UCSC Java Network Computer Procedures [1]

UCSC-CRL-96-30

Bruce R. Montague [2], Elizabeth A. Baldwin [3], and Mike R. Allen [4]
Computer Science Department
University of California, Santa Cruz

10 Dec 1996

[1]This work was supported in part by a gift from National Semiconductor.
[2]brucem@cse.ucsc.edu
[3]libby@cse.ucsc.edu
[4]mallen@cse.ucsc.edu

Abstract

This document describes how to work with the UCSC Java and JN source. Although intended primarily for developers, this document is of use to anyone interested in the UCSC Java system or JN. This document primarily discusses operational procedures, such as how to write a UCSC Java application, and how to compile and build the system. Code internals are not described.
keywords: JAVA, JN, procedures.

Contents

Chapter 1

Introduction

1.1 Overview

This document describes how to work with JN and UCSC Java. Although intended primarily for developers, it is of use to anyone interested in JN and UCSC Java.

1.2 What is JN and UCSC Java?

The Java Nanokernel (**JN**) is a small kernel that has been designed specifically to support the Java virtual machine (the **JVM**). JN currently runs only on custom National Semiconductor 486 evaluation boards, although an earlier version ran under a CR32 (National Semiconductor Compact Risc) simulator.

1.2.1 The National Semiconductor NS486

The NS486 is essentially a 486 PC *motherboard on a chip*. It consists of a 486SX core and a UART (serial ports), a PCMCIA interface (smart card), a PC/104 interface (the industrial/embedded system version of the ISA bus), an enhanced parallel port (high-speed/bidirectional), a DRAM controller, DMA controller, 2 peripheral interrupt controllers, multiple real-time clocks, a clock-calendar, an LCD controller, and a synchronous serial interface. All these devices are on the CPU chip. Additional devices can be external to the chip.

The layout of the registers for these devices in x86 I/O space is under program control via a *bus interface* controller. Thus, most components needed to create a system other then the DRAM and connector glue are on the chip. To achieve this, the NS486 does not support virtual memory via page tables (it *does* have segmentation, however), does not support floating-point, and does not support Virtual 8086 mode, that is, the NS486 boots in 32-bit protected mode. The NS486 only runs in 32-bit protected mode. It cannot run old 16-bit code, and thus it cannot run DOS. The NS486 is intended to be used for embedded real-time applications, such as in industrial controllers, instrumentation, fax machines, and network computers.

1.2.2 The Java Virtual Machine

The JVM is basically a single C switch statement, with the cases of the switch identified by the current Java opcode (the Java *bytecode*). Since compiler writers have spent a lot of effort optimizing C switch statements, the JVM is fairly efficient. Java code can call C subroutines. Such subroutines, called *native methods*, are used to perform many run-time tasks such as I/O and C-level thread support. Calls to native methods are performed by the Java run-time, primarily performing I/O and supporting multiple threads concurrently. These calls effectively define an API required to run the JVM.

Java has 2 levels of threading. *High-level* threading occurs at the Java interpreter level. At this level JVM context's are multithreaded by the interpreter. *Low-level* threading occurs at the Java C-runtime level. The Java C-runtime implements the API (Application Programming Interface) supporting the JVM. The Java C-runtime must be multi-threading so that low-level run-time activities can be maintained corresponding to each high-level thread. Examples of such low-level activities are I/O management and high-level interpreter context. The original Java implementation used *Greenthreads*, a conventional multi-threading C runtime developed by Sun. Versions of Greenthreads existed which used both the Unix and Win32 APIs to provide lower-level system services.

1.2.3 JavaOS

The *JavaOS*, in the past known as *Kona*, implemented as much of Greenthreads in Java as possible. A small set of C routines remain which enable the JVM to run *stand-alone*, that is, without using the services of a lower-level operating system. The remaining C code primarily handles interrupt linkage and physical memory access. In JavaOS, even drivers are written in Java. Since memory protection is managed by the interpreter, JavaOS and application Java threads are always in 486 Supervisor Mode, that is, JavaOS does not use hardware supported User Mode.

This approach resembles a stand-alone multi-threading Basic (such as that upon which RSTS/E was based). Other systems that have used a similar approach in the past include JOSS, MUMPS, and PICK. Although this approach proved very convenient, this approach historically did not lend itself to high-performance real-time (especially with regard to character interrupt devices, or the like). These systems have all become high-end interpretive languages hosted on conventional kernels. Time will tell if the current cost of compute power has fallen to the point where this approach is now effective.

1.2.4 The Java Nanokernel

JN is a small, conventional, event-driven soft–real-time kernel. Its overall design is 'classic' in that its *deep structure*, or *software architecture*, resembles that of the low-level kernels of TSS, RSX, VMS, and WindowsNT. Although the term is informal, JN is called a *nanokernel* because it only contains the lowest-level functionality found in these kernels. The JN nanokernel only provides 1 system service (API call). This service runs any specified subroutine as a *kernel subroutine*, that is, serialized with respect to kernel activity. The subroutine thus runs as a *critical section*.

On top of the JN nanokernel are *emulator* services for the file, thread, synchronization, event, and network services that were originally provided by a combination of Greenthreads and Unix (or Win32).

The emulator services were determined by linking Java without Greenthreads. Happily, the required API's are for the most part mundane, mapping almost directly to Java language synchronization functions or Unix I/O. Although Greenthread source was occasionally checked to determine function semantics, JN is very much not simply a port of Greenthreads. Most of the functionality was implemented by considering what the calling routine required, rather than looking at the (dissimilar) Greenthreads implementation. Additionally, some functions were discarded as unneeded in a stand-alone environment.

JN is called a *nanokernel* because:

- There is really only 1 system service, that is, the JN API consists of a single entry point: `thrd_kcall()`. This routine is passed the address of a subroutine and a single arbitrary argument. These values are placed in a queue which is processed by the kernel. When the kernel reaches the corresponding location in the queue, the designated subroutine, with the specified argument, is executed by the kernel as a critical section. The subroutine executes to completion. The kernel executes all its required functionality in this manner. It is fair to think of the nanokernel as an interrupt routine with a central work-loop in which interrupts are enabled and in which subroutines are executed as specified by a subroutine ready queue. All enqueued subroutines are executed to completion and cannot block. Hardware interrupts can occur when such a subroutine is running, but if an interrupt routine has to do anything beyond a trivial amount of work, it explicitly queues a subroutine execution to the kernel. When the nanokernel has nothing else to do, the interrupt routine that constitutes the nanokernel returns from the interrupt that invoked the nanokernel.

- There is no uniform *device database*. No high-level hardware abstraction is provided by a standard kernel hardware or driver model. Each driver is free to do as it desires with respect to data structures providing a model of the hardware the driver is managing. There is, however, a standard model of how drivers interact with the nanokernel itself.

- All code, both JN nanokernel code and application code, runs at supervisor privilege.

- JN only support light-weight threads. JN has no memory management or other protection between threads. Per-thread segment and mapping tables are not used.
- All code is linked together in a single system image. There is no provision for loading C code on the fly. There is only 1 segment table, the Global Descriptor Table (**GDT**), and all code is loaded into 1 flat 32-bit segment which starts at address 0 and is mapped 1:1 with real physical memory addresses.
- Services such as connecting driver interrupt routines to an interrupt vector are not included in the nanokernel, but are rather are performed by a special *embedded system linker* which builds a complete system image, including such data structures as the 486 IDT and GDT.

Chapter 2

Setting up the Hardware

The 2 UART ports on the evaluation board are marked `UART` and `DEBUG`. The `DEBUG` cable is a straight-thru cable and is connected to the PC's COM1 port.

Note the serial mice on the PC's are connected to COM2, which is mildly unusual, and can cause confusion.

The UART port marked `UART` is connected to the VT100 via a standard null modem cable. The VT100's need to be manually set to 9600 baud for both transmit and receive (press `SET-UP`, then `SETUP A/B` (the 5 key), then use the transmit and receive speed keys (the 7 and 8 keys) to set the VT100 baud rate). The bottom of a VT100 keyboard has a command help/map. It is convenient to enable VT100 smooth scroll.

The 10-baseT Ethernet cables can be disconnected to and from the controller and network hub at any time. The network hub is daisy-chained to the Baskin center 14 net hub (128.114.14.x) via a 10-baseT cross connector connected to the network tap in the wall. The hub can be disconnected from the wall at any time, that is, the cable can be connected and disconnected without any harm (unless of course you happen to be in the middle of a network operation). This can be useful if you want to eliminate large random RIP routing protocol table update messages while debugging networking code.

Chapter 3

Development Cycle Procedures

3.1 The Development Cycle

The following steps describe the typical development cycle, using a NS486 evaluation board and a DOS PC. Currently all development is directly hosted on DOS. The free tools provided with the NS486 evaluation board are all DOS based (the commercial versions are available in NT versions). Although the DOS tools can be made to run under NT, the Metaware compiler requires the PharLap DOS extender, and running this under NT is excruciatingly slow, even with 32 Mbytes. Additionally, the serial port throughput appears at least 2-3 times as fast under DOS when downloading. Since the development cycle is fast and DOS specific, no routine use is made of the standard Baskin center environment.

The following steps show how to build, link, load and run the master version on the PC `masterblend`. Section 3.1.3 describes how to check out your own version of the system for editing. The commands which are batch files are described in detail in Section 3.5. It is useful to know what the batch files do and their input files, as some of them may need to be modified for use with a checked out version.

```
dos> b                       <-- pwb will start. Edit source files.
dos> nmaker                  <-- A "make" will compile all needed files.
dos> e                       <-- A link will be done, leaving the output
                                  in file "\javanew.omf". Note that link
                                  errors are not necessarily fatal.

dos> f                       <-- Change to \host and run the flashloader

dos> esp                     <-- Run the utility ESP3 (includes loading)

esp> go                      <-- At the ESP3 prompt
                             <-- JN output should appear on the VT100
                                  connected to the board.
                             <-- At this point test output will occur on the VT100 screen.
                                  Press RETURN on the VT100 2 times, when prompted at the
                                  end of the output. If the VT100 appears frozen, press
                                  "NOSCROLL" (the key on the bottom left).
VT100: -----------------------
RETURN
RETURN

top> j                       <-- At this point the JN top-level "top>" prompt will
                                  appear. Type "h" for help, or "j" to start Java.

java> thttp                  <-- At this point the Java web server is running.
                                  An "UnknownHostException" upon startup is
                                  currently normal. The web server runs ‘‘forever’’.

java> wwwsimp ./index_tmp.html
                             <-- An alternate web server. A very simple
```

```
 server. Serves up one file (the argument).
```

Whenever possible, upon booting JN, run the `port_read` program on DOS as described below, so that any diagnostic or crash output will be captured.

Execution of the DOS ESP3 program can occur at any time after JN is booted (the batch file `esp` runs program ESP3). To terminate ESP3, press Control-C or Control-Break. After a few seconds ESP3 will respond with a prompt. At the prompt, enter `quit`.

3.1.1 Using the `uart_printf` function

A C function and a UART driver have been implemented to allow output from the system to be sent to the PC's monitor (that is, the DOS terminal). Use the command `uart_printf` in C code just as you would use a `printf` statement. To capture the output, after starting the system (after typing `go` at ESP3 prompt, but before entering anything at the VT100 terminal) press Control-C or Control-Break and wait for the ESP3 prompt. Enter `quit` to get back to the DOS prompt. From the `\host` directory, enter the following:

```
dos> portread <filename.out>
```

A message will be returned to the screen confirming the port read and the output filename. If no file name is specified the output will scroll off the screen. Now run the Java system in the normal way. To stop the output capture at any time, hit the space bar or any other key except control-C. The output will be written to the specified file.

3.1.2 Obtaining a Crash Dump

If the system has crashed or appears hung, you may want to generate a crash dump. To get a crash dump you must run `portread` on the DOS host as previously described. You can start `portread` at any time after JN boots, although it is most useful to do so immediatly so that any crash message will be captured in the crash dump output file.

To generate a crash dump simply press the NMI switch on the NS486 motherboard. This switch is away from the power LED and next to the board battery (which appears about the size and shape of a quarter). Once pressed, output will appear on the DOS screen and be captured in the portread output file.

The crash dump output contains the PC at the time the NMI switch was pressed, and much other information. A complete crash dump may be quite large.

3.1.3 Checking Out a Version for Editing

The master copy of the system is kept on the PC `masterblend` in the `\master` directory, and contains a working, up-to-date version. This directory contains a complete package: all source files, object files, a flat source directory, and all make and link specification files. The procedure to check out a version of the system is to create a new root level directory and copy the entire `\master` directory to it. This can be done easily using the `pkzip` utility used for making backups as described in Section 6. To use the existing batch files, copy each batch file which references files in `\master` and replace `\master` with the new directory name. Note that the flashloader batch file (`f`) and the ESP3 batch file (`esp`) do not reference the `\master` directory and do not need to be changed.

After you have modified the checked out version and tested it thoroughly, coordinate with the other developers to integrate the new parts of the system into the master version. Build and test the new master version to verify that the new changes are working correctly and have not broken anything (regression testing).

3.1.4 Editing

The editor is `pwb` (Programmer's Workbench). It is not required, it is simply the default DOS editor distributed with Microsoft's MASM DOS assembler. The `pwb` editor can keep a current edit context in dozens of files simultaneously, and will maintain these contexts across editing sessions (and machine boots). Thus, it does not really matter from which directory `pwb` is run.

- To change to the build directory and start `pwb` (this will leave you ready for a compile and link in the build directory when you exit `pwb`):

  ```
  dos> b
  ```

  This is a single-character user-defined DOS batch file, located in directory `\bat`.

- To simply start `pwb` in the current directory:

  ```
  dos> pwb
  ```

3.1.5 Compiling and Linking

The directory structure of the build kit follows Sun's Java directory tree. The development directory is:

```
c:\master\j\build\ns486\java\java
```

To compile:

- *cd* to the development directory, either manually or by using the following single-character custom command:

  ```
  dos> c
  ```

  If you have used the single-character `b` command to invoke `pwb`, you will already be in the build directory when you exit `pwb`.

- Compile via *nmaker*:

  ```
  dos> nmaker
  ```

This is a Microsoft version of *make*. Beware that there is also an `nmake` utility – they are not the same. Assure that `nmaker` completes successfully before continuing.

The output object files are placed in directory `\master\obj_g`. Delete all the object files in this directory to force a complete recompile. This can be accomplished by `nmaker clean` in the build directory. The `makefile` in the build directory controls the compile. The makefile also uses 3 `.arg` files in the build directory which contain C compiler options. These are not in the makefile because of DOS command line length limitations. Default Metaware compiler arguments are also specified in the file `c:\highc\bin\hc386set.cnf`

- Link via the single-character command `e`. The full command line is:

```
dos> elink386 @javanew.cmd
```

The commands controlling the link are not in the makefile, they are in the file `javanew.cmd`. This was done to get around linker, makefile, and DOS limitations.

The output of the link is file `c:\javanew.omf`. There is also a very useful map file, file `c:\javanew.map`. Note that link errors are not necessarily fatal. there are currently 2 link errors, resulting from bugs in Metaware libraries, which have to do with floating point to 64-bit integer conversion. We have been ignoring these link errors.

3.1.6 Booting

- Assure there is a current `\javanew.omf` file. This may require a build, that is, a compile and link.
- `cd \host`. This directory contains SSI and National Semiconductor tools. This directory contains 2 download utilities, the `flashldr`, and `ESP3`, the Softprobe loader/debugger.
- Run the flashldr: `flashldr`
- Press the `RESET` button on the development board. This switch is adjacent to the green power LED on the board. Assure the switch wired to the `MONITOR` jumper is in the leftmost position. This switch is simply glued to the bottom of the board near the power connectors. When facing the switch, its pole should be in the position towards the power connectors. Under normal circumstances, this switch should not be touched. If the board does not have a switch, assure that jumper `W6`, the monitor jumper, is strapped. The above will cause the `RESET` button to boot a small download utility from on-board flash memory that cooperates with the `flashldr`. The operation of this download utility can be verified by, instead of executing the `flashldr`, executing program `c:\host\in_blast`. If the board is awaiting the `flashldr`, and the `DEBUG` cable is connected to the PC correctly, this program should output `A`'s to the screen about twice a second.
- Type: `init`. This will establish communication between the `flashldr` and the boot routine in the board's flash ROM. You must wait at this point until a message, `target system on-line`, is output.
- Type `dir`. This is not essential, but is a good check that the board and the `flashldr` are running and communicating normally. A simple directory of the contents of flash memory will be displayed.
- Run the board side of the SoftProbe load/debugger: `boot ns96mon`
- exit the flashldr: `quit`
- Run the SoftProbe loader/debugger: `esp`. This is a user-defined command that corresponds to `esp3 -dev 1`, that is, it uses COM1 at 9600 baud.
- Go to the ESP3 command line by pressing Escape twice.
- Set the highest baudrate possible: `set baud 115200`
- Load the executable: `load "\javanew.omf"`
- Start the JN system: `Go`

3.1.7 Running JAVA

Currently, the first code in the downloaded JN system that executes consists of 2 ‘test’ screens which display output from modified versions of the National Semiconductor hardware test suite. Press `RETURN` when prompted.

When JN comes up, the startup task presents a simple, old-fashioned, CLI using hierarchical menus. This very simple CLI is mostly used for testing. At the top level of this menu, Java can be started by simply typing `j` in response to the `top>` prompt.

An entire Java command line can be specified, that is, normal Java command line options such as `-v` and `-t` can be specified: `top> java -t`.

3.1.8 Bringing up the WebServer

There are 3 web servers on the system. There is a *very* simple web server built into the `{t}est` option of the JN CLI, there is a very simple web server written in Java, and a “real” web server written in Java. The simple Java web server always responds to requests with the contents of a single file that is specified on the command line. Use the `{s}` command (superdir) in the JN CLI or in response to the Java `java>` prompt to inspect available files.

The simple Java web server can be started via:

```
java> wwwsimp  ./index_tmp.html
```

The `wwwsimp` Java web server uses a native method to determine current register contents from the NS486 board.

The real web server can be started as shown below.

```
java> thttp
```

It is configured to return the file `index.html` when no file is specified by the client. Currently it can also return `.gif`, `jpg`, `.html` and plain text files.

3.1.9 Exercising the Connectix Camera

To obtain an image from the Connectix camera, first run `thttp` on the NS486 as previously described. Then, from a Web browser on a workstation or PC, access one of the NS486 pages on `hazelnut` or `vanilla`. This can be done via an existing link (for instance, see `http://www.cse.ucsc.edu/research/embedded/java.html`), by entering the name of the NS486 machine (`hazelnut` or `vanilla`), or by entering the direct IP address of the NS486 system (128.114.14.16 for `hazelnut` and 128.114.14.15 for `vanilla`).

The web page that is displayed currently contains links to a number of tests, one of which is an applet that controls the camera and displays the acquired image in the browser.

3.2 Downloading Java Code

Java `.class` files can be downloaded to a JN system and placed on a RAM disk. This process allows Java code to be developed and debugged in a traditional Unix or Windows environment, and then moved to a JN system that does not have a network connection.

Note that the files to be downloaded do not have to be `.class` files, although this is the most common use of the download procedure. Any file, including an ASCII `.java` file can be downloaded.

3.2.1 Adding a New File to the RAM Disk

- Compile the Java file to produce a `.class` file. Currently, most such files are stored under directory `/projects/pdebug/new`
- Edit file `class.list` in directory `/projects/pdebug/newCR/needclasses`. This file is a control file for a tar-like utility, `vol_make.dec`, which is used to transfer a set of files to the PCMCIA RAM disk on the NS486 board. The format of the `class.list` file consists of a control section for each directory containing files to be placed on the RAM disk. Each section consists of an `IN_DIR=` command line specifying the directory containing input files, an `OUT_DIR=` command line specifying the directory on the RAM disk in which the files are to be placed, and a list of filenames found in the input directory that are to be copied to the RAM disk. The `vol_make.dec` program is simple and fragile. All lines should be terminated with a RETURN, and blank spaces should not extend after the last filename character.
- Run the program `vol_make.dec` on a DEC Alpha platform such as `gawain` or `lestrade`. This program will create a new `vol.ram` file ready to be downloaded to the NS486.

  The binary is a DEC Alpha binary. The `vol_make` program is sensitive to the ‘endianess’ of the processor on which it runs. Compiling and running this program on a big-endian Sun platform will currently result in a corrupt `vol.ram` file. This is another ‘feature’ of `vol_make` that should be enhanced.

  It may be useful to delete file `vol.ram` before running `vol_make.dec`.
- Now create a new RAM disk image as specified in the following section.

3.2.2 Creating a New RAM Disk

To create a new RAM disk, first create a `vol.ram` file containing the files to be placed on the RAM disk. This procedure is described in the preceding section. This file must then be downloaded to the PCMCIA RAM disk on the NS486 board as follows:

- Use `ftp` to move the `vol.ram` file from Unix directory `/projects/pdebug/newCR/needclasses` to the DOS directory `c:\jn\util` on the PC hosting the NS486. Alternatively, a floppy disk and Unix `mtools` can be used. FTP is used as follows:

```
On the DOS PC:

dos> cd \         <-- Change to the root directory.
dos> nos          <-- Bring up the KA9Q shell.

nos> cd \jn\util  <-- Go to the utility directory. This directory
                       contains the last vol.ram file used.
                       It is important to perform this cd before
                       starting ftp (ftp output will go here).
nos> ftp gawain   <-- Or some other machine in Baskin center.
                      At the username/password, supply your
                      username and password, etc..

ftp> cd /projects/pdebug/newCR/needclasses
ftp> binary
ftp> get vol.ram
ftp> quit
```

```
nos> exit

dos> cd \host
```

- After the new copy of `vol.ram` is in directory `c:\jn\util`, boot JN. This will usually require a new download on JN.
- Do not start Java. Using the JN CLI, you now need to initialize the RAM card, and then download the contents of `vol.ram` to the RAM card. The DOS executable `c:\jn\utils\vol_load` cooperates with the `{v}ol load` option of the `{u}tility` menu of the JN CLI. Although `vol_load` could be used to incrementally copy files to the RAM disk, in practice we have allways been initializing the disk clean and then loading an entire volume image – this simplifies the maintenance of the `class.list` file.
- Use the `{c}onfig` option to initialize the PCMCIA RAM disk. PCMCIA RAM disks are credit card sized boards with an edge connector. They were originally based on Japanese 'debt cards', and look somewhat like thick versions of the Xerox copy cards used at UCSC. The come in many types: RAM disks, moving head disks, Ethernet controllers, modems, etc.. PCMCIA RAM cards and PCMCIA flash cards are not the same. RAM cards contain conventional DRAM. Flash cards contain electrically erasable ROMs that can only be written with special hardware (although this hardware is sometimes built into the motherboard or controller). The RAM disks are convenient because of their size, speed, and because the memory is battery-backed, that is, it does not lose its contents when the power is turned off. PCMCIA RAM cards have a large lithium 'hearing aid' style battery embedded in a corner. A small locking slider can be found on the card. Undoing this slider allows the battery to be removed and replaced. Under normal circumstances, this should only need to be done about once a year.

  On the NS486 board the PCMCIA slot is located under the PC/104 controller. The card is directly under the Ethernet cable on the boards we have. Under normal circumstances, there is no reason to remove the card, however, no problems are caused by doing so. PCMCIA hardware is intended to be 'hot swappable'. JN currently assumes that a single board is simply left in place.

  Initializing the card is effectively a JN file system format. To do this:

  ```
  top> c          <-- Use the {c}onfig submenu.

  config> c       <-- This is the {c}ard init option.
                      It will dismount the disk , initialize it,
                      and remount the new 'clean' disk.

  config> x       <-- Go back to the top level menu.
  ```

- Copy the files in the `vol.ram` DOS file to the JN RAM disk:

  ```
  JN:

  top> u          <-- use the {u}til submenu.

  util> l         <-- This is the "card_{l}oad" option.
                       At this point you will be prompted on the
                       JN VT100 to run VOL_LOAD on the DOS PC.

  If you are still inside ESP3 on the PC, type Control-C or,
  Control-Break. This will return you to the DOS prompt.

  DOS:

  dos> cd \jn\util
  dos> vol_load
  ```

The files will be displayed on the VT100 as they are copied.

3.2.3 The Java CLI

The Java CLI is active when Java has been started and the `java>` prompt is current. Typing **h** will result in the following help display:

```
{h}elp           - prints this menu

ChangePrio [new priority]  (no arg gives current priority)
{j}n_stat        - run jn_stat() (reports mem used, etc.)
{c}hecksum       - run sysCollectChecksum()
{m}emory ck      - run mem_fence_check()
lo{g}_dump       - run log_dmp()
{s}uper_dir      - Dir listing of all files
{l}s <file>      - usage: l <filename>
t{y}pe <file>    - Display file
d{u}mp <file>    - Dump file contents (in hex)
{e}rase <file>   - Delete file
```

These functions all use native methods to call native JN C functions.

3.3 Running the JAVA Test Harness

A number of Java test programs exist on the RAM card. These programs can be run interactively under a test harness.

The java program `Harness` can take an input file specifying which tests to run.

```
java> Harness [<input file>]
```

The input file must contain lines consisting of single characters in the first column, with the last line being `q` (which quits the harness). For example, an input file that runs tests 1 and 2 would look like:

```
1
2
q
```

An input file which runs all tests once then quits is included on the `vol.ram` in the file `jt.a`.

If no input file is given it will print the menu shown below and require user interaction.

```
Enter the number of the area(s) would you like to test.
Enter "a" to run all tests. Include "i" to run sequence infinitely.

        A) All Tests

        1) Strings

        2) Exceptions

        3) Utilities

        4) Threads and Synchronization (currently hangs)

        5) I/O  (not implemented)

        6) Garbage Collection

        I) Run Sequence in Infinite Loop

        Q) Quit
```

Some sections are not yet implemented or are under development. Existing tests should not crash the system.

A bigger test harness called `BigHarness` is in the making. The big harness will have a massive number of tests and is meant to be loaded (into ram disk) only when needed.

3.3.1 Adding Test Programs to the Java Test Harness

The test harness consists of two files:

```
/projects/pdebug/newCR/needclasses/src/harness/Harness.java
/projects/pdebug/newCR/needclasses/src/harness/MenuHandler.java
```

Harness Test Files

Most of the test files used by the harness are in: `/newCR/needclasses/src/harness/jt/`, and are thus in the java package `jt`, which is necessarily imported into `MenuHandler.java`.

Some of the test files are in the main source directory, so that they can be called from the command line as well as from the java prompt without having to specify the package name.

Adding New Test Files to the Harness

To add a new test file to the harness it is necessary to do five things:

- Add the line `‘package jt;’` to the test's source file and recompile it.
- Copy the source and compiled code to the `/projects/pdebug/newCR/needclasses/src/harness/jt/` directory.
- Edit `src/harness/MenuHandler.java` so the new test program is called. Put the program in the correct test group (strings, exceptions, utilities, I/O, threads, or GC) or create a new group if necessary.

  Since the `main()` methods of the test programs are called from another Java program, a `String[]` argument must be hard-coded. Edit the local variable `argStringArray` if your program is expecting command line arguments.

  For example, in the file `MenuHandler.java`:

  ```
  argStringArray[0] = new String("first_arg_string");
  argStringArray[1] = new String("256");
  argStringArray[2] = new String("13.12");

  ClassThatTakesThreeArgs.main(argStringArray);
  ```

  Note: The `argStringArray` is hard-coded to be of a certain size (32 arbitrarily). If the java test program executes `args.length == some_const` to check for argument length, it will always be 32. If the argument length test is crucial to the test program create your own `String[]` of the correct size.
- Edit `needclasses/class.list` to include the new files. Some java programs compile into multiple `.class` files so be sure to include all the files. Find the `IN_DIR` that names the location of the `/projects/.../harness/jt/` files and add the new files to this list.
- Run the program `vol_make.dec` to create the new `vol.ram`. The program will fail on error. Some potential annoying errors: including extra spaces at the end of a file name in the `class.list` file or the user not having read permissions on the source file. Do a `chmod g+r *.class` on all class files so other users can make new `vol.ram` files.

It is a good idea to run the java program alone and from the test harness on Solaris before trying it on JN. Currently the java programs in JN can not “`execute`” any system programs, run any `awt` library calls, and many other functions which the new test file may use. The correct output on the Java system may be different from the correct output on a Solaris host.

3.3.2 Simple Java Test Programs

The following simple Java programs are useful point tests:

```
GCTest <int1> <int2>     <-- Fills up garbage collected heap
                           <int1> specifies the size of a block,
                           <int2> specifies the number of blocks.

CallGC                   <-- calls Garbage Collector - see it go!

Other                    <-- Sanity Check - prints 1 word to stdout.

jt.Game                  <-- Synchronized PingPong game - tests threads.
```

3.3.3 JN Java Utility Programs

ChangePrio

This program changes priority of the main/startup thread. The startup thread starts at Normal Priority (5). All threads created by a thread, including those created by the startup thread, inherit the creating thread's current priority. The format of `ChangePrio` is:

```
ChangePrio [<new priority (1-10)>]
```

SetFlags

The `SetFlags` utility alters global flags that control attributes of either JN or the JVM. These flags are used to control features, such as whether tracing information is displayed. `SetFlags` has the following command line syntax:

```
SetFlags   [options]        <-- specifying no argument results in a menu.

     Options:

           show     shows current value of all flags
    -t      tracing on
    -tx     tracing off
    -tm     Mtracing on
    -tmx    Mtracing off
    -v      verbose on (note: very annoying)
    -vx     verbose off
    -u      UniqueCLI on
    -ux     UniqueCLI off
    -r      RoundRobin Scheduling on
    -rx     RoundRobin Scheduling off
```

The initial flag state is `tracing off` and `RoundRobin Scheduling on`.

Note that with `RoundRobin off` starvation is likely unless all thread control is explicit. Under Solaris, Java runs with `RoundRobin off`, while under Win32, Java runs with `RoundRobin on`. JN can emulate either.

ThreadLister

`ThreadLister` simply shows all live threads at the time of its execution. This is useful to determine if any threads are still running after the main thread of a program exits. Normal `ThreadLister` output on JN looks like:

```
        java> ThreadLister

Thread Group: system  Max Priority: 10
    Thread: Idle thread  Priority: 0 Daemon
    Thread Group: main  Max Priority: 10
        Thread: main  Priority: 5
```

The display indentation is meaningful. Any remaining threads will show up under Thread: main. `ThreadLister` can also be called within a Java application program as follows:

```
ThreadLister.listAllThreads(System.out);
```

3.4 Using the Debugger

The SSI loader/debugger provides most of the conventional debugging capabilities. Several manuals can be found in the Embedded Systems Lab.

To view the current source code through the debugger use a batch file like `bigcopy4` to copy all of the source files to a single directory. After loading (but before entering `go`), specify the directory containing the source code by pulling down the `Config` menu in the ESP3 window and selecting `Source file path`. It will prompt for the full path name of the directory containing the source code.

To view commands or data during execution it is necessary to set a break point before entering `go`. One way is to break at a function call. At the ESP3 prompt type `view source` to display the C source code in the debugger's `Source` window. Alternatively, enter `view mix` to display C source code and assembler. Select the `Source` window by clicking the cursor on the window title bar. Hit ALT-G to get the prompt for the source code you want to view and enter the function name. The code for the function will be displayed in the `Source` window with code addresses in the left column of the display. Click on the address of the line you would like to break at, making sure that it becomes highlighted. Confirm that the breakpoint has been set by pulling down the `Debug` menu and selecting `Breakpoints`. The address of breakpoints will be listed in a window.

When the breakpoint has been reached, continue or step through execution by selecting one of the commands under the `Execute` menu.

3.5 Standard Development Aids (DOS user-commands)

The directory `C:\BAT` is included in the DOS path and contains a number of convenience command files:

```
b         -- Change to the build directory and start pwb.
bigcopy4  -- Copy all .C and .H files in the source tree to \master\big_src
big_src   -- cd to \master\big_src
c         -- cd to the build directory (pwb is _not_ started).
e         -- Link the JN system using \verb+javanew.cmd+ input file.
f         -- cd to \host and run the flashload utility, using the input
             file \verb+flsh.txt+.  To run the loader by hand use:

             dos> cd \host
             dos> flashldr  <-- The flashloader utility will start.

             PRESS the RESET button on the NS486 board.
             (near the green power LED)

             flash_cmd: init  <-- Enter the "init" command.
                              <-- At this point a valid TARGET SYSTEM
                                  ON-LINE message must appear.
             flash_cmd: boot ns96mon  <-- boots the "board" part of ESP3.
             flash_cmd: quit

esp       -- Start ESP3, the SoftProbe loader/debugger using input file
             \verb+espcom,text+. This batch file will set the baud rate and
             load the .omf file. When the prompt finally appears,
             type ‘‘go’’. JN output will appear on VT100 window.
             To run ESP3 by hand use:

             dos> esp      <-- The full-screen interactive loader/debugger
                               utility ESP3 will start. (must be in c:\host)
             ESC
             ESC           <-- To exit the on-screen forms and get
                               to the command line.

             esp> set baud 115200
             esp> load "\javanew.omf"  <-- At this point ESP3 will take
                                           about 3 minutes to download
                                           the javanew file to the board.
             esp> go                   <-- JN output should appear on the VT100
                                           connected to the board.

new_vol   -- cd to \jn\util, and run the vol_load utility. This is
             the PC side of downloading a vol.ram set of files to the
             RAM disk.
tall      -- Touch all .C and .H files in the source tree so as to
               update the last modification time.
toinet    -- cd to the directory contains the TCP/IP source.
tojnsrc   -- cd to the directory containing the JN nanokernel source.
```

These command can all be executed directly at the DOS prompt.

3.6 Integration

Once new features have been added and debugged in a private kit, they must be reintegrated with the master kit. Follow this procedure to reintegrate with the master:

- Assure no one else is updating the master kit (currently on `masterblend`).
- Make a `.zip` of the master (`\master)` directory into backup directory `\zipfiles`.
- Unzip the newly created zip file into another (new or clean) directory tree.
- Copy your new files into the new directory tree. Note that this presumes that you have kept track of which files you have changed. If this is not the case, or possibility of error exists, the DOS `fc` utility can be used to determine which files in the 2 trees (your private tree and the master) differ (use a variant of the `bigcopy` batch file).
- Rebuild **all** object files (that is, do a complete `nmaker clean` followed by an `nmaker`).
- Run low-level JN tests. At the JN CLI prompt: `T` to invoke the test menu, followed by `T` to start saturation tests. This test will run in an infinite loop, testing each JN API. When it wraps back to test 1, reboot.
- Run the Java Test harness. At the JN CLI prompt: `J` to start java. At the `java>` prompt, `Harness 1 2 3 6`.
- If the tests run, rename the new directory `\master`. Rename the old `master.zip` file to `oldmast.zip`.
- Update `\master\doc\log.doc`.

3.7 JavaCam

This section provides an overview in how to bring up the JavaCam files from scratch and get them working. Three steps are necessary: unpacking and compiling the Java files, starting the `ClassLoaderServer`, and bringing up a client program.

3.7.1 Unpacking and Compiling the Java Files

A GNU-ziped tar file of all necessary Java files is located in `/projects/pdebug/.html/src/quickcam.tar.gz`. To uncompress and untar the files, run the following series of commands:

```
sundance > cd [new directory]
sundance > cp /projects/pdebug/.html/src/quickcam.tar.gz .
sundance > zcat quickcam.tar.gz | tar xf -
```

This will create the directory `quickcam` under the current directory, and the `quickcam` directory will contain all the necessary Java files.

To compile the files, set your `CLASSPATH` environment variable to include the new `quickcam` directory and run the `make` utility.

```
sundance > setenv CLASSPATH [new directory]/quickcam:$CLASSPATH
sundance > make
```

You may want to include the new `CLASSPATH` setting in your `.cshrc` file, because it will be required for any of the Java applications to run correctly.

3.7.2 Starting the `ClassLoaderServer`

Once the Java classes are compiled, be sure that the following files are included in the `class.list` file as detailed in section 3.2.1 above:

```
UnsupportedModeException.class
QuickCamTest.class
HexDump.class
QuickCam.class
QuickCamServer.class
QuickCamSocket.class
QuickCamParameters.class
QuickCamApplet.class
QuickCamNetImage.class
ValueBox.class
ImageCanvas.class
UpdateableImageCanvas.class
QuickCamImageWanter.class
TestApplet.class
Camera.class
CameraControl.class
UnknownParameterException.class
TheCamera.class
TheCameraControl.class
CameraControllerServer.class
SimpleClassLoader.class
CameraSecurityManager.class
ClassLoaderServer.class
Diff.class
SimpleClass.class
FloorToken.class
QuickCamInputStream.class
WaitingRoom.class
Queue.class
QueueElement.class
CopyableInputStream.class
```

Create a new `vol.ram` file, load it on to the NS486, start Java and then execute the following command:

```
java > ClassLoaderServer
```

This starts the server that will accept JavaCam servlets (that is, classes which implement the `CameraControl` interface).

3.7.3 Starting a JavaCam client

Currently there are two main clients for JavaCam: `PeriodicDisplay` and a couple variants, and `QuickCamControls`. `PeriodicDisplay` asks for a JavaCam `CameraControl` servlet class on startup, sends that class to the `ClassLoaderServer` and then continuously downloads pictures from the camera. `QuickCamControls` looks exactly like the `QuickCamApplet`, except that it has an extra button used to specify which `CameraControl` servlet to send to the camera.

To bring up either client, at your UNIX prompt type the following commands:

```
sundance > java PeriodicDisplay

OR

sundance > java QuickCamControls
```

If you get a message saying that Java cannot find these classes, your `CLASSPATH` environment variable is probably not set correctly. Check and make sure it includes the directory containing these classes. NOTE TO WINDOWS NT USERS: the `CLASSPATH` variable on NT *must* include the drive letter (i.e. C:) in the path name.

Once started, the `PeriodicDisplay` client will display a file selection window. Select a `CameraControl .class` file and press **OK**. That file will be sent to the NS486 and pictures should start appearing momentarily.

`PeriodicDisplay` has a variant called `TestDisplay`. `TestDisplay` does not display the file selection window. Instead, it simply picks `PeriodicCameraControl.class` and sends it to the camera.

`QuickCamControls` opens a window that looks much the same as the applet. Before requesting a picture, though, you must select a servlet to send. Do this by pressing the **Send a CameraControl class** button, and picking a `CameraControl .class` file from the file selection window. Currently, only the `TheCameraControl.class` servlet speaks the protocol `QuickCamControls` is expecting. Once the servlet is sent, you may request pictures as normal.

3.7.4 Other helpful hints

- If you have a totally black picture, make sure the white and black levels are set correctly. A black level of 50 and a white level of 100 seems to work.
- Michael Allen's thesis has a good explination of most of the internals of JavaCam. Also, check out http://www.cse.ucsc.edu/-research/embeddedsrcjavacam for a quick overview of the pertinent classes involved.
- The native methods in the Java files are bound to the implementations inside of `qcimp.c` in the JN source tree (off the top of Mike's head, the directory is `j/src/ns486/java/quickcam/qcimp.c`).

Chapter 4

System Programming

4.1 Adding a new C test program

The simplest means of adding a simple C test to verify a new system function is to modify file `t_chain.c` in directory `c:\master\j\src\ns486\java\jn`. The `t_chain` program has a very simple test harness that consists of a simple `for` loop. It can easily be modified to call a single routine.

Assure that the `do_test()` routine in file `jn_cli.c` has not commented out the call to `test_main()`. This is sometimes done to save space by not including the test harness. If the test harness is not called, assure that `t_chain` is also included in the `makefile` and in `javanew.cmd`.

4.2 Adding a new C file

We do not use the Sun Java makefile, which is complex and not portable. We use a simple makefile that runs under DOS `nmaker`. This file only performs compilation. Another file, `javanew.cmd` in the build directory, controls the link.

- Modify file `makefile` in directory `c:\master\j\build\ns486\java\java`.
  - Add the object file corresponding to the new C file to the list defined by `OBJ =`.
  - At the end of the makefile, add a rule to compile the new object file. The makefile uses the very simple approach of having a separate rule for each file. These rules have the following format:

    ```
    $(OBJ_DIR)\t_chain.obj:   $(JN_DIR)\t_chain.c
            $(JN_CC)  $(JN_DIR)\t_chain.c
    ```

    individual rules can be customized. The rules differ depending on whether the file is a JN file, a Java file, or a network file.
  - Assembler files (.asm) can be added in a similar manner. JN uses 1 assembler file, `jn_asm.asm`.
- Modify file `javanew.cmd`, in the same directory as the makefile. Add the output object file, which is found in directory `c:\master\obj_g`, to the list at the beginning of this file. Each object file is placed on a unique line in this file, as `elink386` can only handle lines that are 128 characters long.

4.3 Writing a C device-driver

To add a device driver, do the following:

- First, you need to determine what IRQs can be generated by the hardware. In many cases, this will be given to you, that is, the hardware will require a certain IRQ, perhaps as set by a jumper or switch. *Internal devices*, that is, devices actually on the NS486 CPU chip, can be set to selected IRQs under program control as specified by an interrupt controller register map. See pages 69-70 in the NS486 Data Sheet (the CPU Manual). This map must be set up correctly to translate either internal (on chip) device interrupt requests or external IRQs to the IRQss actually seen by the peripheral interrupt controller (PICs). There is both a master and slave PIC. The master PIC controls IRQs 0-7 and the slave IRQs 8-15.

  Currently, file `pic.c` loads the NS486 PIC map. The values placed in the map are defined in file `ns486cfg.h`.

- Once an IRQ has been selected, the file `javanew.cmd` in the build directory needs to be edited so that the Interrupt Descriptor Table (IDT) is built to point the selected interrupt to the assembler interrupt handler.

  First, identify the `gate` section in the file where the `INTnnn_GATE` macros are defined. The interrupt will need a gate. To identify the specific interrupt gate, the IRQ must be added to the base address of the respective PIC. Currently, the master PIC is at 112. For instance, if IRQ 3 is to be used, the resulting interrupt gate will be `INT115_GATE` because 115 = 112 + 3. The name of an assembler interrupt handler must also be selected.

  After the gate has been defined it needs to be added to the IDT table via the `entry =` command. This command contains a gate descriptor for every supported interrupt. For instance, for an interrupt at 113, the line would be `113:INT113_GATE`.

- Write a driver activation routine. This routine is responsible for setting up the hardware device and initializing interrupts. This routine should setup the hardware by initializing any required registers, and enable the PIC to post an interrupt when the device is active at the selected IRQ. This last operation is performed via a call to `PIC_Enable()`.

- Modify routine `initial_driver_activations()` in file `jn.c` to execute the new driver activation routine by invoking `thrd_kcall()`.

- Add an assembler interrupt routine. It is convenient to put all of these in file `jn_asm.asm` in the JN source directory. These routines are simply interface glue. An existing routine can be copied. The name of the C routine called by the handler should be changed.

- Write the C interrupt handler. This routine should manage the device hardware. When it is appropriate to indicate to the PIC that the interrupt at the current IRQ is dismissed, call `PIC_EOI()` with the IRQ.

  The C interrupt handler can queue additional work routines to be processed by the kernel. Such a routine will not run at interrupt level.

  If the C interrupt handler has queued additional kernel work, the C macro `SWITCH_TO_KERNEL` must be invoked to execute the kernel. Executing this macro results in control transferring out of the interrupt routine. If the kernel does not need to be run, the C interrupt handler can simply return.

4.4 Adding a new API (system function)

4.5 Adding a new Native Method

- Write the Java code that contains the call to the native method you wish to define.

- Compile the java code with `javac`.

- Run `javah` against the class file output from the previous compile (`javah <class>`). This will produce a C header file (`<class>.h`) with a typedef for the class and a prototype for the native method.

- Run the `javah` command again with the `-stubs` option (`javah -stubs <class>`). This creates the stub file (`<class>.c`) containing a `_stub` routine that calls the native C routine. Because we are working in a DOS environment, you must edit one line in this file: change `#include <StubPreamble.h>` to `#include <StubPrea.h>`.

- Create the .c file which will contain the function definition. The convention is to use some variant of the class name with the last three letters being `imp`, for implementation.
- Include the files `StubPrea.h` and `<class>.h` in your implementation file.
- Copy the java-compatible prototype for the native function from the generated .h file. Use this in your function definition with the following changes: remove the `extern` keyword; provide parameter names to the parameter types. The first one will be `this` (the object), and the subsequent ones will be whatever names you want for the parameters in your C code.
- Edit `linker_m.c`. Add the name of the `_stub` routine to the table that is found in this file.
- Edit `nativst.h`. Add a prototype for the stub file. Since these are all the same, this can be done at any point before a build.
- Depending on the class and function you are adding, it may be appropriate to create a new directory somewhere in the JN source tree. For example, the `cli` native code is in the `cli` directory.
- Download the one .h file and two .c files and put them in the appropriate directory.
- Modify the `makefile` in the build directory in 2 places to refer to the new .c files (`<class>.c` and `<nativeimp>.c`).
- The files in the make directory with the .arg extension contain the compiler's include directories for different functional groups of code. It may be appropriate to create a new .arg file for the include directories for your code. If so, create a macro in the makefile to reference the new .arg file and use it in the compile command for your files.
- Modify the `javanew.cmd` file for linking the new .obj files.
- Create a new `vol.ram` file containing the Java code calling the native method, and download to the RAM card (see section 3.2).

4.6 Using MiniEdit

The Java program `./MiniEdit` is a 'toy' emacs-style editor written in Java that runs on the VT100. It can be used to edit files on the PCMCIA RAM disk. To invoke it on file `foo`:

```
java> ./MiniEdit foo
```

Files with a name of starting with `card:` will be found or created on the PCMCIA card.

Chapter 5

The Source

5.1 The Source Tree

```
 \master           - The root of the Sun-like Java source tree.
 \master\big_src  - A copy of all .c and .h file sources in the tree.
 \master\docs     - Documentation.
 \master\obj_g    - Object files resulting from compilation are placed here.
 \master\javanew  - The root of the Sun-defined source tree.

 \master\j\build\ns486\java\java         <-- The build directory.
                                                Contains the makefile.

 \master\j\build\ns486\java\java\cclass  <-- .h files are in these dirs.
                             ...   \cclass\java\io
                             ...   \cclass\java\lang
                             ...   \cclass\java\util
                             ...   \cclass\suntools\debug

 \master\j\ns486\include      <-- Default JN .h files for JN #include <>.
 \master\j\ns486\include\sys  <-- Default JN system .h files.

 \master\j\src\ns486\java\include
                        ... \inet    <-- The TCP stack source.
                        ... \javai   <-- The UCSC Java startup code.
                        ... \jn      <-- The JN nanokernel and drivers.
                        ... \runtime <-- The Java system specific runtime: I/O, signals.
                        ... \ucsc    <-- Some native method stubs.

 \master\j\src\share\java\cli     <-- The UCSC Java CLI.
                        ... \include
                        ... \lang    <-- The compiler, runtime, and common threading.
                        ... \net     <-- A few network convenience functions.
                        ... \runtime <-- The interpreter, class initializer, GC.
                        ... \util    <-- Built in utilities (zip).

 \jn\util  - Contains the vol_load utility used to
             perform the DOS side of a RAM card init.

Other directories of interest:
 \bat      - Contains user-defined DOS commands.
 \highc    - Contains the Metaware HIGHC compiler
 \ll386eva - Contains the SSI tools, that is, the loader/debugger.
 \masm611  - Contains the Microsoft assembler and PWB.
 \nos      - Contains the KA9Q executable.
```

5.2 Searching Source Files

The most convenient way to search all source files is to use the `fg` utility after changing to the `\master\big_src` directory. This directory contains a copy of all source .c and .h files. To update the contents of the `big_src` directory, use the user-defined DOS command `big_copy`.

The `fg` utility (fast grep) can be used similarly to Unix `grep`. The command `fg foobar *` will search all files for *foobar*.

The `big_src` directory and `big_copy` were not defined primarily for `fg`. The SSI ESP3 debugger supports source debugging on the target board, but all source must be located in a single directory.

Chapter 6

Backup Procedures

6.1 Backups

6.1.1 To Make a Backup

```
dos> rem --- Remove all files from "\zipfiles".
dos> pkzip -ex -r -p \zipfiles\ucsc.zip \master\*.*
dos> pkzip -ex -r -p \zipfiles\jn.zip   \jn\*.*
dos> backup \zipfiles a: /s
```

6.1.2 To Restore a Backup

```
dos> rem -- Make a temporary directory, cd to it...
dos> rem -- Restore all zip files:
dos> restore a: c:*.zip /s
dos> pkunzip -d \zipfiles\master.zip
dos> pkunzip -d \zipfiles\jn.zip
```

Chapter 7

Networking

7.1 JN Networking

7.1.1 Testing Network Connectivity

On a DOS machine running KA9Q, enter *trace* mode:

```
 dos> cd \
 dos> nos

 nos> trace pk0 0211
```

In this mode the DOS PC will act as a *datascope*, that is, all network packets that it processes will be displayed on the screen and formated. Now bring up JN and `ping` the DOS machine by using the ping utility built into the JN CLI:

```
top>  u
util> n
net>  p 128.114.14.12
```

If the ping does not work, the prompt will simply appear in a few seconds. If the ping does work, a `rtt xx Msec` message will appear, indicating the number of Milliseconds required for the round trip to the designated machine.

7.1.2 Changing Network Configuration

The JN TCP/IP network configuration is managed by routine `net_init()` in file `inetmain.c` in the JN networking directory, `c:\master\j\src\ns486\java\inet`.

Currently, there are 7 #define's and 3 `char *` definitions needed to supply all the network information required by a JN TCP/IP node. There are a number of such definitions present at the head of file `inetmain.c`. Each such section is enclosed in a `#if 0` conditional. One of these sections should be included in the compilation be changing the conditional to a `#if 1`. To support a new machine, or to locate an existing machine on a new subnet, copy one of the sections and customize it.

In addition to the required header, there are sections at the bottom of routine `net_init()` that predefine an `arp` table which provides IP address to Ethernet address translation for the local subnet. In many cases `arp` will automatically manage this table, but it can be useful to predefine it by hand, if known.

The information in the `net_init()` should be obtained from a file on the RAM card that could easily be edited.

For additional information on network setup, see also the section **Setting up the KA9Q NOS**.

7.1.3 Current Subnet Information

```
Maxwell      --  128.114.14.12  <-- The secondary development DOS PC.
Masterblend  --  128.114.14.13  <-- The primary development DOS PC.
Peet         --  128.114.14.14  <-- Pak Chan’s old 386.
Vanilla      --  128.114.14.15  <-- The primary NS486 development system.
Hazelnut     --  128.114.14.16  <-- The secondary NS486 development system.

Madrone      --  128.114.24.58  <-- The SUN SLC workstation in 320.
```

7.2 DOS Networking

Internet services are provided on the DOS PC's by `KA9Q`. This TCP/IP package is a single C program. It is public-domain (for non-commercial purposes). `KA9Q` does not run transparently on the PC's. Rather, think of it as a special DOS shell that contains the basic TCP/IP utilities (ping, telnet, ftp), and which allows user written TCP/IP applications to be run from within it.

The JN TCP/IP stack is based on a customized subset of `KA9Q`.

KA9Q will communicate with most PC Ethernet boards via the so-called `packet driver`. This is a standard DOS Ethernet driver that hardware vendors supply (usually for free). It provides a common interface to packages such as `KA9Q`. A suitable packet driver must be installed (usually in `c:\autoexec.bat` or `c:\config.sys`) before KA9Q can run.

The `KA9Q` package consists of a single executable file, `c:\nos\nos11a.exe`. Additional information describing NOS can be found in text file `c:\nos\ka9qdocs.txt`.

The user of the DOS machine does not need to worry about usernames or passwords. However, one can protect the DOS PC from external access by specifying usernames and passwords for external users via the file `c:\nos\ftpuser`.

7.2.1 Using the Net from DOS

To access the Internet from the DOS development PC's:

```
dos> cd \
dos> nos

nos>
```

The batch file in the root directory is `c:\nos.bat`. Running this results in a change to the 'NOS shell'. When running the NOS shell the PC is on the Internet. You can execute TCP/IP commands, such as `ping`, `ftp`, and `telnet`, and other machines (which know the machine's password) can access the PC.

Obtain help via the `?` command. Additional help can be obtained by typing the name of a command followed by `?`. When you exit NOS the PC is no longer on the network. Starting NOS takes only seconds. Exit NOS via `nos> exit`, which is immediate.

Useful NOS commands are `ping <machine_name>`, `ftp <machine_name>`, and `telnet <machine_name>`. Before downloading files to the PC via `ftp`, it is useful to do a `cd` under NOS to the target directory. If this is not done, files obtained via `ftp get` will be placed in the `c:\nos` directory.

`KA9Q` can maintain multiple full-screen sessions, that is, it can simultaneously support a number of on-going `telnet` and `ftp` sessions. The F8 key can be used to switch between sessions.

A useful test of network connectivity can be performed by `ping` from NOS with the following format:

```
 nos> ping arapaho  1 5000
```

This will send a 1 byte test message every 5 seconds.

7.2.2 Setting up the KA9Q NOS

All `KA9Q` files are located in directory `c:\nos`. A startup file, `c:\autoexec.nos`, specifies all options that can readily be altered. This may be required if the kit is moved to a new machine, or an existing machine moved to a new subnet.

The options that usually make sense to change are 'underlined' with caret characters on the line below each line in which they occur.

To get `KA9Q` to work on the Internet, it is typically only necessary to specify:

- The machine's IP address (assigned by whoever is in charge of the subnet to which the machine is attached).

- The subnet mask. This also is supplied by whoever is in charge of the subnet.
- The IP address of the Primary Domain Name server, and the IP address of the Secondary Domain Name server. These will sometimes be the same IP address. This is the address of a server machine that contains a database used to translate between mnemonic network names and Internet 4-dot notation names.
- The IP address of the Default Gateway, a.k.a., the Domain Server. This is the address of a machine on the subnet to which all packets, destined for somewhere beyond the subnet, are sent.

Although the above items are usually sufficient to 'get on the net', it may also be helpful to edit the `arp` commands at the end of the `autoexec.nos` file. These commands can be used to pre-specify, by hand, a small table on the PC containing the Ethernet to IP address mapping of the machines on the local subnet. This can be useful when you do not know what protocols are supported on the subnet.

7.3 Using the Sniffer

The department has a Network General Distributed Sniffer System (DSS). The DSS Server is a PC attached to the `14` subnet with 2 Ethernet connections: one for capturing traffic, and one for sending data to the Console program. The user opens a Console to the Server, as described in the next section.

The most appropriate DSS application for network debugging may be the Analyzer. In addition to capturing frames and interpreting headers, the Analyzer watches for certain predefined conditions in the network traffic, such as slow response time, misdirected packets, or network overload. The default values for the trigger events are reasonable, but may be redefined by the user. When the Analyzer detects certain events, it displays the Symptoms and possibly a Diagnosis. The Analyzer allows you to display sequences of frames over time, or to look at individual frames. All headers, as well as packet data, from each frame may be displayed as hex values, ASCII, or interpreted according to the protocol level.

7.3.1 Running the Analyzer

The Sniffmaster for X (smx) Console must be run on `sundance` (where the license manager is running), and is started by running `/usr/local/smx/bin/smx xsniffmaster`. The user must be added to group `ngc` by a system administrator in order to use this software. The next window to come up will prompt for the name of the Server to connect to (currently `hound1`). If the connection is successful, a Console window will come up with the DSS Main Menu displayed. Use the arrow keys to highlight `Analyzer` and hit enter.

Problems with the Server

The Console may be unable to connect to the Server because of Server problems or network problems. If there is a networking problem between the Server and the network, or the Console and the Server, a system administrator may need to check the cables and the Server Ethernet card connections. If a Console connection had already been established, but the Server encountered a problem during execution, a remote reboot may be required (Console window may ask you to reboot the server). This can be done by pulling down the X window System menu and selecting Remote Reboot. Wait 5 minutes before trying to establish a new connection by selecting System - New Connection from the window menu.

Fonts

Although the `smx` program is supposed to configure itself and set the proper font path, so far it has not worked correctly. You may have to copy the entire directory `/usr/local/smx/X11/R5fonts` to your local machine (to the machine where your X server is running) and run `xset fp+ <full-name-of-new-font-dir>` to include the font path necessary for the display.

Console-Server Interface

The Console interface to the Server is through a VT100 terminal-style display in the Sniffmaster window. The interface is a hierarchical DOS program and you will navigate through the menus using the arrow and enter keys, PgUp, PgDn, Home, etc. Most actions have a short help sentence which appears near the bottom the screen when they are selected. Different menu items require different keys sequences to activate or select them. Use the arrow keys to highlight different items on the menu. If there are more options or parameters to the item, they will appear in a new column to the right of the selected item. Use the arrow key again to move the cursor to those items, which may also have additional parameters that appear to the right in a new column. Toggled items are selected or deselected with the space bar (check-mark means selected, x-mark means deselected). Highlighting and hitting enter over some items results in a pop-up list of possible values, or a prompt for a new value.

When the Console is in a certain state, numbers and a brief description appear in shaded boxes at the bottom of the window. These refer to the Function keys used to select them. The Function keys may change the state of the capture, or result in a different data display. Some Function keys may toggle between different displays, others require the `ESC` key to return to the preceding state. Although the Server interface is not sophisticated or intuitive, with practice and trial and error, navigating through the menus and settings can become an acquired taste.

Getting to the Sniffmaster Main Menu

If you are not at the main menu (yellow menu text in the upper corner will say `Main Selection Menu - Release 4.0`) and you want to be, but don't know how to get there from the current menu: select `Exit` or `Return to Main Menu` from your current menu, if available. You may have to exit from multiple levels to get back to the main menu. If you find yourself at a DOS prompt, go to the root directory and type `menu` to bring up the main menu.

Managing Host Names

When running the Analyzer, you may want the sniffer to always display a host by alphabetical name instead of by an address. To add host names to the Analyzer name data base, from the Analyzer menu select `Display -> Manage names -> Edit names` and hit enter. The list of all station names will be displayed. Select the type of address by which you will identify the station. For example, to identify a station by IP address, select the row for a new station at the `IP` level. Be sure the `Save names` menu item is selected (check-mark next to it) so that the new names will be saved when you exit the Analyzer.

Capturing Frames

From the Analyzer menu you will see a red box with `10 New Capture` in it. Hit the `F10` key to start a new capture. The `Expert Overview` window will immediately begin showing the traffic statistics in a table, dividing the traffic among network layers, Object/Symptoms, and Diagnoses. Select a row which has values in it and hit enter to see more detail. The `F2` key toggles between the `Expert Overview` and `Global Statistics` window. The `Global Statistics` window shows a breakdown of the traffic by protocol family, the average, current and maximum bandwidth and frames/second, and also a breakdown of TCP/IP traffic. The bar along the bottom shows the current frames per second (solid line) and the maximum frames per second achieved (dotted line).

If a diagnoses appears, move to that column and hit enter multiple times for various details of the diagnoses. The frames can not be viewed until the capture is stopped or suspended. After stopping (`F10`) or suspending (`F9`) the capture, hit `F3` to load the data through the display filter. Depending on the display filter values, frame summary or details will be displayed.

Display Filters

It may be useful to redefine the filter for the displayed packets. From the main Analyzer menu, select `Display -> Filters` to view the filter options in the column to the right of the `Filters` item. The `Filters` item itself can be toggled on/off, so make sure there is a check mark next to it if you want to apply the display filter. Continue moving right with the arrow keys and select the type of filter you would like to define or modify. It is recommended to browse through all the filter items when initially displaying data in order to be aware of the type of filtering being done.

As an example, to display only those packets exchanged between two stations, select the `Station Address` item and continue to the right to `Match 1`. Select the `From` and `To` stations by hitting enter on these key words, and then selecting the stations from the list that comes up. Set the desired values for the options below the `From` and `To` items as well. Make sure that `Match 1` is selected and the other `Match` definitions are deselected if you want to apply only one address match. When you display the data again, or return from the `Display Options` menu, the new display will include this filtering.

Saving/Loading Setup Files

The display filters and other capture parameters can be saved and reloaded. One setup has been defined and saved as an example. It specifies that frame summary, detail (interpretation), and hex values be shown for all network layers. To load this file, from the main Analyzer menu go to `Files -> Load -> Setups` and select the `SETUP3.ENS` file. Display configurations can be saved similarly by selecting `Files -> Save -> Setups`.

Saving Data to a Text File

After you have run and then stopped the capture, you can save the frame to a CSV (comma separated value) ASCII file. The values saved are not the frame data but rather the frame statistics, such as absolute and relative arrival times, total size in bytes, bandwidth used, and header summary. From the Analyzer menu select `Display -> Print` and continue right to the `Print` options. Use the space bar to select the first and last frame numbers you want to save. Select `File` for the output and `CSV` for the format. After setting the options, return to the `Print` command and press enter to save the data to a file. You will be asked to name the file. Data in this form can be most readily translated into graphs or compiled into statistics. Refer to the File Transfer Utility Section (Section 7.3.2) for information about downloading your file from the Server.

The frame contents can also be saved as plain text. Select `Plain Text Format` instead of `CSV` in the `Print` options menu. The plain text format saves all information about every frame, including header interpretation and data, and will create a large file if you have captured many frames. It may be more practical to view individual frames through the Console instead of saving the data in a file.

7.3.2 Other Useful Sniffer Utilities

File Transfer Utility

Captured frame data may be saved to a comma separated value (CSV) file or to a plain text file. Data in CSV format can be easily parsed and compiled into a form useful for plotting statistics. Select `File Transfer Utility` from the main menu to put the Server in file transfer mode. Once the Server has entered this mode, you may copy files to and from the Server. When the Server has confirmed that file transfer mode has been entered, select (with the mouse) the File Transfer item under the System menu of the X-window menu. Select the activity you want to perform (i.e., Copy File) and follow the directions of the subsequent menus.

Exit to the Operating System

Select the item `Exit to the Operating System` to get to the Server's DOS prompt. Several DOS utility programs (such as `dir`, `copy`, `del`) are on the Server, although the system was not intended to be used at the DOS level. However, accessing the Server through DOS may be helpful when locating or deleting files. To get back to the Main Menu from the DOS prompt, return to the root directory and enter `menu`.