating through the administrator. The specifics of the frame grabber and neural network device protocols can be found in frame_grabber_protocol.h (see **Appendix E**) and neural_net_protocol.h (see **Appendix I**), respectively.

This example demonstrates several key features of SIE. First, SIE's flexibility is exhibited in the implementation styles employed. While the administrator is implemented in strict C (primarily to support multi-threading), the user-identifier agent is implemented in C++. In reality, the devices and agents can be individually implemented in any language with bindings to the sockets API. As long as the communications protocol is followed, one module does not need to know the implementation details of any other module.

Another example of SIE's abstraction of implementation details is the way in which external hardware can be easily changed without disrupting SIE. If the current Sun video cameras were replaced by different peripherals for visual perception, then all that would be needed would be a new frame grabber module that used the same communications protocol for the previously existing agents to still be useful. This system allows agents to not be concerned with the specifics of the neural network itself (see administrator.c in **Appendix B** and agent.C in **Appendix M**).

Finally, the largest single benefit of SIE is the ease of implementation of the user-identifier agent. Simplicity was further increased by the development of C++ wrapper classes to encapsulate the communication with each device; when the agent creates an instance of each wrapper class, the constructor handles the startup communication with the administrator. The agent then calls methods of the classes to send all the requests to the devices (via the administrator) and collect the resulting data (see Frame_Grabber.H in **Appendix G**, Frame_Grabber.C in **Appendix H**, Neural_Net.H in **Appendix K**, and Neural_Net.C in **Appendix L**). These classes are declared and defined in separate files from the user-identifier so that they can be used by other agents. Of course, the implementation of future agents that use these same devices need not employ the wrapper classes, and can rely directly on the protocol header files.

7 Future Work

This research thus far has focused primarily on the design of (SIE), with selective implementation of key points to test theories, verify strategies, and prove concepts. With the network and communication protocols established, as well as the overall flow of SIE well defined, the next step is implementation. The administrator module exists in skeletal form, and while the majority of the central issues are addressed, the multiple agent support and write permission functionality, though well defined, is yet to be implemented.

The following is a partial list of some additional future contributions which could be made to SIE:

- **Network-ability:** Currently, agents can connect to the administrator if it is running on the localhost, or if it is running on any other computer networked to the host of the agent.

12

However, because the administrator must create the device processes, there is currently no means for the devices to be run on a computer other than the one upon which the administrator is being run. Perhaps an investigation of an RPC (remote procedure call) package may provide a solution [6]. An alternative answer could be to not have devices get executed by the administrator when they are needed, but rather run them explicitly and connect them to the administrator at administrator startup, leaving them connected for the entire duration of SIE. In this way, the user could explicitly run the device module from any networked computer, though an obvious disadvantage would be that it would then become a responsibility of the user to ensure that all needed devices were running and connected. Or, perhaps a separate administrator could be run on each computer of a networked cluster, so that when one administrator needed to access resources on another computer, it could be done through inter-administrator cooperation. The advantages to distributing SIE over a networked cluster are many, as are the issues involved which would require addressing.

- **Varying Agent Priorities:** A priority scheme for agent ownership of devices maybe useful for SIE in situations where many agents are being run concurrently. This however, creates the added complexity of communicating to a "less important" agent that a more important agent has come along and ruthlessly usurped device ownership.

- **Midprocess Device Requests:** The present design of SIE forces all device requests by an agent to occur when the agent initially connects to the administrator. A potentially very useful modification could be a design for agents to request devices anytime during their session with the administrator, and not only upon connection.

- **Device "Short-circuiting:"** In the case where an agent essentially streams the data coming in from one device to another device (sending image data from a video frame grabber to an artificial neural network, for example) there would be a performance increase gained if the agent could tell the administrator to channel the data to a specific device rather than send it back to the agent. Very quickly, complications arise when considering this scheme due to the implicit need this creates for differing devices to have compatible communication protocols, which is otherwise not an issue in the current design of SIE.

- **Development of Agents and Devices:** Perhaps the most obvious contribution to be made to SIE is the development of both agents and devices. Ideally, device modules can be created, along with a defined protocol for accessing their functionality, and then saved – building a library of devices which can then be used by agents as they are created. Most likely, device module development will be driven by necessity – when an agent requires access either to a peripheral or some distinct functional unit for which there is no current device module written. It is hoped that the design philosophy of building separate, reusable modules will be followed to maximize code reuse and long-term productivity.

8   Conclusion

The SHELLEY Integrated Environment (SIE) is designed primarily with the goal of easing implementation of future projects by providing an easy means for accessing the devices which allow SHELLEY to interact with her surroundings. The design emphasizes flexibility and expandability, as well as simple code reuse in the form of separate modules. The agent/administrator/device model upon which SIE is built allows the implementors of agents to not be concerned with the inner

workings of accessing specialized hardware – this is localized to specific device modules. Inter-module communication is accomplished using the sockets API, which offers the future opportunity to distribute SIE over a network of computers. Multiple agents can be run in conjunction, building an integrated system of behaviors. It is the administrator's responsibility to regulate and manage agent access to devices, much like in the modern operating system, it is the kernel's responsibility to regulate and manage processes and their access to system resources. Finally, the specifics of how SHELLEY maps sensory input to behavioral output is encapsulated in the agents, thus SIE serves only to facilitate and does not confine how future researchers approach the problem of building an artificially intelligent entity.

Appendix A   sie_protocol.h[†]

```
1     /**********************************************************************
2        Andy Ritger
3        Research Honors
4        4-26-99
5
6      _ The constants which comprise the SIE protocol follow the simple naming
7        convention where the first word is either AGENT, DEVICE, or ADMIN to
8        designate who is sending the message, followed by an underscore separated
9        description of the message.
10
11       Note that all constants are sent across sockets as type int8.
12
13       **********************************************************************/
14
15
16
17    #ifndef ___SIE_PROTOCOL___
18    #define ___SIE_PROTOCOL___
19
20
21
22    /**********************************************************************
23       When porting SIE to other platforms, edit these typedefs as needed so that
24       byte8 is an unsigned 8-bit value, and int32 is an unsigned 32 bit value.
25       **********************************************************************/
26
27    typedef unsigned char int8;
28    typedef unsigned int int32;
29
30
31
32    /**********************************************************************
33       The AGENT_CONNECT message is sent by the agent after a socket connection
34       has been established.  The agent then awaits confirmation from the
35       administrator.
36       **********************************************************************/
37
38    #define AGENT_CONNECT 32
39
40
41
42    /**********************************************************************
```

[†]All of the source code listed in these appendices can be found at www.iwu.edu/~shelley/sie

```c
43      The ADMIN_ACKNOWLEDGE_AGENT_CONNECT is sent by the administrator to an
44      agent after the agent has sent the AGENT_CONNECT message.  This is simply
45      a means of "handshaking" so that one can verify the other's existence.
46      ******************************************************************/
47
48   #define ADMIN_ACKNOWLEDGE_AGENT_CONNECT 33
49
50
51
52   /******************************************************************
53      The AGENT_DEVICE_REQUEST message is sent by the agent to the administrator
54      after the ADMIN_ACKNOWLEDGE_AGENT_CONNECT is received.  The
55      AGENT_DEVICE_REQUEST is followed by an int32 which specifies the id number
56      for a device as given in the file device_list.
57      ******************************************************************/
58
59   #define AGENT_DEVICE_REQUEST 34
60
61
62
63   /******************************************************************
64      The ADMIN_CONFIRM_DEVICE_REQUEST is sent by the administrator to the agent
65      to confirm that the requested device has been verified and the agent granted
66      ownership.  This message is followed by an int32 which is the unique
67      identifying number of the specific instance of the device module, which
68      the agent and administrator will use for all future communication regarding
69      the device module.
70      ******************************************************************/
71
72   #define ADMIN_CONFIRM_DEVICE_REQUEST 35
73
74
75
76   /******************************************************************
77      The ADMIN_DEVICE_UNKNOWN is sent by the administrator to the agent when
78      the requested device id is unknown to the administrator (the given id
79      number is not listed in the device_list configuration file.  This message
80      is followed by an int32 which is the unique identifying number of the
81      specific instance of the device module, which the agent and administrator
82      will use for all future communication regarding the device module.  This
83      number is not really needed, but is given to conform with the conventions
84      followed for other other possible responses made by the administrator
85      regarding device module requests.
86      ******************************************************************/
87
88   #define ADMIN_DEVICE_UNKNOWN 36
89
90
```

```
91
92      /*******************************************************************
93        The ADMIN_DEVICE_ALREADY_OWNED message is sent by the administrator to the
94        agent in response to a device module request if the device module exists,
95        but is already owned by another agent (thus write permission cannot be
96        granted to the requesting agent).  This message is followed by an int32
97        which is the unique identifying number of the specific instance of the
98        device module, which the agent and administrator will use for all future
99        communication regarding the device module.
100       *******************************************************************/
101
102     #define ADMIN_DEVICE_ALREADY_OWNED 37
103
104
105
106     /*******************************************************************
107       The ADMIN_DEVICE_NOT_AVAILABLE message is sent by the administrator to the
108       agent in response to a device module request if the device module is known
109       by the administrator, but it is not available -- either the maximum number
110       of agents are already using it, or there was an error when the administrator
111       attempted to create the device.  This message is followed by an int32 which
112       is the unique identifying number of the specific instance of the device
113       module, which the agent and administrator will use for all future
114       communication regarding the device module.
115       *******************************************************************/
116
117     #define ADMIN_DEVICE_NOT_AVAILABLE 38
118
119
120
121     /*******************************************************************
122       The AGENT_ACCEPT message is sent by the agent to the administrator after
123       one of the above three error messages have been sent (ADMIN_DEVICE_UNKNOWN,
124       ADMIN_DEVICE_ALREADY_OWNED, or ADMIN_DEVICE_NOT_AVAILABLE) to specify
125       that the conditions imposed by the given error will be accepted and the
126       agent wishes to continue.
127       *******************************************************************/
128
129     #define AGENT_ACCEPT 39
130
131
132
133     /*******************************************************************
134       The AGENT_FAIL message is sent by the agent to the administrator after
135       one of the above three error messages have been sent (ADMIN_DEVICE_UNKNOWN,
136       ADMIN_DEVICE_ALREADY_OWNED, or ADMIN_DEVICE_NOT_AVAILABLE) to specify
137       that the conditions imposed by the given error will not be accepted and the
138       agent wishes to fail without proceeding further.  As soon as this message
```

17

```
139    is received, the administrator assumes that the agent is gone, and ignores
140    its existence.
141       ********************************************************************/
142
143   #define AGENT_FAIL 40
144
145
146
147   /********************************************************************
148     The AGENT_DEVICE_REQUEST_DONE message is send by the agent after it has
149     requested all necessary device modules and has dealt with the administrator's
150     responses.
151       ********************************************************************/
152
153   #define AGENT_DEVICE_REQUEST_DONE 41
154
155
156
157   /********************************************************************
158     The ADMIN_QUERY_DEVICE message is sent to the device by the administrator
159     after the device has connected to ensure that the device really is a device.
160       ********************************************************************/
161
162   #define ADMIN_QUERY_DEVICE 42
163
164
165
166   /********************************************************************
167     The DEVICE_READY message is sent by the device to the administrator in
168     response to the administrator's ADMIN_QUERY_DEVICE.  The message indicates
169     that the device is ready to receive commands.
170       ********************************************************************/
171
172   #define DEVICE_READY 43
173
174
175
176   /********************************************************************
177     The DEVICE_FAILED message is sent by the device to the administrator in
178     response to the administrator's ADMIN_QUERY_DEVICE.  The message indicates
179     that the device experienced some internal error and is not able to function.
180     The administrator assumes that the device module goes away after this message
181     is sent
182       ********************************************************************/
183
184   #define DEVICE_FAILED 44
185
186
```

```
187
188    /**********************************************************************
189      The ADMIN_DISCONNECT_DEVICE message is sent by the administrator to a
190      device module to tell the device that it is no longer needed and should
191      exit.
192      **********************************************************************/
193
194    #define ADMIN_DISCONNECT_DEVICE 45
195
196
197
198    /**********************************************************************
199      The AGENT_QUERY_DEVICE_OWNERSHIP message is sent by an agent to the
200      administrator.  The message is immediately followed by an int32 holding
201      the specific identification number of a device module.  This is used
202      by an agent to query the ownership of a device module.
203      **********************************************************************/
204
205    #define AGENT_QUERY_DEVICE_OWNERSHIP 46
206
207
208
209    /**********************************************************************
210      The ADMIN_THIS_AGENT_OWNS_DEVICE is sent by the administrator to an agent
211      is response to the agent's AGENT_QUERY_DEVICE_OWNERSHIP message.  This
212      indicates that the requesting agent is owner of the device module in question
213      (the agent has write permission).
214      **********************************************************************/
215
216    #define ADMIN_THIS_AGENT_OWNS_DEVICE 47
217
218
219
220    /**********************************************************************
221      The ADMIN_ANOTHER_AGENT_OWNS_DEVICE is sent by the administrator to an agent
222      is response to the agent's AGENT_QUERY_DEVICE_OWNERSHIP message.  This
223      indicates that an agent other than the requesting agent is the the owner
224      of the device in question (thus the requesting agent only has read
225      permission).
226      **********************************************************************/
227
228    #define ADMIN_ANOTHER_AGENT_OWNS_DEVICE 48
229
230
231
232    /**********************************************************************
233      The ADMIN_NO_AGENT_OWNS_DEVICE message is sent by the administrator to an
234      agent is response to the agent's AGENT_QUERY_DEVICE_OWNERSHIP message.  This
```

19

```
235    indicates that no agent owns the device in question (no agent has write
236    permission).
237    *****************************************************************/
238
239  #define ADMIN_NO_AGENT_OWNS_DEVICE 49
240
241
242
243  /*****************************************************************
244    The AGENT_REQUEST_DEVICE_OWNERSHIP message is sent by an agent to the
245    administrator. The message is immediately followed by an int32 holding
246    the specific identification number of a device module.  This is used
247    by an agent to request that it be made owner of the device in question.
248    *****************************************************************/
249
250  #define AGENT_REQUEST_DEVICE_OWNERSHIP 50
251
252
253
254  /*****************************************************************
255    The ADMIN_GRANT_DEVICE_OWNERSHIP message is sent the administrator to
256    an agent in response to the agent's AGENT_REQUEST_DEVICE_OWNERSHIP
257    message.  This indicates that the agent now is owner (has write permission)
258    for the device module in question.
259    *****************************************************************/
260
261  #define ADMIN_GRANT_DEVICE_OWNERSHIP 51
262
263
264
265  /*****************************************************************
266    The ADMIN_DENY_DEVICE_OWNERSHIP message is sent by the administrator to
267    an agent in response to the agent's AGENT_REQUEST_DEVICE_OWNERSHIP message.
268    This indicates that the agent does not receive ownership (given write
269    permission) for the device module in question.
270    *****************************************************************/
271
272  #define ADMIN_DENY_DEVICE_OWNERSHIP 52
273
274
275
276  /*****************************************************************
277    The AGENT_SEND_DEVICE message is sent by an agent to the administrator to
278    indicate that what is following is a message for a device.  It is followed
279    by two int32s; the first indicates the id number of the device to which
280    the message should be sent, and the second specifies the length of the
281    message to be sent to the device module.
282    *****************************************************************/
```

```
283
284    #define AGENT_SEND_DEVICE 53
285
286
287
288    /****************************************************************
289       The AGENT_DISCONNECT message is sent by an agent to the administrator to
290       indicate that the agent is done and is exiting.  Upon receiving this message,
291       the administrator releases any resources used exclusively by the
292       disconnecting agent, and hereafter assumes that the agent no longer exists.
293       ****************************************************************/
294
295    #define AGENT_DISCONNECT 54
296
297
298
299    #endif
```

Appendix B administrator.c

```c
1     /**********************************************************************
2       Andy Ritger
3       Research Honors
4       4-26-99
5
6      . administrator.c
7
8       This is the administrator source code.  The administrator is still in *VERY*
9       rudimentary form, but the basic functionality is here.
10
11      **********************************************************************/
12
13
14
15    #include <stdlib.h>
16    #include <stdio.h>
17    #include <string.h>
18    #include <netinet/in.h>
19    #include <netdb.h>
20    #include <unistd.h>
21    #include <fcntl.h>
22    #include <ctype.h>
23    #include <sys/stat.h>
24    #include <sys/wait.h>
25    #include <sys/types.h>
26    #include <sys/socket.h>
27
28    #include "sie_protocol.h"
29
30
31
32    /**********************************************************************
33       structures
34       **********************************************************************/
35
36    typedef struct device_info
37    {
38      device_info *next;   /* pointer so that we can have this in a linked list */
39      int dev_id;          /* the device id as specified in the device_list file */
40      char *path;          /* path and name of binary of device module */
41      char *bin;           /* file name of binary */
42      char *host;          /* host on which to run device module (not implemented)*/
43      int shared;          /* 1 = device can be shared; 0 = not */
```

```
44      int multiple;       /* 1 = device can have multiple instances; 0 = not */
45    };
46
47
48
49    /*********************************************************************
50      prototypes
51        ********************************************************************/
52
53
54
55    /*********************************************************************
56      The function agent_listener () binds a socket, and listens for an agent
57      connecting on that socket.  After this happens, we attempt to acquire
58      all the device modules requested by the connecting agent.  Finally, we
59      sit in an loop and pass device commands from the agent to the appropriate
60      device, and then from the device back to the agent.
61        ********************************************************************/
62
63    void agent_listener (int port_number);
64
65
66
67    /*********************************************************************
68      The connect_device () function is called for each device requested.  A
69      child process is forked off to exec the device module program.  We then
70      listen for the device to connect back to us.
71        ********************************************************************/
72
73    int connect_device (int n);
74
75
76
77    /*********************************************************************
78      The read_device_list () function parses the device_list file, and stores
79      all the relevant information in a linked list.
80        ********************************************************************/
81
82    device_info *read_device_list ();
83
84
85
86    /*********************************************************************
87      The function find_next_data () takes a file handler and moves the handler
88      past any whitespace or lines with pound signs ("#") so that next thing is
89      valid data.
90        ********************************************************************/
91
```

```
92     void find_next_data (FILE *file);
93
94
95
96     /***********************************************************************
97       global variables -- need to be changed
98       **********************************************************************/
99
100    device_info *device_list;
101    int the_socket;
102    sockaddr_in name;
103    int number_of_devices = 0;
104    int device_connections [10];
105
106
107
108    /***********************************************************************
109      main ()
110      **********************************************************************/
111
112    int main (int argc, char **argv)
113    {
114      /* read the device_list file */
115
116      device_list = read_device_list ();
117
118      /* listen for agents trying to connect */
119
120      agent_listener (2048);
121
122      /* tell all connected devices to go away */
123
124      int8 send = ADMIN_DISCONNECT_DEVICE;
125      for (int i = 0; i < number_of_devices; i++)
126        write (device_connections [i], &send, sizeof (int8));
127
128      printf ("ADMINISTRATOR: done.\n");
129      exit (0);
130    } /* main () */
131
132
133
134    /***********************************************************************
135      agent_listener ()
136      **********************************************************************/
137
138    void agent_listener (int port_number)
139    {
```

```
140     int *temp, response, i, n;
141     int ns, player_no;
142     int len, res;
143     char response2, *pname;
144     int8 received, answer;
145
146     /* create the socket */
147     the_socket = socket (AF_INET, SOCK_STREAM, 0);
148
149     /* set the port number */
150     name.sin_family = AF_INET;
151     name.sin_port = htons (port_number);
152     n = INADDR_ANY;
153     memcpy (&name.sin_addr, &n, sizeof (long));
154
155     /* enables reuse of the port number -- this is a very good thing */
156     temp = (int*) malloc (sizeof (int));
157     *temp = 1;
158     setsockopt (the_socket, SOL_SOCKET, SO_REUSEADDR, (char*)temp, sizeof (int));
159
160     /* sets the size of the send and receive buffer -- 64 kb */
161     *temp = 64; /* kilobytes */
162     setsockopt (the_socket, SOL_SOCKET, SO_SNDBUF, (char*) temp, sizeof (int));
163     setsockopt (the_socket, SOL_SOCKET, SO_RCVBUF, (char*) temp, sizeof (int));
164
165     /* attempts to bind the socket -- failure returns -1 */
166     res = bind (the_socket, (struct sockaddr*) (&name), sizeof (sockaddr_in));
167     if (res == -1)
168     {
169       printf ("ADMINISTRATOR ERROR: unable to bind socket to port number %d.\n",
170         port_number);
171       return;
172     }
173
174     /* listen to the socket, waiting for agents to connect... */
175     listen (the_socket, 5);
176     len = sizeof (sockaddr_in);
177
178     ns = accept (the_socket, (struct sockaddr*) (&name), &len);
179
180     /* we've received a connection, check to see if it's an agent */
181     read (ns, &received, sizeof (int8));
182
183     if (received == AGENT_CONNECT)
184     {
185       printf ("ADMINISTRATOR: an agent has connected...\n");
186       answer = ADMIN_ACKNOWLEDGE_AGENT_CONNECT;
187       write (ns, &answer, sizeof (int8));
```

```
188    }
189    else
190    {
191      printf ("ADMINISTRATOR: something has connected on the agent port,\n");
192      printf ("                but it didn't identify itself as an agent.\n");
193      printf ("                Proceeding, but problems may arise.\n");
194    }
195
196    /* this will either be a device request, or a device done... */
197    read (ns, &received, sizeof (int8));
198
199    int32 val;
200
201    while (received != AGENT_DEVICE_REQUEST_DONE)
202    {
203      printf ("ADMINISTRATOR: handling device request...\n");
204
205      if (received == AGENT_DEVICE_REQUEST)
206      {
207        read (ns, &val, sizeof (int32));
208        device_connections [number_of_devices] = connect_device (val);
209        number_of_devices++;
210      }
211
212      else
213      {
214        printf ("ADMINISTRATOR: agent device request protocol not followed\n");
215        printf ("                by connecting agent. Proceeding, but\n");
216        printf ("                problems may arise.\n");
217      }
218      /* get next command (either a device request or a device done) */
219      read (ns, &received, sizeof (int8));
220    }
221
222    printf ("ADMINISTRATOR: device module setup complete\n");
223
224    int32 device_id, length;
225    char* message;
226
227    /* block on a read until the agent tells us to do something */
228    read (ns, &received, sizeof (int8));
229    while (received != AGENT_DISCONNECT)
230    {
231      if (received == AGENT_SEND_DEVICE)
232      {
233        /* which device? */
234        read (ns, &device_id, sizeof (int32));
235
```

```
236          /* how long is the message? */
237          read (ns, &length, sizeof (int32));
238
239          /* this would be where we would examine the request, and verify ownership
240      (if necessary) */
241
242          message = (char*) malloc (length);
243
244          /* just past the message through */
245          read (ns, message, length);
246          write (device_connections [device_id-1], message, length);
247          free (message);
248
249          /* listen on the device socket and pass what we get back to the agent */
250
251          /* length of the message */
252          read (device_connections [device_id-1], &length, sizeof (int32));
253          message = (char*) malloc (length);
254
255          /* the message */
256          read (device_connections [device_id-1], message, length);
257          write (ns, message, length);
258          free (message);
259        }
260        /* block on a read until the agent tells to do something, again */
261        read (ns, &received, sizeof (int8));
262      }
263    } /* agent_listener () */
264
265
266
267    /***********************************************************************
268       connect_device ()
269       ***********************************************************************/
270
271    int connect_device (int n)
272    {
273      device_info *current_node;
274      current_node = device_list;
275
276      char *bin, *path;
277
278      /* look for device id n in the linked list */
279      while (current_node)
280      {
281        if (current_node->dev_id == n)
282        {
283          bin = current_node->bin;
```

27

```
284         path = current_node->path;
285       }
286     current_node = current_node->next;
287     }
288
289     /* we assume that the dev_id is in the list... needs error trapping */
290
291     /* fork off a process to execute the device program */
292     pid_t childpid;
293     if ((childpid = fork ()) == 0)
294     {
295       if (execl (path, bin, NULL) < 0)
296       {
297         printf ("ADMINISTRATOR: unable to execute %s\n", bin);
298         return (-1);
299       }
300       exit (0);
301     }
302
303     /* set up the socket to listen...*/
304
305     listen (the_socket, 5);
306     int len = sizeof (sockaddr_in);
307
308     int return_val = accept (the_socket, (struct sockaddr*) (&name), &len);
309     int8 val = ADMIN_QUERY_DEVICE;
310     write (return_val, &val, sizeof (int8));
311
312     /* listen for a response */
313     read (return_val, &val, sizeof (int8));
314     if (val != DEVICE_READY)
315       printf ("ADMINISTRATOR: the device is not behaving as expected...\n");
316
317     return (return_val);
318  } /* connect_device () */
319
320
321
322
323
324  device_info *read_device_list ()
325  {
326    int dev, len, max_share, max_mult;
327    char bin[100], host[100], share[100], mult[100];
328
329    /* open the device list file -- it must be in our directory */
330    FILE *devfile = fopen ("device_list", "r");
331    if (devfile == NULL)
```

```
332      {
333        printf ("ADMINISTRATOR ERROR: unable to open device_list\n");
334        exit (0);
335      }
336
337      find_next_data (devfile); /* skip the comments and find data */
338
339      /* create the first node in our linked list */
340      device_info *dev_list = NULL;
341      device_info *current_node = NULL;
342      device_info *last_node = NULL;
343
344      dev_list = (device_info *) malloc (sizeof (dev_list)+100);
345      current_node = dev_list;
346
347      while ((feof(devfile)) == 0)
348      {
349        /* allocate memory for the next one in line */
350        current_node->next = (device_info *) malloc (sizeof (dev_list) + 100);
351
352        /* read the data from file */
353        fscanf (devfile, "%d %s %s %s %s\n", &dev, bin, host, share, mult);
354
355        /*    printf ("%d %s %s %s %s\n", dev, bin, host, share, mult);*/
356
357        /* copy the device id */
358        current_node->dev_id = dev;
359
360        /* copy the binary path and name */
361        len = strlen (bin);
362        current_node->path = (char*) malloc (len+1);
363        strncpy (current_node->path, bin, len);
364        current_node->path [len] = '\0';
365
366        /* walk backwards and get the binary itself...*/
367        char ch = '\0';
368        int temp_len = len;
369        while (ch != '/')
370        {
371          temp_len--;
372          ch = bin [temp_len];
373        }
374
375        temp_len++;
376        current_node->bin = (char*) malloc (len - temp_len);
377
378        for (int i = 0; i < len-temp_len; i++)
379          current_node->bin [i] = bin [temp_len + i];
```

```
380
381        current_node->bin [len-temp_len] = '\0';
382
383        /* copy the host name */
384        len = strlen (host);
385        current_node->host = (char*) malloc (len+1);
386        strncpy (current_node->host, host, len);
387        current_node->host [len] = '\0';
388
389        /* interpret the share */
390        if (strcmp ("no-share", share) == 0)
391          current_node->shared = 0;
392        else if (strcmp ("share", share) == 0)
393          current_node->shared = 1;
394        else
395        {
396          printf ("ADMINISTRATOR WARNING: cannot understand \'%s\' for ", share);
397          printf ("device %d\n", dev);
398          current_node->shared = 0;
399        }
400
401        /* interpret the multiple */
402        if (strcmp ("no-multiple", mult) == 0)
403          current_node->multiple = 0;
404        else if (strcmp ("multiple", mult) == 0)
405          current_node->multiple = 1;
406        else
407        {
408          printf ("ADMINISTRATOR WARNING: cannot understand \'%s\' for ", mult);
409          printf ("device %d\n", dev);
410          current_node->multiple = 0;
411        }
412
413        /* maintain the linked list */
414        last_node = current_node;
415        current_node = current_node->next;
416
417        /* skip the comments and whitespace -- just find data */
418        find_next_data (devfile);
419      }
420      fclose (devfile);
421
422      free (last_node->next);
423      last_node->next = NULL;
424
425      return (dev_list);
426    } /* read_device_list () */
427
```

```
428
429
430  /***********************************************************************
431     find_next_data ()
432     **********************************************************************/
433
434  void find_next_data (FILE *file)
435  {
436    /* if the next character is a '#' -- the line is a comment and should be
437       discarded... if the next character is whitespace, it should likewise be
438       removed */
439
440    char c;
441    int loop = 1;
442
443    char garbage [100];
444
445    while (loop)
446    {
447      c = fgetc (file);
448
449      /* if we have a pound sign, then the rest of that line is comment;
450         keep reading until we find a \n */
451
452      if (c == '#')
453        while (c != '\n') c = fgetc (file);
454
455      /* if it's a digit... then we have data */
456      else if (isdigit (c))
457      {
458        ungetc (c, file);
459        loop = 0;
460      }
461
462      /* if we hit the end of the file... break the loop */
463      else if (c == EOF)
464        loop = 0;
465    }
466  } /* find_next_data () */
```

Appendix C   shelley_sockets.h

```
1    /********************************************************************
2       Andy Ritger
3       Research Honors
4       4-26-99                                    .
5
6     . shelley_sockets.h
7
8       The shelley_sockets are a small collection of functions to create a socket
9       connection between two independent processes either existing on the same
10      computer, or distributed over a network.
11
12      ********************************************************************/
13
14
15
16   #ifndef ___SHELLEY_SOCKETS___
17   #define ___SHELLEY_SOCKETS___
18
19
20
21   #include <stdlib.h>
22   #include <stdio.h>
23   #include <string.h>
24   #include <sys/types.h>
25   #include <sys/socket.h>
26   #include <netinet/in.h>
27   #include <netdb.h>
28   #include <unistd.h>
29
30
31
32   /********************************************************************
33      This function should be called when we want to have our server listen for
34      connections on a specific port number.  The return value is the socket
35      id, or -1 if the function fails.
36      ********************************************************************/
37
38   int shelley_sockets_server_listen_for_client (int port_number);
39
40
41
42   /********************************************************************
43      This function should be called when we want to have our client connect to
```

```
44      an existing server.  We pass in the port on which to connect, and the
45      name of the machine on which the server is being run -- set this to
46      "localhost" if the server is on the same machine as the client.  Returns
47      -1 if it fails to connect.
48      *****************************************************************/
49
50   int shelley_sockets_client_connect_to_server (int port_number, char* hostname);
51
52
53
54   /*****************************************************************
55      Read from the socket; this blocks until there is length_to_read bytes
56      to read at the socket.  When this returns, what was read is pointed to by
57      the data pointer.
58      *****************************************************************/
59
60   int shelley_sockets_read (int the_socket, char *data, int length_to_read);
61
62
63
64   /*****************************************************************
65      Write to the socket.  What should be written should be pointed to by the
66      data pointer, and be length_to_write bytes long.
67      *****************************************************************/
68
69   int shelley_sockets_write (int the_socket, char *data, int length_to_write);
70
71
72
73   #endif
```

Appendix D shelley_sockets.c

```
1     /*****************************************************************
2        Andy Ritger
3        Research Honors
4        4-26-99
5
6      . shelley_sockets.c
7
8        ***************************************************************/
9
10
11
12    #include "shelley_sockets.h"
13
14
15
16    int shelley_sockets_server_listen_for_client (int port_number)
17    {
18      int addr = port_number; /* won't need this */
19      int *temp, response, i, n;
20      int ns, player_no, the_socket;
21      int len, res;
22      char response2, *pname;
23      sockaddr_in name;
24
25      /*********************************************************
26        see the man page on socket (SunOS 5.5, socket(3N)):
27
28        socket() creates an endpoint for communication and returns a descriptor.
29
30        ...The domain parameter specifies a communications domain within which
31        communication will take place; this selects the protocol family which
32        should be used.  The protocol family generally is the same as the address
33        family for the addresses supplied in later operations on the socket.
34        These families are defined in the include file <sys/socket.h>...
35
36        ...A SOCK_STREAM type provides sequenced, reliable, two-way connection-
37        based byte streams... Sockets of type SOCK_STREAM are full-duplex byte
38        streams, similar to pipes.  A stream socket must be in a connected
39        state before any data may be sent or received on it.  A connection to
40        another socket is created with a connect(3N) call.
41        *********************************************************/
42
43      the_socket = socket (AF_INET, SOCK_STREAM, 0);
```

```
44
45      /* set the port number */
46      name.sin_family = AF_INET;
47      name.sin_port = htons (port_number);
48      n = INADDR_ANY;
49      memcpy (&name.sin_addr, &n, sizeof (long));
50
51      /* enables reuse of the port number -- this is a very good thing */
52      temp = (int*) malloc (sizeof (int));
53      *temp = 1;
54      setsockopt (the_socket, SOL_SOCKET, SO_REUSEADDR, (char*)temp, sizeof(int));
55
56      /* sets the size of the send and receive buffer -- 64 kb */
57      *temp = 64; /* kilobytes */
58      setsockopt (the_socket, SOL_SOCKET, SO_SNDBUF, (char *)temp, sizeof(int));
59      setsockopt (the_socket, SOL_SOCKET, SO_RCVBUF, (char *)temp, sizeof(int));
60
61      /* attempts to bind the socket -- failure returns -1 */
62      res = bind (the_socket, (struct sockaddr*) (&name), sizeof (sockaddr_in));
63      if (res == -1)
64      {
65        printf ("SHELLEY SOCKET ERROR: unable to bind socket\n");
66        return (-1);
67      }
68
69      /* listen to the socket, waiting for a client to connect... if no client
70          connects, then the we will listen forever, perhaps some timing mechanism
71          should be implemented */
72
73      listen (the_socket, 5);
74      len = sizeof (sockaddr_in);
75      ns = accept (the_socket, (struct sockaddr*) (&name), &len);
76
77      /* if we get to this point, then a connection has been made */
78
79      return (ns);
80
81  } /* shelley_sockets_server_listen_for_client () */
82
83
84
85
86  int shelley_sockets_client_connect_to_server (int port_number, char* hostname)
87  {
88    int *temp, response, i, n, ns, player_no, the_socket, len, res;
89    struct hostent *hp;
90    struct sockaddr_in name;
91    char buffer [50];
```

35

```
92
93      the_socket = socket (AF_INET, SOCK_STREAM, 0);
94
95      /* enable reuse of port number */
96      temp = (int *) malloc (sizeof (int));
97      *temp = 1;
98      setsockopt (the_socket, SOL_SOCKET, SO_REUSEADDR, (char*) temp, sizeof(int));
99
100     /* set the size of the send and receive buffer */
101     *temp = 64; /* assuming kilobytes - if bytes must set to 65536 */
102     setsockopt (the_socket, SOL_SOCKET, SO_SNDBUF, (char*)temp, sizeof (int));
103     setsockopt (the_socket, SOL_SOCKET, SO_RCVBUF, (char*)temp, sizeof (int));
104
105     memset (&name, 0, sizeof (struct sockaddr_in));
106     name.sin_family = AF_INET;
107
108     name.sin_port = htons (port_number);
109     hp = gethostbyname (hostname);
110
111     memcpy (&name.sin_addr, hp->h_addr_list[0], hp->h_length);
112     len = sizeof (struct sockaddr_in);
113
114     /* connect to server */
115     if ((connect (the_socket, (struct sockaddr *) &name, len)) == -1)
116     {
117       the_socket = -1;
118       printf ("SHELLEY SOCKET ERROR: client unable to connect to server\n");
119     }
120
121     return (the_socket);
122   } /* shelley_sockets_client_connect_to_server () */
123
124
125
126   int shelley_sockets_read (int the_socket, char *data, int length_to_read)
127   {
128     int amount_to_read;
129     int amount_read;
130     int return_val = 0;;
131     char *pointer;
132
133     pointer = data;
134
135     amount_to_read = length_to_read;
136
137     /* loop as long as it takes to read all the data */
138     while (amount_to_read > 0)
139     {
```

```
140        amount_read = read (the_socket, pointer, amount_to_read);
141        if ((amount_read == EOF) || (amount_read == 0))
142        {
143          printf ("SHELLEY SOCKETS: unable to read from socket\n");
144          amount_to_read = 0;
145          return_val = -1;
146        }
147        amount_to_read -= amount_read;
148        pointer += amount_read;
149      }
150
151      return (return_val);
152    } /* shelley_sockets_read () */
153
154
155
156    int shelley_sockets_write (int the_socket, char *data, int length_to_write)
157    {
158
159      int amount_to_write = 0;
160      int amount_written = 0;
161      char *pointer = data;
162
163      amount_to_write = length_to_write;
164
165      /* loop as long as it takes to write all the data */
166      while (amount_to_write > 0)
167      {
168        amount_written = write (the_socket, pointer, amount_to_write);
169        amount_to_write -= amount_written;
170        pointer += amount_written;
171      }
172
173      return (0);
174
175    } /* shelley_sockets_write () */
```

Appendix E frame_grabber_protocol.h

```
 1    /**********************************************************************
 2       Andy Ritger
 3       Research Honors
 4       4-26-99
 5
 6     . The Frame Grabber Device Module Protocol
 7
 8       Note that all constants are sent across sockets as type int8.
 9
10       **********************************************************************/
11
12
13
14    #ifndef ___SIE_FRAME_GRABBER_PROTOCOL___
15    #define ___SIE_FRAME_GRABBER_PROTOCOL___
16
17
18
19    #define FRAME_GRAB_INIT_VALUE 0
20
21
22
23    /**********************************************************************
24       The FRAME_GRAB_REQUEST_RESOLUTION message asks the frame grabber device
25       module what resolution it is returning frames at.  This returns two int32s:
26       the width and height (in pixels).
27       **********************************************************************/
28
29    #define FRAME_GRAB_REQUEST_RESOLUTION 1
30
31
32
33    /**********************************************************************
34       The FRAME_GRAB_SET_RESOLUTION message is followed by a float which
35       corresponds to the scale factor which will be multipled by the original
36       dimensions of the image.  The FRAME_GRAB_YES message is returned upon
37       success.
38       **********************************************************************/
39
40    #define FRAME_GRAB_SET_RESOLUTION 2
41
42
43
```

```
44    /*************************************************************************
45      The FRAME_GRAB_QUERY_GREYSCALE_SUPPORT message queries the device module
46      if it can support grabbing frames in greyscale.  One of the messages
47      FRAME_GRAB_NO or FRAME_GRAB_YES are returned in response.
48      ************************************************************************/
49
50    #define FRAME_GRAB_QUERY_GREYSCALE_SUPPORT 3
51
52
53
54    /*************************************************************************
55      The FRAME_GRAB_QUERY_RGB_SUPPORT message queries the device module if it can
56      support grabbing frames in rgb color encoding.  One of the messages
57      FRAME_GRAB_NO or FRAME_GRAB_YES are returned in response.
58      ************************************************************************/
59
60    #define FRAME_GRAB_QUERY_RGB_SUPPORT 4
61
62
63
64    /*************************************************************************
65      The FRAME_GRAB_NO and FRAME_GRAB_YES message are used as respones to
66      requests to the device module.
67      ************************************************************************/
68
69    #define FRAME_GRAB_NO 5
70    #define FRAME_GRAB_YES 6
71
72
73
74    /*************************************************************************
75      The FRAME_GRAB_SELECT_INPUT_PORT message is followed by an int32 which
76      specifies the port number the device should use to receive video.
77      ************************************************************************/
78
79    #define FRAME_GRAB_SELECT_INPUT_PORT 7
80
81
82
83    /*************************************************************************
84      The FRAME_GRAB_GRAB_FRAME message tells the device module to flush the video
85      buffer and grab the current frame of video.  Returned is a row major stream
86      of bytes, where each byte is a grey-scale pixel value.
87      ************************************************************************/
88
89    #define FRAME_GRAB_GRAB_FRAME 8
90
91
```

```
 92
 93    /*********************************************************************
 94      The FRAME_GRAB_ENABLE_DISPLAY message tells the video grabber to show the
 95      grabbed frame to an X-Window.  Returns FRAME_GRAB_YES.
 96      ********************************************************************/
 97
 98    #define FRAME_GRAB_ENABLE_DISPLAY 9
 99
100
101
102    /*********************************************************************
103      The FRAME_GRAB_DISABLE_DISPLAY message tells the video grabber to not show
104      the grabbed frame to an X-Window.  Returns FRAME_GRAB_YES.
105      ********************************************************************/
106
107    #define FRAME_GRAB_DISABLE_DISPLAY 10
108
109
110    #endif
111
```

Appendix F frame_grabber.c

```
 1    /***********************************************************************
 2       Andy Ritger
 3       Research Honors
 4       4-26-99
 5
 6     _ frame_grabber.c
 7
 8       Interface to sun video cameras; modified from original sample program
 9       included with hardware...
10
11       ***********************************************************************/
12
13
14
15    #include <stdio.h>
16    #include <stdlib.h>
17    #include <xil/xil.h>
18    #include <signal.h>
19
20    #include "sie_protocol.h"
21    #include "shelley_sockets.h"
22    #include "frame_grabber_protocol.h"
23
24
25
26    /* needed by init_cmap () */
27
28    #define CMAPSIZE      256
29    #define TOP2   50 /* reserve the top two entries of the
30      * colormap to reduce colormap flashing */
31
32
33
34    /* function prototypes */
35    void close_cleanly (int sig);
36    void rip_frame (XilImage img, unsigned char *data, int w, int h);
37    void init_cmap (XilLookup xil_cmap, Display * display, Window window,
38      int offset);
39
40
41    /* global variables */
42    XilSystemState _xil_state;
43    Display *xdisplay;
```

```
44
45
46
47     /*************************************************************************
48        main ()
49        *************************************************************************/
50
51     main (int argc, char **argv)
52     {
53        XilDevice device;
54        XilImage rtvc_image, rtvc_luma, rtvc_scaled;
55        XilDataType datatype;
56
57        /* fun with Xlib */
58        Window xwindow;
59        XEvent event;
60        int display_depth;
61
62        int32 width, height, original_width, original_height, nbands;
63
64        char *devname = "/dev/rtvc0";
65        int max_buffers = 0;
66        float scale_factor = 1.0;
67        int32 port_number = 1;
68        int window_shown = 0;
69        int display_enabled = 0;
70
71        unsigned char *data = NULL;
72
73        /************************
74           open the xil library
75           ***********************/
76
77        _xil_state = xil_open ();
78        if (_xil_state == NULL)
79        {
80           fprintf (stderr, "unable to open xil library\n");
81           exit (1);
82        }
83
84        /* catch "^C" so that we can close things cleanly */
85
86        signal (SIGINT, close_cleanly);
87
88        /* create a device so that we can set its attributes */
89
90        if (!(device = xil_device_create (_xil_state, "SUNWrtvc")))
91        {
```

```
92          fprintf (stderr, "Unable to create a device object\n");
93          xil_close (_xil_state);
94          exit(1);
95       }
96
97       xil_device_set_value (device, "DEVICE_NAME", (void *) devname);
98       xil_device_set_value (device, "MAX_BUFFERS", (void *) max_buffers);
99       xil_device_set_value (device, "PORT_V", (void *) port_number);
100
101      /* create an xil image with the above defined device values */
102
103      if (!(rtvc_image = xil_create_from_device (_xil_state, "SUNWrtvc", device)))
104      {
105        fprintf (stderr, "failed to open SUNWrtvc device\n");
106        xil_close (_xil_state);
107        exit (1);
108      }
109
110      /* release the xil device */
111
112      xil_device_destroy (device);
113
114      /* get all the information about the image */
115
116      xil_get_info (rtvc_image, &original_width, &original_height,
117      &nbands, &datatype);
118
119      width = (int32) (original_width * scale_factor);
120      height = (int32) (original_height * scale_factor);
121
122      /* create a copy of the image that will have just the 1st band */
123
124      rtvc_luma = xil_create_child (rtvc_image, 0, 0, original_width,
125      original_height, 0, 1);
126
127      /* create a scaled image to put our scaled copies */
128
129      rtvc_scaled = xil_create (_xil_state, width, height, 1, datatype);
130
131      /* setup the Xwindow and other fun things... */
132
133      /* xlib window creation */
134
135      xdisplay = XOpenDisplay (NULL);
136
137      if (!xdisplay)
138      {
139        fprintf (stderr, "Unable to connect to X-server\n");
```

```
140        xil_close (_xil_state);
141        exit (1);
142    }
143
144    display_depth = DefaultDepth (xdisplay, DefaultScreen (xdisplay));
145    xwindow = XCreateSimpleWindow (xdisplay, DefaultRootWindow (xdisplay),
146    0, 0, width, height, 0, 0, 0);
147    if (!xwindow) {
148        fprintf (stderr, "Unable to create X-window\n");
149        xil_close (_xil_state);
150        exit (1);
151    }
152
153    /* we'll only worry about the expose event */
154
155    XSelectInput (xdisplay, xwindow, ExposureMask);
156
157    /* We're operating at 8 bit display depth */
158
159    XilLookup grayramp;
160    int num_entries = 256;
161
162    Xil_unsigned8 *graydata = (Xil_unsigned8 *) malloc (3 * num_entries);
163    for (int i = 0; i < num_entries; i++)
164        graydata [i * 3 + 2] = graydata [i * 3 + 1] = graydata [i * 3] = i;
165
166    grayramp = xil_lookup_create (_xil_state, XIL_BYTE, XIL_BYTE,
167    3, num_entries, 0, graydata);
168
169    /* connect to administrator on localhost at 2048 */
170
171    int sock = shelley_sockets_client_connect_to_server (2048, "localhost");
172    if (sock == -1)
173    {
174        fprintf (stderr, "failed to connect to server\n");
175        xil_close (_xil_state);
176        exit (1);
177    }
178
179    /* hand shake with administrator */
180
181    int8 my_val;
182    shelley_sockets_read (sock, (char*) &my_val, sizeof (int8));
183    if (my_val != ADMIN_QUERY_DEVICE)
184    {
185        fprintf (stderr, "I am confused\n");
186        exit (1);
187    }
```

```
188    my_val = DEVICE_READY;
189    shelley_sockets_write (sock, (char*) &my_val, sizeof (int8));
190
191    /* start the request loop */
192
193    int32 return_val;
194    int8 command = FRAME_GRAB_INIT_VALUE;
195    int val;
196    while (command != ADMIN_DISCONNECT_DEVICE)
197    {
198       /* block until we get the next command */
199
200       if ((shelley_sockets_read (sock, (char*) &command, sizeof (int8))) == -1)
201          close_cleanly (0);
202       else
203       {
204          switch (command)
205          {
206          case FRAME_GRAB_REQUEST_RESOLUTION:
207    /* return the resolution we're using... */
208    return_val = sizeof (int32) * 2;
209    shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
210    shelley_sockets_write (sock, (char*) &width, sizeof (int32));
211    shelley_sockets_write (sock, (char*) &height, sizeof (int32));
212    break;
213
214          case FRAME_GRAB_SET_RESOLUTION:
215    /* reset the scale factor */
216    shelley_sockets_read (sock, (char*) &scale_factor, sizeof (float));
217
218    /* need to error trap scale factor values */
219    width = (int) (original_width * scale_factor);
220    height = (int) (original_height * scale_factor);
221
222    /* destroy the current scaled image */
223    xil_destroy (rtvc_scaled);
224
225    /* create a new scaled image, and xwindow */
226    XResizeWindow (xdisplay, xwindow, width, height);
227    if ((display_enabled) && (window_shown))
228       rtvc_scaled = xil_create_from_window (_xil_state, xdisplay, xwindow);
229    else
230       rtvc_scaled = xil_create (_xil_state, width, height, 1, datatype);
231
232    return_val = 1;
233    my_val = FRAME_GRAB_YES;
234    shelley_sockets_write (sock, (char*) &return_val, 4);
235    shelley_sockets_write (sock, (char*) &my_val, 1);
```

```
236
237    break;
238
239          case FRAME_GRAB_QUERY_GREYSCALE_SUPPORT:
240    /* yes, we do support greyscale */
241    my_val = FRAME_GRAB_YES;
242    return_val = sizeof (int8);
243    shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
244    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
245    break;
246
247          case FRAME_GRAB_QUERY_RGB_SUPPORT:
248    /* no, we do not support rgb (yet?...) */
249    my_val = FRAME_GRAB_NO;
250    return_val = sizeof (int8);
251    shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
252    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
253    break;
254
255          case FRAME_GRAB_SELECT_INPUT_PORT:
256    /* select which video port (1 or 2)... needs to be error trapped */
257    shelley_sockets_read (sock, (char*) &port_number, sizeof (int32));
258    xil_set_device_attribute (rtvc_image, "PORT_V", (void *) port_number);
259    break;
260
261          case FRAME_GRAB_ENABLE_DISPLAY:
262    /* enable the X window */
263    display_enabled = 1;
264    return_val = 1;
265    my_val = FRAME_GRAB_YES;
266    shelley_sockets_write (sock, (char*) &return_val, 4);
267    shelley_sockets_write (sock, (char*) &my_val, 1);
268    break;
269
270          case FRAME_GRAB_DISABLE_DISPLAY:
271    /* disable the X window */
272    display_enabled = 0;
273    if (window_shown)
274    {
275      /* these two lines give focus to the xwindow, and then take the
276         focus away; this is so the window manager colors are released
277         when the xwindow is unmapped. */
278      XSetInputFocus (xdisplay, xwindow, RevertToNone, CurrentTime);
279      XSetInputFocus (xdisplay, PointerRoot, RevertToNone, CurrentTime);
280
281      /* unmap the xwindow (hide it) */
282      XUnmapWindow (xdisplay, xwindow);
283
```

```
284        /* force any updates which need to happen */
285        XFlush (xdisplay);
286        window_shown = 0;
287      }
288
289      /* destroy and recreate the scaled image so that it is not connected
290          to the xwindow */
291      xil_destroy (rtvc_scaled);
292      rtvc_scaled = xil_create (_xil_state, width, height, 1, datatype);
293
294      return_val = 1;
295      my_val = FRAME_GRAB_YES;
296      shelley_sockets_write (sock, (char*) &return_val, 4);
297      shelley_sockets_write (sock, (char*) &my_val, 1);
298      break;
299
300          case FRAME_GRAB_GRAB_FRAME:
301      /* flush, and grab a current frame of video */
302      if ((window_shown == 0) && (display_enabled))
303      {
304        window_shown = 1;
305        XMapWindow (xdisplay, xwindow); /* make the window visible */
306        do    /* wait for the window to be mapped (an Expose event) */
307          XNextEvent (xdisplay, &event);
308        while (event.xany.type != Expose);
309        xil_destroy (rtvc_scaled);
310        rtvc_scaled = xil_create_from_window (_xil_state, xdisplay, xwindow);
311        init_cmap (grayramp, xdisplay, xwindow, 0);
312      }
313
314      /* flush */
315      xil_set_device_attribute (rtvc_image, "FLUSH_BUFFERS", NULL);
316
317      /* if display is connected to an xwindow, the scale draws it to
318          screen; otherwise, this just makes an internal scaled copy which
319          we need so that we can grab the data */
320
321      xil_scale (rtvc_luma, rtvc_scaled, "nearest", scale_factor,
322          scale_factor);
323      if (data == NULL)
324        data = (unsigned char *) malloc (width * height);
325      rip_frame (rtvc_scaled, data, width, height);
326      return_val = width*height;
327      shelley_sockets_write (sock, (char*) &return_val, sizeof (int32));
328      shelley_sockets_write (sock, (char *) data, (width * height));
329      break;
330
331          case ADMIN_DISCONNECT_DEVICE:
```

47

```
332      /* we're supposed to quit, now */
333      close_cleanly (0);
334      break;
335
336          } /* switch statement */
337
338        } /* if the read succeeded */
339
340      } /* while */
341      return 0;
342
343    } /* end main () */
344
345
346
347    /***********************************************************************
348      close_cleanly ()
349      ***********************************************************************/
350
351    void close_cleanly (int sig)
352    {
353      if (xdisplay) XCloseDisplay (xdisplay);
354      xil_close (_xil_state);
355      exit (0);
356    } /* end close_cleanly () */
357
358
359
360    /***********************************************************************
361      rip_frame ()
362      ***********************************************************************/
363
364    void rip_frame (XilImage img, unsigned char *data, int w, int h)
365    {
366      /* allocate pixel values buffer */
367      float *pixel_vals = (float *) malloc (3 * sizeof (float));
368
369      for (int y = 0; y < h; y++)
370        for (int x = 0; x < w; x++)
371        {
372          /* get a specific pixel from the image */
373          xil_get_pixel (img, x, y, pixel_vals);
374
375          /* copy the data into our data stream to be sent to an agent */
376          data [(y * w) + x] = (unsigned char) pixel_vals [0];
377        }
378    } /* end rip_frame () */
379
```

```
380
381
382     /**********************************************************************
383        init_cmap()
384
385        Initialize the X colormap with a 3-banded XilLookup, using the 'offset'
386        argument to determine the starting pixel value in the X colormap. If the
387        'offset' argument is negative, automatically calculate the starting pixel
388        value in such a way as to minimize colormap flashing.
389
390        Finally, adjust the offset of the XilLookup accordingly.
391        **********************************************************************/
392
393     void init_cmap (XilLookup xil_cmap, Display * display, Window window,
394        int offset)
395     {
396        unsigned long junk[CMAPSIZE], pixels[CMAPSIZE], mask;
397        XColor cdefs[CMAPSIZE];
398        Colormap rcmap;
399        int cmapsize;
400        int i;
401        Xil_unsigned8 cmap_data[CMAPSIZE * 3];
402        Xil_unsigned8 *ptr;
403
404        rcmap = XCreateColormap(display, window,
405          DefaultVisual(display, DefaultScreen(display)),
406          AllocNone);
407
408        cmapsize = xil_lookup_get_num_entries(xil_cmap);
409
410        /* determine the offset for the colormap */
411        if (offset < 0) {
412          offset = 256 - cmapsize - TOP2;
413          if (offset < 0)
414            offset = 0; /* in case cmapsize >= 255 */
415        }
416
417        if (offset) {
418          if (!XAllocColorCells(display, rcmap, 0, &mask, 0, junk, offset)) {
419            fprintf (stderr, "XAlloc1 failed\n");
420          }
421        }
422
423        if (!XAllocColorCells(display, rcmap, 0, &mask, 0, pixels, cmapsize)) {
424          fprintf (stderr, "XAlloc2 failed\n");
425        }
426
427        /* free the unused colors in the front */
```

```
428     if (offset) {
429         XFreeColors (display, rcmap, junk, offset, 0);
430     }
431
432     for (i = 0; i < cmapsize; i++) {
433         cdefs[i].pixel = i + offset;
434     }
435
436     xil_lookup_get_values(xil_cmap, xil_lookup_get_offset(xil_cmap),
437     cmapsize, cmap_data);
438
439     ptr = cmap_data;
440     for (i = 0; i < cmapsize; i++) {
441         cdefs[i].flags = DoRed | DoGreen | DoBlue;
442
443         /*
444          * since 24-bit XIL images are in BGR order, colormaps are also in
445          * BGR order
446          */
447         cdefs[i].blue = *ptr++ << 8;
448         cdefs[i].green = *ptr++ << 8;
449         cdefs[i].red = *ptr++ << 8;
450     }
451     XStoreColors(display, rcmap, cdefs, cmapsize);
452
453     /*
454      * This will cause the colormap to be installed unless the cursor is
455      * moved to another window -- any other window; if this happens, then
456      * colormap flashing may occur.
457      */
458     XSetWindowColormap(display, window, rcmap);
459     XInstallColormap(display, rcmap);
460     XSync(display, False);
461
462
463     /* set the offset of the XilLookup */
464     xil_lookup_set_offset(xil_cmap, offset);
465 }
```

Appendix G   Frame_Grabber.H

```
1      /****************************************************************************
2         Andy Ritger
3         Research Honors
4         4-26-99
5
6       . Frame_Grabber.H
7
8         The Frame_Grabber class is a wrapper for the SIE frame grab API.
9
10        ****************************************************************************/
11
12
13
14     #ifndef ___FRAME_GRABBER_WRAPPER___
15     #define ___FRAME_GRABBER_WRAPPER___
16
17
18
19     #include "sie_protocol.h"
20     #include "frame_grabber_protocol.h"
21     #include "shelley_sockets.h"
22
23
24     class Frame_Grabber
25     {
26     public:
27       Frame_Grabber (int sock_number);
28       ~Frame_Grabber () {};
29
30       int get_width () { return width; };
31       int get_height () { return height; };
32
33       void set_scale_factor (float factor);
34
35       void enable_display ();
36       void disable_display ();
37
38       void select_input_port (int32 port);
39
40       void grab_frame (unsigned char *data);
41
42     private:
43       int sock;
```

```
44      int32 width;
45      int32 height;
46      bool error;
47    };
48
49
50
51    #endif
```

Appendix H Frame_Grabber.C

```
1     /*****************************************************************************
2        Andy Ritger
3        Research Honors
4        4-26-99
5
6      . Frame_Grabber.C
7
8        ****************************************************************************/
9
10    #include "Frame_Grabber.H"
11
12
13
14    Frame_Grabber::Frame_Grabber (int sock_number)
15    {
16       // initialize internal things...
17
18       // we make the assumption that we have already connected to the admin
19       sock = sock_number;
20       error = false;
21
22       // what we're doing
23       int8 val = AGENT_SEND_DEVICE;
24       shelley_sockets_write (sock, (char*) &val, sizeof (int8));
25
26       // to which device
27       int32 val32 = 1;
28       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
29
30       // how long the message is
31       val32 = sizeof (int8);
32       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
33
34       // the message (it's about time)
35       val = FRAME_GRAB_REQUEST_RESOLUTION;
36       shelley_sockets_write (sock, (char*) &val, sizeof (int8));
37
38       // the next thing coming back is the resolution
39       shelley_sockets_read (sock, (char*) &width, sizeof (int32));
40       shelley_sockets_read (sock, (char*) &height, sizeof (int32));
41
42    } // constructor
43
```

```
44
45
46   void Frame_Grabber::set_scale_factor (float factor)
47   {
48      if (sock == -1) return;
49
50      // what we're doing
51      int8 val = AGENT_SEND_DEVICE;
52      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
53
54      // to which device
55      int32 val32 = 1;
56      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
57
58      // how long the message is
59      val32 = sizeof (int8) + sizeof (float);
60      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
61
62      // the message (finally)
63      val = FRAME_GRAB_SET_RESOLUTION;
64      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
65      shelley_sockets_write (sock, (char*) &factor, sizeof (float));
66
67      // frame grab returns a yes or no...
68      shelley_sockets_read (sock, (char*) &val, sizeof (int8));
69
70      /* now get the new width and height */
71
72      // what we're doing
73      val = AGENT_SEND_DEVICE;
74      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
75
76      // to which device
77      val32 = 1;
78      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
79
80      // how long the message is
81      val32 = sizeof (int8);
82      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
83
84      // the message (it's about time)
85      val = FRAME_GRAB_REQUEST_RESOLUTION;
86      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
87
88      // the next thing coming back is the resolution
89      shelley_sockets_read (sock, (char*) &width, sizeof (int32));
90      shelley_sockets_read (sock, (char*) &height, sizeof (int32));
91
```

```
92      } // set_scale_factor ()
93
94
95
96      void Frame_Grabber::enable_display ()
97      {
98        if (sock == -1) return;
99
100       // what we're doing
101       int8 val = AGENT_SEND_DEVICE;
102       shelley_sockets_write (sock, (char*) &val, sizeof (int8));
103
104       // to which device
105       int32 val32 = 1;
106       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
107
108       // how long the message is
109       val = sizeof (int8);
110       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
111
112       // the message (it's about time)
113       val = FRAME_GRAB_ENABLE_DISPLAY;
114       shelley_sockets_write (sock, (char*) &val, sizeof (int8));
115
116       // frame grab returns a yes or no...
117       shelley_sockets_read (sock, (char*) &val, sizeof (int8));
118
119     } // enable_display ()
120
121
122
123
124     void Frame_Grabber::disable_display ()
125     {
126       if (sock == -1) return;
127
128       // what we're doing
129       int8 val = AGENT_SEND_DEVICE;
130       shelley_sockets_write (sock, (char*) &val, sizeof (int8));
131
132       // to which device
133       int32 val32 = 1;
134       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
135
136       // how long the message is
137       val32 = sizeof (int8);
138       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
139
```

```
140      // the message (it's about time)
141      val = FRAME_GRAB_DISABLE_DISPLAY;
142      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
143
144      // frame grab returns a yes or no...
145      shelley_sockets_read (sock, (char*) &val, sizeof (int8));
146
147    } // disable_display ()
148
149
150
151
152    void Frame_Grabber::select_input_port (int32 port)
153    {
154      if (sock == -1) return;
155
156      // what we're doing
157      int8 val = AGENT_SEND_DEVICE;
158      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
159
160      // to which device
161      int32 val32 = 1;
162      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
163
164      // how long the message is
165      val32 = sizeof (int8) + sizeof (int32);
166      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
167
168      // the message (it's about time)
169      val = FRAME_GRAB_SELECT_INPUT_PORT;
170      shelley_sockets_write (sock, (char*) &val, sizeof (int8));
171      shelley_sockets_write (sock, (char*) &port, sizeof (int32));
172
173      // frame grab returns a yes or no...
174      shelley_sockets_read (sock, (char*) &val, sizeof (int8));
175
176    } // select_input_port ()
177
178
179
180
181    void Frame_Grabber::grab_frame (unsigned char *data)
182    {
183      if (sock == -1) return;
184      if (data == NULL) return;
185
186      // what we're doing
187      int8 val = AGENT_SEND_DEVICE;
```

```
188    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
189
190    // to which device
191    int32 val32 = 1;
192    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
193
194    // how long the message is
195    val32 = sizeof (int8);
196    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
197
198    val = FRAME_GRAB_GRAB_FRAME;
199    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
200
201    /* read the BIG 1-d array of chars... */
202    shelley_sockets_read (sock, (char *) data, (width * height));
203
204  } // grab_frame ()
```

Appendix I   neural_net_protocol.h

```
1     /*******************************************************************
2        Andy Ritger
3        Research Honors
4        4-26-99
5
6      . The Artificial Neural Network Device Module Protocol
7
8        Note that all constants are sent across sockets as type int8.
9
10       ***************************************************************/
11
12
13
14    #ifndef ___SIE_NEURAL_NETWORK_PROTOCOL___
15    #define ___SIE_NEURAL_NETWORK_PROTOCOL___
16
17
18
19    #define NEURAL_NET_INIT_VALUE 0
20
21
22
23    /*******************************************************************
24       NEURAL_NET_CREATE is followed by three 32-bit integers: the number of input,
25       hidden and output nodes for the new network.
26       ***************************************************************/
27
28    #define NEURAL_NET_CREATE 1
29
30
31
32    /*******************************************************************
33       NEURAL_NET_FEED_FORWARD computes an answer(s) for the network by applying
34       the feed forward algorithm to the input layer and arriving at values at
35       the hidden layer, and similarly using the hidden layer to arrive at values
36       at the output layer.
37       ***************************************************************/
38
39    #define NEURAL_NET_FEED_FORWARD 2
40
41
42
43    /*******************************************************************
```

```
44    NEURAL_NET_LOAD_INPUT_VECTOR is followed by a stream of data which
45    corresponds to the double-precision values of the input layer.  The
46    data stream is row-major (it can be indexed with: ((y * maxx) + x)).
47    *********************************************************************/
48
49    #define NEURAL_NET_LOAD_INPUT_VECTOR 3
50
51
52
53    /********************************************************************
54      NEURAL_NET_LOAD_TARGET_VECTOR is followed by a 32-bit index value indicating
55      which output node we're talking about, which is then followed by the double
56      value which we want to assign to that target node.
57      *****************************************************************/
58
59    #define NEURAL_NET_LOAD_TARGET_VALUE 4
60
61
62
63    /********************************************************************
64      NEURAL_NET_TRAIN trains the network.  NEURAL_NET_YES is returned when
65      training is complete.
66      ****************************************************************/
67
68    #define NEURAL_NET_TRAIN 5
69
70
71
72    /********************************************************************
73      NEURAL_NET_GET_OUTPUT_VALUE is followed by a 32-bit integer which
74      indicates which output node is in question.  A double precision value
75      is returned indicating the value of that node.
76      ***************************************************************/
77
78    #define NEURAL_NET_GET_OUTPUT_VALUE 6
79
80
81
82    /********************************************************************
83      Yes and No responses from the Neural Network
84      ****************************************************************/
85
86    #define NEURAL_NET_YES 7
87    #define NEURAL_NET_NO  8
88
89
90
91    #endif
```

Appendix J neural_net.c

```
1       /**************************************************************************
2          Andy Ritger
3          Research Honors
4          4-26-99
5
6       _ neural_net.C
7
8          This is the artificial neural network device module.  We initiate things,
9          and then listen for commands.  This interfaces with the bpnn class.
10
11         **************************************************************************/
12
13      #include <stdio.h>
14      #include <stdlib.h>
15      #include <iostream.h>
16      #include <fstream.h>
17
18
19
20      #include "sie_protocol.h"
21      #include "neural_net_protocol.h"
22      #include "shelley_sockets.h"
23      #include "bpnn.H"
24
25
26
27      int main (int argc, char **argv)
28      {
29        bpnn *neural_net;
30        int32 input, hidden, output;
31        int32 val32;
32        int8  val8;
33
34        double *data;
35        double value;
36
37        double output_error, hidden_error;
38
39        // connect to administrator on localhost at 2048
40        int sock = shelley_sockets_client_connect_to_server (2048, "localhost");
41        if (sock == -1)
42        {
43           cout << "NEURAL NET: failed to connect to administrator." << endl;
```

```
44          exit (1);
45        }
46
47        // hand shake with administrator
48        shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
49        if (val8 != ADMIN_QUERY_DEVICE)
50        {
51          cout << "NEURAL NET: I am confused." << endl;
52          exit (1);
53        }
54        val8 = DEVICE_READY;
55        shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
56
57        // now we can expect to receive neural net protocol commands
58        int8 command = NEURAL_NET_INIT_VALUE;
59
60        // loop in here until we're told to exit
61        while (command != ADMIN_DISCONNECT_DEVICE)
62        {
63          // block until we get the next command
64          if ((shelley_sockets_read (sock, (char*) &command, sizeof (int8))) == -1)
65            exit (1);
66          else
67          {
68            // perform some action based on what the request is
69            switch (command)
70            {
71            case NEURAL_NET_CREATE:
72        // create a new network, given the dimensions
73
74        shelley_sockets_read (sock, (char*) &input, sizeof (int32));
75        shelley_sockets_read (sock, (char*) &hidden, sizeof (int32));
76        shelley_sockets_read (sock, (char*) &output, sizeof (int32));
77
78        neural_net = new bpnn (input, hidden, output);
79        neural_net->initialize (false, true, 0.0); // yes, it's hardcoded...
80
81        // confirm that we received the data and that the network is created
82
83        val32 = 1;
84        val8  = NEURAL_NET_YES;
85        shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
86        shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
87        break;
88
89            case NEURAL_NET_FEED_FORWARD:
90        // feed what is in the input layer through the network
91        neural_net->feedforward ();
```

```
92
93      // confirm that we did apply feedforward
94      val32 = 1;
95      val8  = NEURAL_NET_YES;
96      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
97      shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
98      break;
99
100         case NEURAL_NET_LOAD_INPUT_VECTOR:
101     // load the input vector
102     data = new double [input];
103     shelley_sockets_read (sock, (char*) data, sizeof (double) * input);
104
105     for (int i = 0; i < input; i++)
106       neural_net->load_input_value (i, data [i]);
107
108     delete (data);
109
110     // confirm that we did load the input vector
111     val32 = 1;
112     val8  = NEURAL_NET_YES;
113     shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
114     shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
115     break;
116
117         case NEURAL_NET_LOAD_TARGET_VALUE:
118     // set the target values
119     shelley_sockets_read (sock, (char*) &val32, sizeof (int32)); // index
120     shelley_sockets_read (sock, (char*) &value, sizeof (double)); // value
121
122     neural_net->load_target_value (val32, value);
123
124     // confirm that we did load the target value
125     val32 = 1;
126     val8  = NEURAL_NET_YES;
127     shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
128     shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
129     break;
130
131         case NEURAL_NET_TRAIN:
132     // train the network
133
134     neural_net->train (0.3, 0.3, &output_error, &hidden_error);
135
136     // confirm that we did train the network
137     val32 = 1; // our response will be 1 byte
138     val8  = NEURAL_NET_YES;
139     shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
```

```
140    shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
141    break;
142
143        case NEURAL_NET_GET_OUTPUT_VALUE:
144    // return the value of an output node
145    shelley_sockets_read (sock, (char*) &val32, sizeof (int32)); // index
146    value = neural_net->get_output_value (val32);
147
148    // send back the value
149    val32 = sizeof (double);
150    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
151    shelley_sockets_write (sock, (char*) &value, sizeof (double));
152
153    break;
154
155        case ADMIN_DISCONNECT_DEVICE:
156    // we should exit now
157    exit (0);
158    break;
159
160        } // switch
161
162      } // if
163
164     } // while
165
166    } // main ()
```

Appendix K Neural_Net.H

```
1    /*****************************************************************
2       Andy Ritger
3       Research Honors
4       4-26-99
5
6    _ Neural_Net.H
7
8       The Neural_Net class is a wrapper for the SIE neural net API.
9
10      *****************************************************************/
11
12
13
14   #ifndef ___NEURAL_NET_WRAPPER___
15   #define ___NEURAL_NET_WRAPPER___
16
17
18
19   #include "sie_protocol.h"
20   #include "neural_net_protocol.h"
21   #include "shelley_sockets.h"
22
23
24
25   class Neural_Net
26   {
27   public:
28     Neural_Net (int sock_number);
29     ~Neural_Net () {};
30
31     void feedforward ();
32     void load_input_vector (double* data);
33     void load_target_value (int32 index, double value);
34     void train ();
35     double get_output_value (int32 index);
36
37
38   private:
39     int sock;
40     bool error;
41   };
42
43
```

```
44
45      #endif
```

Appendix L   Neural_Net.C

```
1    /********************************************************************
2        Andy Ritger
3        Research Honors
4        4-26-99
5
6      . Neural_Net.C
7
8        *******************************************************************/
9
10   #include "Neural_Net.H"
11
12
13
14   #define _SIZE_ 9612 // this is only a temporary fix - the number of inputs
15
16
17
18   Neural_Net::Neural_Net (int sock_number)
19   {
20       int8 val8;
21       int32 val32;
22
23       // initialize internal things
24
25       // we make the assumption that we have already connected to the admin
26       sock = sock_number;
27       error = false;
28
29       // what we're doing
30       val8 = AGENT_SEND_DEVICE;
31       shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
32
33       // to which device
34       val32 = 2;
35       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
36
37       // how long the message is
38       val32 = (sizeof (int8)) + (sizeof (int32) * 3);
39       shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
40
41       // the message (it's about time)
42       val8 = NEURAL_NET_CREATE;
43       shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
```

```
44
45      val32 = _SIZE_; // input layer
46      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
47
48      val32 = 4; // hidden layer
49      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
50
51      val32 = 4; // output layer
52      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
53
54      // neural net returns a yes or no...
55      shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
56
57    } // constructor
58
59
60
61
62    void Neural_Net::feedforward ()
63    {
64      int8 val8;
65      int32 val32;
66
67      // what we're doing
68      val8 = AGENT_SEND_DEVICE;
69      shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
70
71      // to which device
72      val32 = 2;
73      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
74
75      // how long the message is
76      val32 = sizeof (int8);
77      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
78
79      // the message (it's about time)
80      val8 = NEURAL_NET_FEED_FORWARD;
81      shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
82
83      // neural net returns a yes or no...
84      shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
85
86    } // feedforward ()
87
88
89    void Neural_Net::load_input_vector (double* data)
90    {
91      int8 val8;
```

```
92          int32 val32;
93
94          // what we're doing
95          val8 = AGENT_SEND_DEVICE;
96          shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
97
98          // to which device
99          val32 = 2;
100         shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
101
102         // how long the message is
103         val32 = (sizeof (double) * _SIZE_) + (sizeof (int8));
104         shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
105
106         // the message (it's about time)
107         val8 = NEURAL_NET_LOAD_INPUT_VECTOR;
108         shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
109         shelley_sockets_write (sock, (char*) data, sizeof sizeof(double) * _SIZE_);
110
111         // neural net returns a yes or no...
112         shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
113
114    } // load_input_vector ()
115
116
117
118    void Neural_Net::load_target_value (int32 index, double value)
119    {
120         int8 val8;
121         int32 val32;
122
123         // what we're doing
124         val8 = AGENT_SEND_DEVICE;
125         shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
126
127         // to which device
128         val32 = 2;
129         shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
130
131         // how long the message is
132         val32 = (sizeof (double)) + (sizeof (int32)) + (sizeof (int8));
133         shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
134
135         // the message (it's about time)
136         val8 = NEURAL_NET_LOAD_TARGET_VALUE;
137
138         shelley_sockets_write (sock, (char*) &val8, sizeof (int8)); // command
139         shelley_sockets_write (sock, (char*) &index, sizeof (int32)); // index
```

```
140      shelley_sockets_write (sock, (char*) &value,  sizeof (double)); // value
141
142      // neural net returns a yes or no...
143      shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
144
145  } // load_target_value ()
146
147
148
149
150  void Neural_Net::train ()
151  {
152      int8 val8;
153      int32 val32;
154
155      // what we're doing
156      val8 = AGENT_SEND_DEVICE;
157      shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
158
159      // to which device
160      val32 = 2;
161      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
162
163      // how long the message is
164      val32 = sizeof (int8);
165      shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
166
167      // the message (it's about time)
168      val8 = NEURAL_NET_TRAIN;
169      shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
170
171      // neural net returns a yes or no...
172      shelley_sockets_read (sock, (char*) &val8, sizeof (int8));
173
174  } // train ()
175
176
177
178  double Neural_Net::get_output_value (int32 index)
179  {
180      int8 val8;
181      int32 val32;
182
183      // what we're doing
184      val8 = AGENT_SEND_DEVICE;
185      shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
186
187      // to which device
```

```
188    val32 = 2;
189    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
190
191    // how long the message is
192    val32 = sizeof (int8) + sizeof (int32);
193    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
194
195    // the message (it's about time)
196    val8 = NEURAL_NET_GET_OUTPUT_VALUE;
197    shelley_sockets_write (sock, (char*) &val8, sizeof (int8));
198    shelley_sockets_write (sock, (char*) &index, sizeof (int32)); // index
199
200    double value;
201
202    // neural net returns double
203    shelley_sockets_read (sock, (char*) &value, sizeof (double));
204
205    return (value);
206
207  } // get_output_value ()
```

Appendix M   agent.C

```
1     /*************************************************************************
2        Andy Ritger
3        Research Honors
4        4-26-99
5
6      . agent.C
7
8        This is an example agent which initially presents the user with options to:
9
10       [1] identify user
11       [2] capture frames to pgm
12       [3] train
13       [4] quit
14
15       Identifying the user (1) grabs a frame of video, feeds it into the input of
16       the neural network, and simply reports the output values.
17
18       Capturing frames of video to pgm (2) grabs X number of video frames, and
19       saves them as pgm images for future use in training.
20
21       Training (3) uses the pgm images and trains the network to recognize the
22       faces of the people in the pgms.
23
24
25
26       More important than the face recognition functionality that this agent
27       provides, this program demonstrates a simple example of using SIE and
28       how an agent should interact with the administrator and send requests to
29       devices.
30
31       *************************************************************************/
32
33
34
35     #include <stdio.h>
36     #include <stdlib.h>
37     #include <iostream.h>
38     #include <fstream.h>
39
40     #include "sie_protocol.h"
41     #include "shelley_sockets.h"
42     #include "Frame_Grabber.H"
43     #include "Neural_Net.H"
```

```cpp
44    #include "pgmImage.H"
45
46
47
48    #define _SIZE_ 6912 // this is a temporary fix -- the size of the network input
49
50
51
52    /**********************************************************************
53      Given a byte stream of data, writes a pgm image to disk.
54      **********************************************************************/
55
56    void write_pgm_file (char* filename, unsigned char *data,
57          int width, int height)
58    {
59      // streams are better than file handlers
60
61      ofstream *foobar;
62      foobar = new ofstream (filename);
63
64      *foobar << "P2" << endl;
65      *foobar << width << " " << height << endl;
66      *foobar << "255" << endl;
67
68      for (int y = 0; y < height; y++)
69      {
70        for (int x = 0; x < width; x++)
71          *foobar << (int) data [(y * width) + x] << " ";
72        *foobar << endl;
73      }
74      foobar->close();
75    } // write_pgm_file ()
76
77
78
79    /**********************************************************************
80      Requests a frame of video, passes it to the input of the neural network,
81      and prints the resulting outputs of the network.
82      **********************************************************************/
83
84    void identify (Frame_Grabber *grabber, Neural_Net *network)
85    {
86      // get the current dimensions of the frame
87      int size = grabber->get_width () * grabber->get_height ();
88
89      // allocate memory for the data
90      unsigned char *data = new unsigned char [size];
91      double *inputs = new double [size];
```

```
92
93      // allow the frame to be shown to the screen, and
94      grabber->enable_display ();
95      grabber->grab_frame (data);
96
97      // convert from chars to doubles 0.0 >= x > 1.0
98      for (int i = 0; i < size; i++)
99        inputs [i] = ((double) data [i]) / ((double) 256.0);
100
101     // feed the converted frame data into the network's inputs
102     network->load_input_vector (inputs);
103     network->feedforward ();
104
105     // print the results
106     for (int i = 0; i < 4; i++)
107       cout << "   [" << i << "] = " << network->get_output_value (i);
108     cout << endl;
109
110     // disable the video display
111     grabber->disable_display ();
112
113     // free the memory that we allocated
114     delete (data);
115     delete (inputs);
116
117   } // identify ()
118
119
120
121   /*************************************************************************
122     Reads image.list and loads the specified images, training the network
123     to recognize each.  For more details on the network is trained, see the
124     bpnn.H documentation, as well as documentation relating to the GNNV
125     project (www.iwu.edu/~shelley/gnnv).
126     *************************************************************************/
127
128   void train (Neural_Net *network)
129   {
130     // load the images
131     pgmImageList *list = new pgmImageList ("image.list");
132
133     cout << "number of epochs: ";
134     int max_epochs = 10;
135     cin >> max_epochs;
136
137     int epoch;
138     double output_error, hidden_error, error_sumation;
139     int numcorrect = 0;
```

```
140      double sum_error = 0.0;
141      double value;
142      double *inputs = new double [_SIZE_];
143      pgmImage *img;
144      double answers [4];
145      int index;
146
147      // for each epoch, we examine all the images in the image list
148      for (epoch = 0; epoch < max_epochs; epoch++)
149      {
150        numcorrect = 0;
151        for (int i = 0; i < list->numberOfImages (); i++)
152        {
153          img = list->getImage (i);
154
155          // load the input vector
156          index = 0;
157          for (int j = 0; j < img->rows; j++)
158    for (int k = 0; k < img->cols; k++)
159    {
160      inputs [index]  = ((double) (img->getPixel (j, k))) / 256.0;
161      index++;
162    }
163
164          // feed the input data to the network
165          network->load_input_vector (inputs);
166
167          /* load the target vector*/
168
169          // start all targets low
170          network->load_target_value (0, 0.1);
171          network->load_target_value (1, 0.1);
172          network->load_target_value (2, 0.1);
173          network->load_target_value (3, 0.1);
174
175          /*
176    For this test, we use the simple convention where the name of all
177    image for person 1 begin with the number 1.  For example, a pgm
178    filename may be: 1.4.pgm, which means that it is picture number 4 for
179    person 1.  This way, we can just look at the first character of the
180    name when we want to load the target vector.
181          */
182
183          if (img->basefilename [0] == '0')
184    network->load_target_value (0, 0.9);
185          else if (img->basefilename [0] == '1')
186    network->load_target_value (1, 0.9);
187          else if (img->basefilename [0] == '2')
```

74

```
188     network->load_target_value (2, 0.9);
189         else if (img->basefilename [0] == '3')
190     network->load_target_value (3, 0.9);
191
192         // train...
193         network->train;
194
195         // count for ourselves
196         answers [0] = network->get_output_value (0);
197         answers [1] = network->get_output_value (1);
198         answers [2] = network->get_output_value (2);
199         answers [3] = network->get_output_value (3);
200
201         if ((img->basefilename [0] == '0') &&
202     (answers [0] > 0.5))
203     numcorrect++;
204
205         else if ((img->basefilename [0] == '1') &&
206             (answers [1] > 0.5))
207     numcorrect++;
208
209         else if ((img->basefilename [0] == '2') &&
210             (answers [2] < 0.5))
211     numcorrect++;
212
213         else if ((img->basefilename [0] == '3') &&
214             (answers [3] < 0.5))
215     numcorrect++;
216
217       } // each image
218
219       cout << "epoch: " << epoch << " " << numcorrect << " correct" << endl;
220
221     } // each epoch
222
223     // free all the memory we allocated
224     delete (inputs);
225     delete (list);
226
227 } // train ()
228
229
230
231 /*************************************************************************
232     Here we grab frames of video, and send the data to the write_pgm_file ()
233     function to save it in pgm file format.
234     *************************************************************************/
235
```

```
236    void capture_frames (Frame_Grabber *grabber)
237    {
238        // get the user's number (0-3)
239        cout << "please enter your number [0-3]: ";
240        int name;
241        cin >> name;
242
243        // get the number of frames to grab (which is how many pgms will be saved)
244        cout << "please enter the number of frames to grab: ";
245        int frames;
246        cin >> frames;
247
248        char filename [50];
249
250        // this is where we put the frames
251        unsigned char *data = (unsigned char *) malloc (grabber->get_width () *
252         grabber->get_height ());
253
254        // enable the video display of the grabbed frame
255        grabber->enable_display ();
256
257        // for each frame we want to grab
258        for (int i = 0; i < frames; i++)
259        {
260            // get the data
261            grabber->grab_frame (data);
262
263            // make the filename
264            sprintf (filename, "images/%d.%d.pgm", name, i);
265
266            // send the data off to be written to disk
267            write_pgm_file (filename, data, grabber->get_width (),
268             grabber->get_height ());
269
270            // pause for 1 second
271            sleep (1);
272        }
273
274        // disable the display
275        grabber->disable_display ();
276
277        // free the memory that we allocated
278        free (data);
279
280    } // capture frames
281
282
283
```

```
284   /******************************************************************
285     main ()
286     ******************************************************************/
287
288   int main (int argc, char **argv)
289   {
290     // connect to the administrator; sock is the socket identifier
291     int sock = shelley_sockets_client_connect_to_server (2048, "localhost");
292
293     // tell the administrator that we're an agent
294     int8 val = AGENT_CONNECT;
295     shelley_sockets_write (sock, (char*) &val, sizeof (int8));
296
297     // this is the administrator telling us that he is an administrator
298     shelley_sockets_read (sock, (char*) &val, sizeof (int8));
299
300     // request devices...
301
302     val = AGENT_DEVICE_REQUEST;
303     shelley_sockets_write (sock, (char*) &val, sizeof (int8));
304     int32 val32 = 1; // request frame grabber
305     shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
306
307     // eventually we'll need to listen for a response
308
309     val = AGENT_DEVICE_REQUEST;
310     shelley_sockets_write (sock, (char*) &val, sizeof (int8));
311     val32 = 2; // request neural net
312     shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
313
314     // eventually we'll need to listen for a response
315
316     // tell the administrator we're done requesting devices
317     val = AGENT_DEVICE_REQUEST_DONE;
318     shelley_sockets_write (sock, (char*) &val, sizeof (int8));
319
320     // talk to the frame grabber
321     Frame_Grabber *grabber = new Frame_Grabber (sock);
322     int width, height;
323     width = grabber->get_width ();
324     height = grabber->get_height ();
325     cout << "Initial frame dimensions are "
326          << width << " x " << height
327          << endl;
328     grabber->set_scale_factor (0.15);
329     width = grabber->get_width ();
330     height = grabber->get_height ();
331     cout << "Frames resized to "
```

```
332            << width << " x " << height
333            << endl;
334
335     // talk to the neural network
336     Neural_Net *network = new Neural_Net (sock);
337
338     // present the user with our menu
339     int choice = 0;
340
341     while (choice != 4)
342     {
343       printf ("[1] identify user\n");
344       printf ("[2] capture frames to pgm\n");
345       printf ("[3] train\n");
346       printf ("[4] quit\n");
347
348       cout << "choice: ";
349       cin >> choice;
350
351       if (choice == 1)
352         identify (grabber, network);
353
354       else if (choice == 2)
355         capture_frames (grabber);
356
357       else if (choice == 3)
358         train (network);
359     }
360
361     // disconnect from the administrator
362     val = AGENT_DISCONNECT;
363     shelley_sockets_write (sock, (char*) &val, sizeof (int8));
364
365   } // main ()
```

Appendix N   Makefile

```
1      # Andy Ritger
2      # Research Honors
3      # 4-26-99
4
5      # Makefile for demonstation of SIE
6
7      CC =                g++
8      LIBS =              -lsocket -lnsl
9      XILHOME =           /opt/SUNWits/Graphics-sw/xil
10     CFLAGS =            -I$(OPENWINHOME)/include
11     XILLIBS =           -L$(XILHOME)/lib -L$(OPENWINHOME)/lib\
12                         -lxil -lX11 -ldl -ldga -lm -lthread\
13                         -R $(XILHOME)/lib:/usr/openwin/lib\
14                         -lsocket -lnsl
15
16     all:                administrator agent frame_grabber neural_net
17
18     clean:
19                         rm -f core *.o *~
20
21     administrator:      administrator.c sie_protocol.h
22                         $(CC) $(LIBS) administrator.c -o administrator
23
24     agent:              agent.C shelley_sockets.o Frame_Grabber.o \
25                         Neural_Net.o sie_protocol.h pgmImage.o
26                         $(CC) $(LIBS) shelley_sockets.o agent.C \
27                         Frame_Grabber.o Neural_Net.o pgmImage.o -o $@
28
29     frame_grabber:      frame_grabber.o shelley_sockets.o
30                         $(CC) $(XILLIBS) frame_grabber.o \
31                         shelley_sockets.o -o $@
32
33     neural_net:         neural_net.C neural_net_protocol.h bpnn.o \
34                         shelley_sockets.o
35                         $(CC) $(LIBS) neural_net.C bpnn.o \
36                         shelley_sockets.o -o $@
37
38     shelley_sockets.o:  shelley_sockets.c shelley_sockets.h
39                         $(CC) -c $<
40
41     frame_grabber.o:    frame_grabber.c frame_grabber_protocol.h sie_protocol.h
42                         $(CC) $(CFLAGS) -c $<
43
```

```
44    Frame_Grabber.o:      Frame_Grabber.C Frame_Grabber.H
45                          $(CC) $(CFLAGS) -c $<
46
47    Neural_Net.o:         Neural_Net.C Neural_Net.H
48                          $(CC) $(CFLAGS) -c $<
49
50    bpnn.o:               bpnn.C bpnn.H
51                          $(CC) $< -c
52
53    pgmImage.o:           pgmImage.C pgmImage.H
54                          $(CC) $< -c
```

References

[1] Stuart Russell and Peter Norvig, *Artificial Intelligence, A Modern Approach* (Prentice-Hall 1995).

[2] *Illinois Wesleyan Intelligence Network on Knowledge, member Cognitive Science Consortium.* http://www.iwu/edu/~lshapiro/wink.html

[3] Abraham Silberschatz and Peter B. Galvin, *Operating System Concepts* (Addison-Wesley 1995).

[4] Graham Glass, *Unix for Programmers and Users* (Prentice Hall 1993).

[5] W. Richard Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols* (Addison-Wesley 1994).

[6] W. Richard Stevens, *Unix Network Programming: Interprocess Communications Vol. 2*, Second Edition (Prentice Hall 1999).

[7] Douglas Gage, Network Protocols for Mobile Robot Systems, SPIE Mobile Robots XII, Pittsburgh, PA (October 1997).

[8] Bjarne Stroustrup, AT&T Labs, *The C++ Programming Language*, 3rd Edition (Addison-Wesley 1997).

[9] Joseph L. Jones and Anita M. Flynn, *Mobile Robots, Inspiration to Implementation* (A K Peters 1993).

[10] Rodney A. Brooks, *Elephants Don't Play Chess*, Robotics and Autonomous Systems 6 (MIT Press 1990).

[11] Tom M. Mitchell, *Machine Learning* (McGraw-Hill 1997).

Acknowledgements