The NPS Small Robotic Technology Initiative, man-portable robots for low intensity conflict

Ferry, Todd W.

Monterey, California : Naval Postgraduate School

http://hdl.handle.net/10945/34057

Downloaded from NPS Archive: Calhoun
THE NPS SMALL ROBOTIC TECHNOLOGY INITIATIVE, MAN-PORTABLE ROBOTS FOR LOW INTENSITY CONFLICT

by

Todd W. Ferry

June 2001

Thesis Advisor: Richard M. Harkins
Second Reader: Thomas J. Hofler

Approved for public release; Distribution is unlimited
REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
June 2001

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE: Title (Mix case letters)
The NPS Small Robotic Technology Initiative, Man-Portable Robots for Low-intensity Conflict

5. FUNDING NUMBERS

6. AUTHOR(S)
Ferry, Todd W.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE: A

13. ABSTRACT (maximum 200 words)
The Naval Postgraduate School's Small Robotic Technology (SMART) Initiative is an ongoing research effort within the Combat Systems Science and Technology Curriculum that engages in forward-looking applications of small robotic technology for military employment. The immediate goal of which is to develop a multipurpose robotic platform that is capable of hosting varied sensor packages for military research. This thesis successfully accomplished initial background research and integration of a low cost, lightweight, all-terrain, robotic vehicle to fulfill this requirement. The areas of robotic investigation included: research and procurement of a Foster Miller Lemming tracked vehicle; the selection of a robust, network enabled, real-time microcontroller called the ipEngine; selection of Differential GPS as a highly accurate autonomous vehicle positioning technique; and the development of the ipEngine software environment for integration and testing of the microcontroller’s wireless interfacing. Wireless communication tests using TCP/IP sockets, serial communication, telnet and a common Internet Web Browser validated the ability to remotely operate the vehicle under both direct and autonomous control. Ultimately, this thesis laid the foundation for follow-on NPS students to research and integrate varied robotic sensing techniques, including synthetic array seismic sonar’s and chemical detection devices, and to participate in cooperative research with other military laboratories.

14. SUBJECT TERMS

15. NUMBER OF PAGES
182

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

NSN 7540-01-280-5500

Form Approved OMB No. 0704-0188

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
THE NPS SMALL ROBOTIC TECHNOLOGY INITIATIVE, MAN-PORTABLE ROBOTS FOR LOW INTENSITY CONFLICT

Todd W. Ferry
Captain, United States Marine Corps
B.S., United States Naval Academy, 1992

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2001

Author: Todd W. Ferry

Approved by: Richard M. Harkins, Thesis Advisor

Thomas J. Hofler, Second Reader
William B. Maier II, Chairman Physics Department
ABSTRACT

The Naval Postgraduate School’s Small Robotic Technology (SMART) Initiative is an ongoing research effort within the Combat Systems Science and Technology Curriculum that engages in forward-looking applications of small robotic technology for military employment. The immediate goal of which is to develop a multipurpose robotic platform that is capable of hosting varied sensor packages for military research. This thesis successfully accomplished initial background research and integration of a low cost, lightweight, all-terrain, robotic vehicle to fulfill this requirement. The areas of robotic investigation included: research and procurement of a Foster Miller Lemming tracked vehicle; the selection of a robust, network enabled, real-time microcontroller called the ipEngine; selection of Differential GPS as a highly accurate autonomous vehicle positioning technique; and the development of the ipEngine software environment for integration and testing of the microcontroller’s wireless interfacing. Wireless communication tests using TCP/IP sockets, serial communication, telnet and a common Internet Web Browser validated the ability to remotely operate the vehicle under both direct and autonomous control. Ultimately, this thesis laid the foundation for follow-on NPS students to research and integrate varied robotic sensing techniques, including synthetic array seismic sonar’s and chemical detection devices, and to participate in cooperative research with other military laboratories.
# TABLE OF CONTENTS

**LIST OF FIGURES** .............................................................................................................. XI

**LIST OF TABLES** ................................................................................................................. XIII

I. SMALL ROBOTIC TECHNOLOGY OVERVIEW ........................................................................ 1
   A. BACKGROUND .............................................................................................................. 1
      1. Force Multiplication ............................................................................................... 2
      2. Casualty Reduction ................................................................................................. 2
   B. MOTIVATION FOR THE NPS SMALL ROBOTIC TECHNOLOGY PROJECT ................................................................. 3

II. THE SMART VISION .................................................................................................................. 5
   A. COMBAT SYSTEMS CURRICULUM ........................................................................... 5
      1. SMART Platform Requirements Overview ............................................................. 5
         a. Cost Considerations .............................................................................................. 5
         b. All-Terrain Mobility ............................................................................................. 5
         c. Lightweight/Man-Portable ................................................................................... 6
         d. Modular Adaptability to the Department's research requirements .................. 6
         e. Navigation .............................................................................................................. 6
         f. Autonomous, Semi-Autonomous, and Direct Control ...................................... 6
   B. CURRENT SMART RESEARCH ....................................................................................... 7
      2. Coastal Systems Station (CSS), Panama City Florida ............................................ 8
   C. MAN-PORTABLE, TACTICAL ROBOT MANUFACTURES ............................................. 8
      1. Foster-Miller - Tactical Adaptable Robot ................................................................. 8
      2. Mesa Associates - Matilda ...................................................................................... 9
      3. iRobot - Urban Robot ............................................................................................ 10

III. SMART THESIS OBJECTIVES .............................................................................................. 13
   A. THESIS GOALS ........................................................................................................... 13
      1. Selection of a NPS Robotic Platform ....................................................................... 13
      2. Selection of an Embedded Microcontroller ......................................................... 13
      3. Selection of a Navigation Methodology ................................................................. 14
      4. Integrating an Operable Platform for Future NPS Research ............................... 14
      5. Engage in Cooperative Research with current Military Robotic Labs and Programs ................................................................. 15

IV. NPS SMART ROBOT .............................................................................................................. 17
   A. OVERALL DESIGN ...................................................................................................... 17
   B. ROBOT CHASSIS ......................................................................................................... 18
   C. MICROPROCESSOR – IP ENGINE ............................................................................. 20
      1. Capabilities ............................................................................................................. 20

vii
a. Network Integration ........................................................... 20
b. Real Time Operating Kernel ........................................... 20
c. C programmable ............................................................... 21
d. Virtual Interface ............................................................... 21
e. 66 MIPS, Motorola Power PC MPC823 CPU .................. 21

2. FLASH Memory – 256 K .................................................. 23
3. DRAM Memory – 16 MB .................................................... 24
4. ipEngine I/O Ports ............................................................. 25
   a. Dual RS-232 ................................................................. 25
   b. LCD/Video Controller ................................................... 25
   c. 10 BaseT Ethernet ......................................................... 25
d. USB port ......................................................................... 27

5. FPGA (Field Programmable Gate Array) “Virtual Interface” ... 27
   a. FPGA Background ......................................................... 27
   b. Altera Flex 6016 Programmable Logic Device ............... 28

6. Power Requirements ......................................................... 31
7. ipEngine Internal Arrangement ........................................ 31
8. IP Engine SMART Hardware Integration .......................... 32

D. SMART SOFTWARE .......................................................... 35
1. Real Time Operating System ........................................... 35
   a. Background .................................................................. 36
   b. pKernel ........................................................................ 36
2. Portable Operating System Interface (POSIX) Threads .......... 38
   a. POSIX Background ....................................................... 38
   b. POSIX Threads ............................................................ 38
3. Software Component Locations .......................................... 39
   a. ipEngine Directory ....................................................... 40
   b. pKernel Directory ......................................................... 40
   c. GNU Tools Directory ..................................................... 40
4. Essential Software Interfacing Tools ................................... 41
   a. PumpKIN FTP Server .................................................... 41
   b. Command Shell ........................................................... 42
   c. Terminal Emulator ........................................................ 43

5. GNU Software ................................................................. 44
   a. GNU CC .................................................................... 45
   b. GNU Make .................................................................. 45

E. SMART NAVIGATION SYSTEM .......................................... 46

F. COMMUNICATION SYSTEM ............................................... 49
1. Wireless Ethernet Radio Link ........................................... 49

G. ROBOT CONTROL ............................................................. 50

V. SMART INTEGRATION .......................................................... 53
A. EXPERIMENTATION .......................................................... 53
1. Goals .............................................................................. 53
2. Experimental Setup ......................................................... 53
B. SMART CODING ............................................................... 54
1. Writing C code ................................................................. 54
2. Compiling..................................................................... 54
   a. GNU Make.................................................................. 54
   b. Sample SMART MakeFile........................................... 55
C. SMART SOFTWARE INTERFACING .............................................. 57
   1. ipEngine Network Setup.............................................. 57
   2. Building the Program from the MS-DOS Command Shell .... 58
   3. Loading the Program.................................................. 60
      a. PumpKIN FTP Setup.............................................. 60
      b. Tera Term Pro Setup.............................................. 62
      c. FTP a Program to the ipEngine................................ 63
D. SMART I/O EXPERIMENTATION............................................. 65
   1. Sockets Test .............................................................. 66
   2. Serial I/O Test ........................................................... 66
   3. Web Server ............................................................... 67
E. SMART CONTROL INTERFACE PROGRAM.............................. 67
VI. CONCLUSIONS /SMART FUTURE .............................................. 73
APPENDIX A. IP ENGINE DIAGRAMS............................................ 77
   A. MEMORY LOCATION..................................................... 77
   B. POWER CONFIGURATIONS........................................... 77
   C. IPENGINE SPECIFICATIONS......................................... 79
   D. SOFTWARE DIRECTORIES............................................. 89
APPENDIX B. GLOBAL POSITIONING SYSTEM............................... 91
   A. GEOMETRIC VISUALIZATION:......................................... 91
   B. SATELLITE TRANSMISSIONS:......................................... 92
   C. PSEUDO-RANDOM CODES.............................................. 93
   D. MEASURING DISTANCE:............................................... 94
   E. ACCURATE TIME MEASUREMENT..................................... 94
   F. EXTRA MEASUREMENT TO CORRECT TIMING OFFSET.......... 95
   G. GPS SATELLITE ARRANGEMENT..................................... 96
   H. OTHER SOURCES OF GPS ERROR.................................... 97
      1. Atmospheric Signal Delays........................................ 97
      2. Multipath error...................................................... 98
      3. Geometric Errors................................................... 98
   I. DIFFERENTIAL GPS (DGPS).............................................. 99
      1. Eliminating Common Errors....................................... 100
      2. DGPS Error Transmissions......................................... 101
APPENDIX C. MAKEFILES.......................................................... 103
   A. MAKEFILE STRUCTURE............................................... 103
   B. SIMPLE MAKEFILE EXAMPLE........................................ 105
   C. HOW GNU MAKE PROCESSES THIS MAKEFILE............... 106
   D. SIMPLIFYING MAKEFILES USING VARIABLES.................... 107
   E. SIMPLIFYING MAKEFILES BY IMPLICIT RULES.................. 108
   F. NAMING MAKEFILES................................................... 109
ix
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Intensity Spectrum of Conflict</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>USMC Warfighting Lab’s Robotic Systems Web Page. [From Ref. 1]</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Foster Miller Tactical Adaptable Robot (TAR). [From Ref. 3]</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Mesa Associates Matilda Robot. [From Ref. 6]</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>iRobot’s Urban Robot. [From Ref. 7]</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>NPS SMART All-Terrain Robot</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>SMART Vehicle Side Dimensions (± 0.25 in.)</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>SMART Platform Interior Dimensions (± 0.25 in.)</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>ipEngine with Protective Case</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>ipEngine Mounted on Chassis Board</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Relative Sizes of a Standard Gate Array and a FLEX 6000. [From Ref. 12]</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Altera Max +plus II Developmental Software for the Flex 6016</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Altera’s Flex 6016 Mounted on the ipEngine. [From Ref. 8]</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>ipEngine Architecture. [From Ref. 8]</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>SMART Compartment for Housing On-board Processing</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Microcontroller-Centered Robot</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>Proposed I/O Connections on Robot</td>
</tr>
<tr>
<td>Figure 4.13</td>
<td>pKernel Operation System Breakdown. [From Ref. 9]</td>
</tr>
<tr>
<td>Figure 4.14</td>
<td>PumpKIN FTP Application Command Window</td>
</tr>
<tr>
<td>Figure 4.15</td>
<td>MS-DOS Command Shell Window</td>
</tr>
<tr>
<td>Figure 4.16</td>
<td>Terminal Emulation Command Window (Running pKernel Shell)</td>
</tr>
<tr>
<td>Figure 4.17</td>
<td>Differential GPS Signals. [From Ref. 18]</td>
</tr>
<tr>
<td>Figure 4.18</td>
<td>Proxim Radio Link</td>
</tr>
<tr>
<td>Figure 4.19</td>
<td>Sample Operator Control Interface. [From Ref. 15]</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Test Connections from Computer to ipEngine</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Makefile for Hello World Program</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Hello.c</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Directory Tree for Compiling Hello.c</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>setu.bat</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Building hello.bin from MS-DOS Command Window</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>yes.bat</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>go.bat</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>PumpKIN FTP Command Windows</td>
</tr>
<tr>
<td>Figure 5.10</td>
<td>PumpKIN FTP Options Window</td>
</tr>
<tr>
<td>Figure 5.11</td>
<td>Tera Term Connection to Serial Port 1</td>
</tr>
<tr>
<td>Figure 5.12</td>
<td>Tera Term Serial Port Setup</td>
</tr>
<tr>
<td>Figure 5.13</td>
<td>Commands for Loading hello.bin into the ipEngine</td>
</tr>
<tr>
<td>Figure 5.14</td>
<td>Pumpkin FTP Response for Successful Program Loading</td>
</tr>
<tr>
<td>Figure 5.15</td>
<td>Host Computer Screen With All Three Applications Open</td>
</tr>
<tr>
<td>Figure 5.16</td>
<td>I/O Interfacing With the ipEngine</td>
</tr>
<tr>
<td>Figure 5.17</td>
<td>Original Philips 80C552 Microcontroller and Robotic Circuitry</td>
</tr>
</tbody>
</table>
Figure 5.18 SMART Control Remote Interface Modeled in “Smartcontrol.c.”

Figure 5.19 SMART Control Mission Simulation.

Figure 6.1 Synthetic Aperture Seismic Sonar Array Concept. [From Ref. 21]

Figure 6.2 Conceptual Autonomous Robotic Mine Reconnaissance Mission. [From Ref. 21]

Figure A.1 ipEngine Mechanical Specifications. [From Ref. 13]

Figure A.2 Pin Connections for Port A and Port D. [From Ref. 13]

Figure A.3 Pin Connections for Port C and Port D. [From Ref. 13]

Figure A.4 Connector Pin-outs for inEngine. [From Ref. 13]

Figure A.5 J1 Connector (I/O Port) for the ipEngine. [From Ref. 13]

Figure A.6 J2 Connector (Debug Port) for the ipEngine. [From Ref. 13]

Figure A.7 J3 Connector (DC Power) for the ipEngine. [From Ref. 13]

Figure A.8 J10 & J11 Connector (Virtual Interface) for the ipEngine. [From Ref. 13]

Figure A.9 Virtual Interface Pin Functions for the ipEngine. [From Ref. 13]

Figure A.10 Virtual Interface Pin Functions for the ipEngine. [From Ref. 13]

Figure A.11 JP1 Connector to Enable/Disable of on-board Power Supply. [From Ref. 13]

Figure B.1 Geometric Intersection of Four Satellite Range Rings. [From Ref. 18]

Figure B.2 Four Satellite Signal Measurements allow for positioning in X, Y, Z, and T. [From Ref. 18]

Figure C.1 Directory Tree for Compiling Hello.c

Figure D.1 Connection Request from Client to Server. [From Ref. 24]

Figure D.2 Connection Accepted by Server. [From Ref. 24]

Figure F.1 Auto Boot Run Sequence

Figure F.2 Tera Term Telnet Connection (Port 23)

Figure F.3 Tera Term Telnet Command Window (pKernel Shell Running)

Figure F.4 Commands to Create a Telnet Session from MS-DOS Window

Figure F.5 Telnet Application Window, Running pKernel Command Shell

Figure G.1 Information Routing for Sockets Test

Figure G.2 Socket Test Command Window (Port 2000)

Figure H.1 Information Routing for Serial Test Program

Figure H.2 Serial Test Robot Command Window (Port 2000)

Figure H.3 Serial Test Tera Term Window (Serial Com 1)

Figure H.4 Tera Term Setup for New Line Recognition

Figure I.1 Information Routing for Web Page Test

Figure I.2 SMART Control Simulated Web Page

Figure I.3 Remaking the Filesys.zip

Figure I.4 List of files in Filesys.zip

Figure J.1 Information Routing for Web Page Test

Figure J.2 TeraTerm Window (Serial Port 1)

Figure J.3 Tera Term Window (Socket Port 2000)

Figure J.4 Tera Term Telnet Window (Socket Port 23)

Figure J.5 SMART Control Page (Socket Port 80)

Figure J.6 Four Panel Display of Each Communication Window
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>User Options for Power Supply. [From Ref. 13]</td>
<td>31</td>
</tr>
<tr>
<td>Table A.1</td>
<td>ipEngine Memory Location. [From Ref. 13]</td>
<td>77</td>
</tr>
<tr>
<td>Table A.2</td>
<td>Power/Operating Condition Requirements. [From Ref. 13]</td>
<td>77</td>
</tr>
<tr>
<td>Table A.3</td>
<td>Engine Connection Configurations. [From Ref. 13]</td>
<td>78</td>
</tr>
<tr>
<td>Table A.4</td>
<td>Directory Location of Software Components. [From Ref. 9]</td>
<td>89</td>
</tr>
<tr>
<td>Table B.1</td>
<td>Summary of GPS/DGPS Source Errors. [From Ref. 17]</td>
<td>99</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

The author would like to extend much praise and thanks to my wife, Sondra, and children, Maggie and Annie, for their extraordinary love. Thanks to Richard Harkins for his willingness to accept this project with enthusiasm and vision. Thanks to the NPS staff members Don Snyder and George Jaksha for their fantastic support of the Physics Department. Special thanks to Robin Laird, of SPAWAR San Diego, for enduring thousands of questions with much wisdom and patience. Additional thanks goes out to Charles Bernstein of Coastal Systems Station Panama City for giving us a robot to play with. My hope is that one day all our effort will save the lives of some of those who are called to duty at the “tip of the spear.” Soli Deo Gloria.
I. SMALL ROBOTIC TECHNOLOGY OVERVIEW

A. BACKGROUND

Future warfare will be heavily influenced by technologies that increase the lethality of the modern battlefield while permitting the removal of the human participant. Tactical autonomous platforms will play a significant role in this effort. Modern military operations span a spectrum of conflict; [Fig. 1] a continuum that, at its polar extremes, encompasses both the benign and the totality of nuclear war. The intensity level may change gradually or suddenly, and it may combine aspects of multiple intensities throughout the operation. Military operations other than war (MOOTW), while on the low end of the intensity scale, are the most probable type of conflict facing the U.S. armed forces in the immediate future. The application of combat power in this environment is somewhat limited and focused on very specific objectives under specific conditions. MOOTW encompasses:

- Low Intensity Conflict (LIC).
- Peacekeeping.
- Peace-making.
- Humanitarian operations.
- Civil disturbances.
- Disaster relief, etc. [Ref. 1]

![Figure 1.1 Intensity Spectrum of Conflict.](image-url)
With the prospect of U.S. forces being thrust into MOOTW situations, development of technologies that address the need of the soldier and Marine in a low intensity conflict needs to be rapidly pursued. Robotic platforms will become an important piece of this technological vision and a necessary capability for our armed forces. Development and use of robotics and robotic sensing systems give the warfighter two great advantages eagerly sought in all levels of conflict: force multiplication and casualty reduction.

1. **Force Multiplication**

Robotic systems soon will be used as a force multiplier to increase the capability of the individual soldier across the spectrum of conflict. Whether the system is a single surveillance and detection device that improves upon the limited abilities of a human soldier or a team of robotic devices that work rapidly to accomplish dangerous tasks such as minesweeping and surveillance, robotic systems will eventually become a significant part of our warfighting inventory. Since robotic technology promises great increases in the tactical capabilities of an individual soldier, the scientific work in this area of research intensifies in importance. The benefits of committing fewer personnel to the deadly environment of combat, yet still maintaining dominance over the battlefield, cannot be overstated.

2. **Casualty Reduction**

Proliferation of lethal and relatively inexpensive military technology throughout the world has only increased the reality that future conflicts will be more costly to both the combatant and non-combatant. Robotic weapons and sensing systems, if applied to the right missions with the correct methodology, offer our forces a greater probability of
casualty reduction and avoidance over current tactical techniques. Some of these extremely hazardous duties include missions such as:

- Tactical and strategic ordnance delivery against hostile targets.
- Detecting and warning combatants of chemical and biological attack.
- Finding and removing unexploded ordnance.
- Sweeping and clearing maneuver areas of mines.

While we certainly will never be able to eliminate death in warfare, the use of robotic devices can reduce the overall risk our forces will experience in combat.

B. MOTIVATION FOR THE NPS SMALL ROBOTIC TECHNOLOGY PROJECT

The motivation for the NPS Small Robotic Technology (SMART) project began during the fall of 2000 during the traditional SE-3015 course taught in the Combat Systems, Science and Technology (CSS&T) Department. The goal of the course is to provide the students an opportunity to develop an integrated technical project and to endure the frustration of an actual laboratory environment, thereby gaining an appreciation for the complexities of real military projects.

The focus of the course was to model a simulated military operation with robotic platforms. Emphasizing a systems engineering problem solving approach, the class divided into teams and built platforms that used autonomous decision making, wireless network communications, and infrared signal detection to “cooperatively engage” a drone robot. The success of the class project stimulated much imaginative discussion about expanding the limited capabilities of the basic robotic systems used by the CSS&T department by incorporating current military and commercial technology into the curriculum.
At the same time, the Marine Corps announced in August of 2000 that it would acquire two robotic systems for conducting surveillance missions in a MOUT (Military Operations in Urban Terrain) environment. These robotic platforms, developed as a part of the Defense Advanced Research Projects Agency's (DARPA) Tactical Mobile Robotics Program, are scheduled to participate in warfare experiments conducted by the Marine Corps Warfighting Laboratory to test the feasibility of using robotics to aid the warfighter in urban environments. [Ref. 2] This announcement, coupled with the SE-3015 robotic laboratory, propelled me to create the NPS SMART project initiative to try and combine the needs of the services with research here at NPS.

Figure 1.2 USMC Warfighting Lab’s Robotic Systems Web Page. [From Ref. 1]
II. THE SMART VISION

A. COMBAT SYSTEMS CURRICULUM

The NPS SMART vision is to create an ongoing research effort within the Combat Systems Science and Technology Curriculum that engages in a forward-looking application of small robotic technology for military employment. The immediate goal of this project is to develop a multipurpose robotic platform that is capable of hosting varied sensor packages for departmental research.

1. SMART Platform Requirements Overview

To achieve the overall vision, a general group of requirements and variables were considered as we focused on obtaining a new robotic vehicle.

   a. Cost Considerations

      Currently, most military robotic programs are still in some form of research and developmental (R&D) process. This, combined with the limited commercial applicability of these systems, drives the price of purchasing manufactured robotic components exceedingly high. With this in mind, the SMART project intends to incorporate both Commercial Off The Shelf (COTS) technology and robotic technology from other military labs to create our projects. This should drive the direct cost to the department to acceptable levels, encourage cooperation with other research organizations, and take advantage of rapid advancements in commercial research and technology applicable to the SMART environment.

   b. All-Terrain Mobility

      The biggest change from the existing NPS robotic vehicles is the desire to replace the older, indoor platforms with all-terrain and all-weather capable SMART
vehicles. In order to expand the operating capabilities to include more realistic military environments, we decided to seek an initial vehicle that would perform acceptably both indoors and out.

c. **Lightweight/Man-Portable**

The preliminary vision of the SMART platform is a man-portable platform that is operated close to soldiers and Marines. This is a current application desired by each of the services and is supported by ongoing laboratory research.

d. **Modular Adaptability to the Department's research requirements**

To be a valuable, multifunctional platform; SMART vehicles need have the capability to carry a wide array of sensor packages developed by the CSS&T department. Therefore, it is desirable to have a robot with enough power and modular space to allow for this flexibility. This vision will require platforms that can be easily configured to test and evaluate many modular research packages.

e. **Navigation**

With the movement toward an all-terrain outdoor operating environment, the need to investigate robust robotic positioning and navigation sensor techniques becomes increasingly important for the SMART platform. Autonomous decision making and advanced sensing techniques require very accurate, sub-meter, positioning methods. This drives the navigation selection toward the incorporation of a Differential Global Positioning System (GPS) for location and navigation calculations.

f. **Autonomous, Semi-Autonomous, and Direct Control**

An additional design characteristic for the SMART platform is to have a mission dependant capability to operate either as an autonomous platform or under the
control of an operator. For both direct and autonomous control, an innovative web-based control environment interface is the initial user interface that we desire to investigate.

B. CURRENT SMART RESEARCH

There are several research organizations that played an important role in the development of the technological architecture and vision of the NPS SMART initiative. This list is in no way inclusive of all the organizations conducting advanced robotic research for military applications, but it provides a framework from which to view current military research and how the Combat Systems, Science and Technology Curriculum can assist that effort.

1. Space and Naval Warfare Systems Command (SPAWAR) Systems Center San Diego - Advanced Systems Division

The overarching mandate of this SPAWAR research division is to "conduct research and development in architectures, sensor data processing, communications, operator machine interfaces and integration for deployable robotic sensor systems." The actual robotic research laboratory within the SPAWAR Advanced Systems Division is called the Adaptive Systems Branch. It serves and partners with industry, academia, and other government agencies to provide "network-integrated robotic solutions for Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance (C4ISR) applications." Much of its research is focused on the robotic development of sensor fusion, communications, and human-machine interface technologies for use in physical security and remote tactical surveillance applications. [Ref. 3] SPAWAR Adaptive Systems Branch's innovative robotic network and sensor applications contributed heavily to the vision and focus of the NPS SMART project.
2. Coastal Systems Station (CSS), Panama City, Florida

CSS Panama City is the U.S. Navy's premier research and development organization focusing on littoral and expeditionary warfare. The Station is the principal repository of the national expertise in these areas that are absolutely critical to the future of Navy and Marine Corps operations. Currently, they are investigating the capabilities of autonomous robots to perform search and detection missions for minefields in the land and surf-zone of amphibious operating area. CSS desires to develop key components for robotic reconnaissance including: accurate positioning and deterministic coverage of an objective area, autonomous control of multiple crawlers, autonomous detection and classification of individual mines, and communications and information sharing techniques for robotic crawlers. [Ref. 4]

C. MAN-PORTABLE, TACTICAL ROBOT MANUFACTURES

At least three manufactures, Foster-Miller, Mesa Associates, and iRobot currently participate in lightweight man-portable military robotic research. Their robotic designs each heavily influenced the direction of the SMART platform initiative. The following are some examples of test platforms each of these companies have developed for military research.

1. Foster-Miller - Tactical Adaptable Robot

Foster-Miller develops light, sturdy, compact robots that can be adapted for ordnance removal, reconnaissance, communication, relays, sensing and security. All the Foster-Miller robots are amphibious, rugged tracked vehicles that can climb stairs, traverse a surf zone, negotiate rock piles, snow and sand, and overcome concertina wire. They can be easily carried, are transportable as "checked baggage" on an airline, and fit
in a standard car trunk. [Ref. 5] The Tactical Adjustable Robot has been a test platform for both SPAWAR and CSS research projects.

![Figure 2.1 Foster Miller Tactical Adaptable Robot (TAR). [From Ref. 3]](image)

2. Mesa Associates - Matilda

The MATILDA Robotic Platform is a rugged, reliable, low cost robotic vehicle that meets numerous requirements for tactical, counter terrorism, explosive ordnance destruction (EOD), and security operations. The lightweight unit is versatile, easy to operate, has multiple uses, and is capable of carrying many different payloads to support a variety of missions. It climbs stairs and negotiates obstacles. MATILDA can also carry the sensors to support remote sensing and sample collection for hazardous material (HAZMAT) and weapons of mass destruction (WMD) operations. [Ref. 6] The Matilda
is currently used in numerous robotic experiments being conducted by the Joint Robotics Program.

![Mesa Associates Matilda Robot](image)

**Figure 2.2** Mesa Associates Matilda Robot. [From Ref. 6].

3. **iRobot – Urban Robot**

iRobot’s Urban Robot project was developed as a robust robot to aid military operations in urban terrain. The resulting robot has become a key platform for DARPA’s Tactical Mobile Robotics Program. The Urban Robot’s mechanisms are designed to afford it complete freedom of movement in indoor and outdoor environments. Its invertible mobility platform is equipped with tracked "flippers" that allow the robot to self-right, climb hills and stairs, and assume an upright posture suitable for navigating narrow, twisting passages. It is housed in an impact-resistant polycarbonate shell that has allowed the prototype system to survive multiple launches from a second story window.
The Urban Robot operates under autonomous, radio-controlled, and supervised autonomous control. [Ref. 7]

Figure 2.3  iRobot's Urban Robot. [From Ref. 7]
III. SMART THESIS OBJECTIVES

A. THESIS GOALS

The goal for this thesis was to research, evaluate, procure and build a SMART platform. While the overarching goal focused on updating and improvement of actual technological components, five specific themes categorized the final effort expended in this thesis research.

1. Selection of a NPS Robotic Platform

Although the recent CSS&T department's robotic systems focused primarily on indoor robotic applications, increasing the military applicability of the research necessitated the capability of operating in a realistic outdoor environment. The move from inside to outside required the development of sequential steps to accomplish this aggressive vision. Certainly a new robotic platform would have to be developed or acquired to move beyond the current indoor, two-wheeled, circular robotic chassis to something with a more realistic military application. Therefore, the investigation and selection of a new platform became a necessary goal of this project.

2. Selection of an Embedded Microcontroller

Under the generalized theme of robotic technology improvement, the selection of a microcontroller initially seemed less crucial to the overall success of the incremental plan. However, it soon became one of the most exciting aspects of the research. In developing criteria for the selection process, one of the desired requirements was that the processor be programmable in C code. This seemed to be a logical step since the CSS&T Curriculum already included a required class in C programming. Other criteria that were considered important were microprocessor speed and flexibility. The future possibility of
computing a reasonable amount of onboard sensor data drove the requirement for increasing the speed of the processor.

Additionally, modular flexibility was crucial because of the intended future 'general purpose' combat systems research mission the vehicle is intended to perform. This flexibility included a robust Input/Output (I/O) capability that could handle many of the possible sensor variations and mission areas being researched within the department's applied physics discipline. Finally, since the initial planning determined that the use of a common wireless Internet network provided great advantages for robotic control and interface, the processor needed to be easily compatible with Internet Protocol standards.

3. Selection of a Navigation Methodology

Autonomous navigation is one of the more complex problems facing robotic designer. Since the NPS SMART platform is designed to have the flexibility of autonomous operation it was necessary to decide on a highly accurate method to update the position of the vehicle. Moving the device to an outdoor environment compelled research into current methods of robotic navigation and the selection of the most advantageous method to pursue. Therefore, an additional goal of navigation research and selection was included in the SMART developmental plan.

4. Integrating an Operable Platform for Future NPS Research

Having researched the platform, microcontroller, and navigation system; the integration of the hardware and software architecture required for the robot could then be implemented. This led to the final goal of actually acquiring and integrating the microprocessor with the robotic platform and making it available for future departmental robotic sensor research.
5. **Engage in Cooperative Research with current Military Robotic Labs and Programs**

A spin-off of the SMART project research was the possibility of collaboration of NPS with other DOD labs working in similar fields of research. Two research institutions, SPAWAR Systems Center, San Diego and the Coastal Systems Station, Panama City appear to be eager to partner with the NPS SMART program in this endeavor.
IV. NPS SMART ROBOT

A. OVERALL DESIGN

The initial research on all-terrain robotic platforms was conducted by surveying numerous research labs and commercial robotic companies. SPAWAR – San Diego, CSS – Panama City, Foster-Miller and Mesa Associates were the principal organizations that contributed to the eventual selection of the current platform. The primary motivators that drove our vehicle selection were: price, laboratory research experience and vehicle availability. Current military experimentation with all-terrain robots leans toward a tank-like tracked vehicle. For man-portable robotic application, the “TAR’s”, “Matilda’s”, and “Urban Robots” are the dominant man-portable vehicles demonstrating potential for future employment in autonomous military applications. However, the research and developmental costs incurred by each of the manufacturing companies drive the current prices beyond the range of what the CSS&T department should pay for a research vehicle. Price alone drove us to obtain our initial vehicle from another Naval research lab.

The specific rover acquired for a SMART platform is a Foster-Miller Lemming tested by the Coastal Systems Station (CSS) Naval research facility in Panama City, Florida and delivered to the Combat Systems department in April of 2001. The Lemming is a small all-terrain tracked-vehicle designed by Foster-Miller with DARPA funding. The Lemming vehicles were developed as early prototypes to test the concept of small tracked rovers in reconnaissance, sensor testing and weapons delivery support. They are early generation models of the current Foster-Miller TAR platforms. Foster-Miller
designed these all-terrain robots with an on-board micro-processing capability that allows for both autonomous and manual vehicle control through a computer interface.

B. ROBOT CHASSIS

The main body of the SMART robot is constructed of an aluminum chassis. There are two tank-like plastic tracks one each side of the robot that are driven by twin DC servomotors. The tracks are kept secure on the vehicle by cogwheels on the front and grooved wheels on the rear mount. The top of the rover has a removable plate that allows for access into the main body cavity. The battery pack that arrived with the robot is a 12-volt, 6.9 amp-hours power supply consisting of three Panasonic, (12-Volt, 2.3 amp-hours) rechargeable, sealed lead batteries.

Figure 4.1 NPS SMART All-Terrain Robot.
Diameter (without track): 6.5 in.

Height Ground to Track: 7 in.

Main Body Height: 3.5 in.

Ground Clearance: 1.5 in.

Vehicle Length (track to track): 20 in.

Figure 4.2 SMART Vehicle Side Dimensions (± 0.25 in.).

Body Length: 15.5 in.

Body With: 15 in.

Motor Box Width: 22 in.

Figure 4.3 SMART Platform Interior Dimensions (± 0.25 in.).
C. MICROPROCESSOR – IP ENGINE

Several criteria were identified as important characteristics for a microcontroller to run the SMART platform: robust I/O, abundant memory, C programmability, and ease of network integration.

Based on these criteria, the controller chosen for the SMART platform was the ipEngine made by Bright Star Engineering (BSE), Inc. The ipEngine is a credit-card sized microcontroller that vastly improves upon the capabilities of the Z-World BL-1500, the previous microcontroller used to drive the CSS&T robots. The microprocessor that drives the ipEngine is a Motorola PowerPC MPC 823. The peripherals included with this processor are designed to easily “internet enable” a broad array of commercial products. There are many capabilities it incorporates that make the ipEngine an ideal microcontroller for the SMART robot.

1. Capabilities
   a. Network Integration

      The ipEngine network integration capability was one of the primary reasons that it was chosen as the SMART on-board microprocessor. The Ethernet connection, an Apache web server, and standard Internet Protocol interfacing facilitate the quick integration of the robot into a network environment.

   b. Real Time Operating Kernel

      The autonomous requirement for the SMART vehicle required a controller with real time programming capability. The ipEngine allows for real time operation through its pKernel Operating system, an efficient run-time system supporting a full range of real-time features including preemptive scheduling, priority inheritance, and nested interrupts.
c. **C programmable**

The ipEngine software developer’s kit allows for programming in C or C++. This was an essential capability of the controller because of the flexibility C programming provides, plus C programming is supported by classes taught in the Combat Systems sequence.

d. **Virtual Interface**

The ipEngine has an 88 pin “virtual interface” Field Programmable Gate Array (FPGA) that gives flexibility in configuring the ipEngine into a robotic controller. The Altera FPGA can be configured to emulate a variety of bus architectures as well as to implement peripheral functions like UART’s (Universal Asynchronous Receiver/Transmitter), PWM (Pulse Width Modulation) control, memory emulation, data capture, and synthesis, and interfacing to a variety of input devices.

e. **66 MIPS, Motorola Power PC MPC823 CPU**

The MPC823 microprocessor is a versatile, one-chip integrated microprocessor and peripheral combination that can be used in a variety of electronic products. It particularly excels in low-power, portable, image capture and personal communication products. It has a universal serial bus (USB) interface and video display controller, as well as the existing LCD controller of the MPC821 device.

The MPC823 microprocessor integrates a high-performance embedded PowerPC core with a communication processor module that uses a specialized RISC (reduced instruction set computer) processor for imaging and communication. A RISC is a microprocessor that is designed to perform a smaller number of types of computer instruction so that it can operate at a higher speed. The MPC823 communication
The processor module can perform embedded signal processing functions for image compression and decompression. It also supports seven serial channels—two serial communication controllers, two serial management controllers, one I²C port, one USB channel, and one serial peripheral interface. [Ref. 8]

The number of MIPS (million instructions per second) is a general measure of the MPC823's computing performance and, by implication, the amount of work it can do. Historically, the cost of computing, measured in the number of MIPS, has been reduced by one-half annually and the 66 MIPS specification of the MPC823 is a generous amount for our current platform.

Figure 4.4 ipEngine with Protective Case.
2. **FLASH Memory – 256 K**

The ipEngine has 4 MB of Flash memory organized as 8 8KB blocks followed by 31 64KB blocks. Flash memory occupies the entire address range from 0xFE000000 through 0xFE1FFFFF. The first four blocks are reserved for use by the system: two for the boot loader and two for non-volatile parameter storage. The memory addresses from 0xFE008000 through 0xFE1FFFFF can be used for program storage. This is a great increase over the former microcontroller used in the department, the BL1500, which had 256K of Flash EPROM. [Ref. 10]
Flash memory is a type of nonvolatile memory that can be erased and reprogrammed in units of memory called blocks. It is a variation of electrically erasable programmable read-only memory (EEPROM), which unlike flash memory, is erased and rewritten at the byte level. Flash Memory works much faster than traditional EEPROMs because instead of erasing one byte at a time, it erases a block or the entire chip, and then rewrites it. You can read and write to flash bytes or blocks, but you can only erase an entire block. Flash memory is different than Flash random access memory (RAM). The difference is that Flash RAM requires power to maintain its contents, while Flash Memory will maintain its data without any external source of power. This allows the permanent storage of programs on to the ipEngine that will not be erased by power loss. [Ref. 10]

3. DRAM Memory – 16 MB

The ipEngine has 16MB of DRAM, which is a dramatic increase over the BL1500’s 128K of SRAM. The address range for the DRAM is 0x00000000 through 0x00FFFFFF. The first 16KB, from 0x0000 to 0x3FFF, is the zero page for the Motorola microprocessor. It is used for things like vector tables and must not be written to. Programs can be written to any address from 0x00004000 through 0x00FFFFFF. [Ref. 9]

Dynamic random access memory (DRAM) is the most common kind of RAM for personal computers and workstations. Random access means that the PC processor can access any part of the memory or data storage space directly, rather than having to proceed sequentially from some starting place. DRAM is dynamic in that, unlike static random access memory (SRAM), it needs to have its storage cells refreshed or given a new electronic charge every few milliseconds. DRAM stores each bit of information in a
storage cell consisting of a capacitor and a transistor. Capacitors tend to lose their charge rather quickly; thus, the need for recharging. DRAM is the place in a computer where the operating system, application programs, and data in current use are kept so that they can be quickly reached by the computer's processes. However, the data in RAM stays there only as long as your computer is running. When you turn the computer off, RAM loses its data. When you turn your computer on again, your operating system and other files are once again loaded into RAM, usually from the hard disk (Flash Memory for the ipEngine). [Ref. 10]

4. **ipEngine I/O Ports**

One of the criteria for selecting the ipEngine was the great flexibility in I/O interfacing with the microprocessor.

- **Dual RS-232**

  The RS-232 is a serial port connection that allows for encoded bits to be transmitted one at a time. The communication protocol for serial communication requires that the transmitter inform the receiver that it is about to send information. Once the receiver detects this signal, called a “start bit”, it will listen to the sequential transmission of information until the transmitter sends a “stop bit” declaring the completion of the transmission. The ipEngine has two standard RS-232 serial I/O ports.

- **LCD/Video Controller**

  The LCD/Video controller allows for an easy method for integrating a liquid crystal display for user interfacing or video camera sensing systems.

- **10 BaseT Ethernet**

  The 10BaseT Ethernet connection allows the ipEngine to easily be configured to a local area network system (LAN). This is a tremendous benefit to our
project since this is one of the innovative methods we intend to explore. Ethernet is currently the most widely installed local area network (LAN) technology. Specified in a standard, Institute of Electrical and Electronics Engineers (IEEE) 802.3, an Ethernet connection typically uses coaxial cable or special grades of twisted pair wires. The most commonly installed Ethernet systems are called 10BASE-T and provide transmission speeds up to 10 Mbps. Devices are connected to the cable and compete for access using a Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol.

On an Ethernet LAN, any device can try to send a "frame" of information at any time. A "frame" is simply a term for data that is transmitted between network points and is transmitted as a unit complete with addressing and necessary protocol control information. A frame is usually transmitted as serial binary digits and contains a header field and a trailer field that "frame" the data. Each device on the LAN senses whether the line is idle and therefore available to be used. If it is, the device begins to transmit its first frame. If another device has tried to send at the same time, a collision is said to occur and the frames are discarded. Each device then waits a random amount of time and retries until successful in getting its transmission sent. Carrier Sense Multiple Access/Collision Detect (CSMA/CD) is the protocol name for this information de-confliction.

10BaseT, one of several physical media specified in the IEEE 802.3 standard for Ethernet LANs, is ordinary telephone twisted pair wire. This designation is an IEEE shorthand identifier. The "10" in the media type designation refers to the transmission speed of 10 Mbps. The "BASE" refers to base band signaling, which means
that only Ethernet signals are carried on the medium. The "T" represents twisted-pair; 10BASE-T supports Ethernet's 10 Mbps transmission speed. [Ref. 10]

d. **USB port**

USB (Universal Serial Bus) is a "plug and play" interface between a computer and add-on devices (such as audio players, joysticks, keyboards, telephones, scanners, and printers). With USB, a new device can be added to the ipEngine microcontroller without having to add a special adapter card.

5. **FPGA (Field Programmable Gate Array) “Virtual Interface”**

Probably the most exciting and unique item that the ipEngine contained was a 16,000 Gate, Flex 6016 Programmable Logic Device produced by the Altera Corporation. The 88-pin Flex 6016 forms the core of the ipEngine “virtual interface” capability. This FPGA-like device allows us to synthetically emulate virtually any device interface through the use of software controlled digital logic.

a. **FPGA Background**

A Field-Programmable Gate Array (FPGA) is an integrated circuit composed of numerous digital logic cells arranged in two-dimensional arrays. These cells can vary widely in their makeup, but a typical cell might contain a flip-flop, a few multiplexers, and perhaps a small look-up table. More sophisticated devices have a much more complex logic cells that might contain several flip-flops, several look-up tables and a selection of other logic gates. The unique characteristic of the FPGA is that the user can determine exactly how the logic gates are interconnected through a programmable interface. Another advantage of a FPGA is that the contents of the memory can be changed as often as desired. One disadvantage is that the content of the memory element...
is volatile and is lost when power is removed. To overcome this, the programmable states for the various interconnections, in the form of a computer program, must be loaded from some non-volatile memory (ipEngine Flash Memory) when power is first applied. [Ref 11]

b.  *Altera Flex 6016 Programmable Logic Device*

The FPGA-like chip on the ipEngine is the 16,000 gate, Flex 6016 Programmable Logic Device produced by the Altera Corporation. The SRAM-based Flex 6016, or EPF6016, is very similar to a standard FPGA, however it uses more efficient programmable logic architecture, the OptiFLEX architecture. The OptiFLEX architecture is built on a 5.0-V, 0.30-micron or a 5.0-V, 0.42-micron, triple-layer metal CMOS process. Every feature in the OptiFLEX architecture is targeted at producing maximum performance and utilization in the smallest possible area.

![Figure 4.6 Relative Sizes of a Standard Gate Array and a FLEX 6000. [From Ref. 12]](image)

The EPF6016 device forms an 88-pin "virtual interface" that is configured according to the consumer's particular needs. It can emulate a variety of bus architectures as well as implement peripheral functions such as UARTs, PWM controls,
memory emulation, data capture and synthesis, and interfacing to a variety of devices. A synchronous 128-Kbyte × 16 SRAM is connected to the EPF6016 device. The SRAM can be used as a high-speed shared buffer storage for data coming from or going to the virtual interface. [Ref. 12]

The programming interface for the EPF6016 is the Altera MAX+PLUS II BASELINE development software. By downloading the software free of charge from the Altera web site, any developer can configure the EPF6016 device. Sample configuration files in the Altera Hardware Description Language (AHDL) and Verilog Hardware Description Language (VHDL) can be found on the BSE web site at http://www.brightstareng.com. The programming interface allows the SMART project to tailor the ipEngine to our specific I/O configuration needs with relative ease. To further simplify the process of configuring the EPF6016 device and defining the virtual interface, Bright Star Engineering is developing a library of pre-compiled configurations for the ipEngine-1 that will be available on their web site.
Figure 4.7  Altera Max+plus II Developmental Software for the Flex 6016.

Figure 4.8  Altera's Flex 6016 Mounted on the ipEngine. [From Ref. 8]
6. Power Requirements

The ipEngine has an on-board switching power supply capable of providing 2 amps of current at 3.3 volts or 2 amps of current at 5 volts. The power supply can be used to provide power for both the ipEngine and the user’s electronics. Typical power supply is advertised as 25mW to 2 Watts depending on the application. The user has the following options for supplying power to the ipEngine.

<table>
<thead>
<tr>
<th>Input Power</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Unregulated 7-18 V DC</td>
<td>If you want to use the ipEngine’s on-board switching power supply.</td>
</tr>
<tr>
<td>2. Regulated 5 V DC</td>
<td>If your system already had a regulated 5 V power supply source and you don’t have a 7 V CD source handy. In this case the on-board power supply generates 3.3 V from the 5 V input.</td>
</tr>
<tr>
<td>3. Regulated 5 V and 3.3 V DC</td>
<td>Use this configuration if switching noise is a concern, and if you wish to completely disable the on-board supply.</td>
</tr>
</tbody>
</table>

Table 4.1 User Options for Power Supply. [From Ref. 13]

The protective case for the ipEngine has an on-board receptacle enabling the connection of the chassis to 110 V AC receptacle through a standard power module. Tables in Appendix A outline several different configurations for supplying power to the ipEngine. [Ref. 13]

7. ipEngine Internal Arrangement

Below is a simplified block diagram of the ipEngine. It shows many of the features of the microcontroller that have previously been described.
8. IP Engine SMART Hardware Integration

The ipEngine will be housed inside the SMART vehicle in the interior of the platform's main body compartment shown in Figure 4.10.
The future vision for the SMART vehicle is to have the one or two ipEngines controlling all communication, navigation and sensor processing with external and internal system devices. To accomplish this, two different on-board configurations will be explored. The first will be a microcontroller centered processing section. This is a traditional method of driving a robotic platform. Figure 4.11 illustrates this configurations with a single microprocessor centered as the control for numerous on-board devices.
The second and more innovative method is to take advantage of LAN technology and use a network hub as the centerpiece of the processing section. For this, an on-board network hub will be the centerpiece of the SMART device connections. The flexibility with this configuration is the ability to use two separate microcontrollers to control different functions on the robot. A model of this would look similar to Figure 4.12. One ipEngine could navigate and drive the platform and another could operate the sensing devices. This arrangement will be a huge advantage when a bigger platform becomes available and we can use the robot as a test platform for sensing packages. A network hub connection will allow a research team to plug their sensor package into the network and begin experimentation with minimum re-configuration to the platform or the device. Similar configurations have been tested with much success at SPAWAR’s robotic division in San Diego.
D. SMART SOFTWARE

1. Real Time Operating System

A real-time operating system (RTOS) is simply an operating system that guarantees a certain capability within a specified time constraint. In general, real-time operating systems are said to require:

- Multitasking
- Process threads that can be prioritized
- A sufficient number of interrupt levels

Real-time operating systems are often required in small, embedded operating systems that are packaged as part of micro devices. Some kernels can be considered to meet the requirements of a real-time operating system. However, since other components, such as device drivers, are also usually needed for a particular solution, a real-time operating system is usually larger than just the kernel. [Ref. 8]
a. Background

The kernel is the essential center of a computer operating system, the core that provides basic services for all other parts of the operating system. A kernel can be contrasted with a shell, the outermost part of an operating system that interacts with user commands. Kernel and shell are terms used more frequently in UNIX-like operating systems.

Typically, a kernel includes an interrupt handler that handles all requests or completed I/O operations that compete for the kernel's services, a scheduler that determines which programs share the kernel's processing time and in what order, and a supervisor that actually gives use of the computer to each process when it is scheduled. A kernel may also include a manager of the operating system's address spaces in memory or storage, sharing these among all components and other users of the kernel's services. A kernel's services are requested by other parts of the operating system or by an application through a specified set of program interfaces sometimes known as system calls.

Because the code that makes up the kernel is needed continuously, it is usually loaded into computer storage in an area that is protected so that it will not be overlaid with other less frequently used parts of the operating system. [Ref. 8]

b. pKernel

The heart of the ipEngine is a real time operating system named pKernel that employs POSIX and ANSI C standards to facilitate an industry standard programming interface. The pKernel operating system integrates the following elements of the ipEngine:
• TCP-IP networking
• Embedded Apache web server
• Local RAM file system
• Access to Internet files via FTP and HTTP
• Interactive command shell
• Software development kit [Ref. 9]

The core of pKernel is an efficient operating system supporting numerous features including: preemptive real-time scheduling, priority inheritance, and nested interrupts. The run-time system also supports many of the basic POSIX (Portable Operating System Interface) 1003b.1 system calls, including those for file/directory management, I/O primitives, memory allocation and time management.

Figure 4.13  pKernel Operation System Breakdown. [From Ref. 9]

The pKernel software developer's kit provides an integrated development environment based upon the industry standard GNU tool chain. The toolkit includes the GCC cross-compiler, a linker, archive librarian and other utilities. Source level debugging of code running on the target system is accomplished via a thread aware version of the GNU GDB debugger. [Ref. 9]
2. **Portable Operating System Interface (POSIX) Threads**

   a. **POSIX Background**

   POSIX is a set of standard operating system interfaces based on the UNIX operating system. Informally, each standard in the POSIX set is defined by a decimal following the POSIX. Thus, POSIX.1 is the standard for an application program interface in the C language. POSIX.2 is the standard shell and utility interface. These are the main two interfaces, but additional interfaces, such as POSIX.4 for thread management, have been developed or are being developed. The Portable Operating System Interface.4a C specification provides a set of application program interfaces that allow a programmer to include thread support in the program. The POSIX interfaces were developed under the auspices of the Institute of Electrical and Electronics Engineers. [Ref. 10]

   b. **POSIX Threads**

   pKernel’s run-time task management and inter-task communication and synchronization are accomplished via POSIX threads. The primary function of just about any real-time operating system (RTOS) is to let you execute one or more sequences of control on a single microprocessor as if they ran concurrently and independently. A thread is simply a placeholder for information associated with a single use of a program that can handle multiple concurrent users. From the program’s point-of-view, a thread is the information needed to serve one individual user or a particular service request. If multiple users are using the program or concurrent requests from other programs occur, a thread is created and maintained for each of them. The thread allows a program to know which user is being served as the program alternately gets called on behalf of different users.
Typically, the operating system switches execution from one flow of control to another, quickly enough to create the illusion that all flows are running concurrently. It also provides various ways for the flows to communicate with each other and with the outside world. In computer lingo, a process is expensive, but a thread is cheap. Switching between processes and managing the memory and its protection requires far more resources than are required if threads are used. The pKernel is a simple, fast, and small operating system. It provides a single global memory space, and manages threads very effectively. POSIX threads are scheduled on a priority basis; the RTOS executes the thread with the highest priority number that is ready to run. pKernel provides a preemptive priority-based scheduler with priority inversion protection to do this.

Threads communicate with each other via semaphores, message queues, and mailboxes. Threads communicate with the outside world via interrupts, at the lowest level, and via the network and other I/O devices on the ipEngine. Software modules called drivers present a software interface to these devices, and manage the interrupt and hardware register manipulation for the programmer. [Ref. 9] For more information about POSIX threads, consult *Programming with POSIX Threads* by David R. Butenhof. [Ref. 14]

3. **Software Component Locations**

The software tools for programming, created by Bright Star Engineering for the ipEngine, are included with the programmer's software developmental package. When installed on a computer, all of the ipEngine software is located in the BSE directory in three components: pKernel, ipEngine and GNU tools. Each of these is
compartmentalized in a directory under the same name. See Appendix A for a Table outline of the BSE software location.

a. \textit{ipEngine Directory}

The ipEngine directory includes only a few items of importance. The most significant is a copy of the ipEngine Hardware Reference manual. This is a pdf file that outlines the setup and operation of the ipEngine.

b. \textit{pKernel Directory}

The pKernel directory contains several items of importance for SMART vehicle programming. The \texttt{doc} subdirectory contains the pdf files for the pKernel Programmers Reference Guide and the Software Developer's Kit Users Guide. The \texttt{include} subdirectory contains the header files that define the programming interfacing to the pKernel. The \texttt{lib} subdirectory lists all the library files required for C programming. The \texttt{samples} subdirectory contains several sample programs that highlight a few of the capabilities of the ipEngine. Finally, the \texttt{downloads} subdirectory stores the binary programs for downloading to the ipEngine.

c. \textit{GNU Tools Directory}

The gnutools directory contains all of the necessary executable programs and documentation for the GNU development software. The \texttt{doc} subdirectory contains the documentation files for the GNU tool set. The \texttt{html} subdirectory contains the documents in the HTML format. The \texttt{pdf} subdirectory lists the same documents in pdf format.
4. Essential Software Interfacing Tools

The process of compiling a program and loading it on the ipEngine requires the understanding of three different software applications. These three required tools are:

- FTP Server
- Command Shell
- Terminal Emulator

a. PumpKIN FTP Server

A Trivial File Transfer Protocol (TFTP) is a standard Internet protocol that will allow the exchange of files between computers on a network. Like Hypertext Transfer Protocol (HTTP), which transfers displayable Web pages and the Simple Mail Transfer Protocol (SMTP) that transfers e-mail, TFTP is an application protocol that uses the Internet’s TCP/IP protocols. The primary method for transferring executable computer code onto the ipEngine is through FTP software. The current software used with the SMART platform is the PumpKIN FTP.

![Figure 4.14 PumpKIN FTP Application Command Window.](image)
b. **Command Shell**

A shell is a UNIX term for the interactive user interface with an operating system. The shell is the layer of programming that understands and executes the commands a user enters. A shell usually implies an interface with a command syntax (think of the DOS operating system and its "C:\>" prompts and user commands such as "dir" and "edit"). As the outer layer of an operating system, a shell can be contrasted with the kernel, the operating system's inmost layer or core of services.

![Figure 4.15 MS-DOS Command Shell Window.](image)

The MS-DOS Shell is the primary command interface for initializing the GNU software for compiling SMART code. However, pKernel also provides an interactive command shell capability that can be accessed by users via a serial port, telnet connection or through a program via the "system" function call. Over 50 pKernel shell commands are available to support debugging and system configuration. In addition, the
pKernel command set and shell can be tailored or completely replaced by the end-user to provide a customized interactive command environment. [Ref. 9]

c. **Terminal Emulator**

The idea of terminal emulation is to allow a user to create a connection from a terminal to a remote computer. It gives the user the opportunity to be on one computer system and do work on a completely separate system, which may be across the street or thousands of miles away. Telnet is the main Internet protocol for creating a connection with the ipEngine. The telnet link from a computer to the ipEngine is accomplished by creating a socket connection. A standard telnet session is normally conducted on socket port 23. This is the default standard for normal telnet interfaces and communication is conducted over the ipEngine Ethernet connection.

For the SMART platform two software programs are currently used as a terminal emulator communication programs. The first is Microsoft Telnet for Windows NT. The second is a freely downloadable software program called Tera Term Pro. Either program allows connection to the IP Engine with a standard TCP/IP connection. However, the Tera Term Pro allows for serial port connections as well and this is an essential capability for loading programs onto the IP Engine.

There are two primary functions that the telnet application provides. The first is the ability to FTP the C code onto the IP Engine through a serial cable connection. The second is to create a connection from a computer terminal to the IP Engine microprocessor for command interfacing or program debugging.
5. GNU Software

The software developer's kit delivered with the ipEngine includes a set of tools developed under the name GNU and includes a compiler, linker and debugger. GNU is a wide-range of Open Source-based, freely downloadable software programs that are licensed under the terms of the General Public License (GPL). GNU, which stands for "Gnu's Not Unix", is the name for the complete Unix-compatible software system. GNU software is created to be very similar to a Unix environment but with many improvements, both practically and politically. There are many open-source development tools available within GNU software. However, the SMART project currently only uses the GNU CC and GNU Make utilities to implement the embedded C code that controls the robot.
a. **GNU CC**

"GCC" is a common shorthand term for the GNU C compiler. This is both the most general name for the compiler, and the name used when the emphasis is on compiling C programs. The GNU C compiler can compile programs written in three languages, C, C++, and Objective C. When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. This simply changes your text file, C source code, and changes it to an executable binary file. There are options within the software that allow you to adjust how the compiler does this, but that is beyond the scope of this document. The utility that directs the compilation process is GNU Make. [Ref. 14]

b. **GNU Make**

Almost all C compilers come with a ‘make’ utility that can simplify and speed the task of working with multiple-source code files. ‘GNU Make’ is the utility in the GNU software package that automates the process of compilation. Ultimately you can use a make utility with any programming language whose compiler can be run with a shell command. Indeed, make is not limited to just programs. You can use it to describe any task where some dependant files must be updated automatically from source files whenever the original source files change. However, for the SMART platform GNU Make is primarily used as the method to link and compile the robot program.

When GNU Make is commanded to compile a program, each user-altered C source file must be recompiled. If a header file has changed, each C source file that includes the header file must be recompiled as well. Each compilation produces an object file corresponding to the original source file. If any source file has been
recompiled, all the object files, whether newly made or saved from previous compilations, must be re-linked together to produce the new executable program.

When compiling large programs, GNU Make automatically figures out which files it need to update, according to which source files the programmer changed. It updates only those non-source files that depend directly or indirectly on the source files that were changed. In a case where one non-source file depends on another non-source file, it also automatically determines the proper order for updating files. GNU Make greatly simplifies the compilation of multi-source SMART C code. [Ref. 15]

E. SMART NAVIGATION SYSTEM

For autonomous applications the SMART vehicle will require very precise positioning and navigation techniques. The primary method to achieve this accuracy will be through the use of a Differential Global Positioning System (DGPS).

DGPS is a way to correct for the various inaccuracies in the normal GPS system. Accuracy with DGPS can yield measurements to a couple of meters in moving applications and sub-meter in stationary situations. For a standard GPS system each of the timing measurements from four satellites are figured into the position calculation and the compounded error of each of these signals translates into an overall positioning error. However, the satellites are so far out in space that the distances traveled here on earth are proportionally insignificant. So, if two receivers are fairly close to each other, say within a few hundred kilometers, the signals that reach both of them will have traveled through virtually the same slice of atmosphere, and so will have virtually the same errors. The Differential GPS uses this concept to correct for error by comparing the received signals
of two GPS units, one that's stationary and another that's roving. The stationary receiver is the key because it ties all the satellite measurements into a solid local reference. [Ref. 17]

**DIFFERENTIAL GPS POSITIONING**

![Diagram of Differential GPS Signals](image)

Figure 4.17 Differential GPS Signals. [From Ref. 18]

Differential GPS eliminates all errors that are common to both the reference receiver and the roving receiver. These include everything except multipath errors and any unique receiver errors. The idea behind differential GPS is to have one receiver measure the timing errors and then provide correction information to the other receivers that are roving around. That way, virtually all errors can be eliminated from the system, even the Selective Availability error that the DOD puts in on purpose. The idea is
simple. Put the reference receiver on a point that's been very accurately surveyed and keep it there. This reference station receives the same GPS signals as the roving receiver, but instead of working like a normal GPS receiver, it attacks the equations backwards by using its known position to calculate timing. It figures out what the travel time of the GPS signals should be, and compares it with what they actually are. This difference is an "error correction" factor. The stationary receiver then transmits this error information to the roving receiver so it can use it to correct its measurements. [Ref. 17]

There are a couple of techniques that the SMART platform can implement in using a DGPS system. The first is to purposely place a stationary receiver in a location from which it can transmit error corrections to all the robotic vehicles. The second and more innovative method is to use the Internet for distributing corrections to a single or distributed system of vehicles from a centrally located command center. For example, consider a fleet of SMART vehicles that we would like to pinpoint on a beachhead with very high accuracy. To obtain this accuracy without equipping each vehicle with expensive differential receivers for every platform an "inverted DGPS" system could be used. With an inverted DGPS system the SMART robots would be equipped with standard GPS receivers and a transmitter would send standard GPS positions back to the tracking center. Then at the tracking center the corrections would be applied to the received positions. It requires a computer to do the calculations and a transmitter to send the data, but it enables accurate positioning for a fleet of vehicles for the cost of one reference station, a computer, and a lot of standard GPS receivers. [Ref. 17]. See Appendix B for further GPS theory.
F. COMMUNICATION SYSTEM

1. Wireless Ethernet Radio Link

The wireless communication system tested for the SMART platform consists of two models of Proxim Radio Frequency (RF) modules. The first is the RangeLAN2 7921 Ethernet Adaptor. The Ethernet adapter is a long range, high performance local area network (LAN) product that allows Ethernet capable products to communicate wirelessly with network computers. The second model is the RangeLAN2 7910 Serial Adapter. The 7910 is a radio module that replaces standard RS-232 serial cables with a transmitter that broadcasts spread spectrum RF.

Figure 4.18 Proxim Radio Link.
G. ROBOT CONTROL

The SMART platform is intended to have the flexibility for both autonomous and user-directed control. To achieve this, the SMART initiative intends to create an innovative Internet Web browser for both active user control and a vehicle situational interface. The ipEngine’s industry standard Apache web server allows for some unique robotic applications. Currently, there are no other commercial projects using the ipEngine’s web server capability. The hosting of an on-board web site by the SMART robot will enable a user to access the site from any terminal with access to the SMART network. A web site interface will be created that will allow for: direct operator control of the robot, robot location reporting, FTP capability of files to update robot in-mission programming, and the visualization of on-board sensor data. Figure 4.19 shows a graphical interface similar to the desired SMART web control page.
Figure 4.19  Sample Operator Control Interface. [From Ref. 15]
V. SMART INTEGRATION

A. EXPERIMENTATION

1. Goals

There were four main goals of the experimentation with the software integration of the ipEngine. The first was to simply gain familiarity with the software package included with the Software Developers Kit. The second goal was to investigate the capabilities of the ipEngine in order to provide a framework for the follow-on students who will continue the development of the SMART platform. The third goal was to actually formulate code that would test the basic communication interfacing required for integration of the ipEngine as the SMART microcontroller. The final goal was to document the lessons learned, in a manual-like format, for the SE-3015 robotics laboratory.

2. Experimental Setup

The software integration was conducted by connecting a desktop computer, loaded with the BSE developmental software, to the ipEngine. The interfacing connections between the computers consisted of two types: serial and Ethernet. Initially, direct cables were used to communicate, however in an effort to test the ability to operate remotely, Proxim RF transmitters later replaced the direct Ethernet and Serial cable connections. Figure 5.1 displays the direct and wireless connections between the computers.
B. SMART CODING

1. Writing C code

The procedures for writing and executing C code for the SMART platform closely follows an UNIX-like programming environment. The actual C Code was written in a standard text editor, like Notepad or WordPad.

2. Compiling
   a. GNU Make

As described earlier, GNU Make is the application software used to compile the C code for SMART programming. The knowledge of how to compile and build programs is directed from within GNU Make by a file called the makefile. Makefile code lists all source and non-source files for the program, describes their relationship to each other and provides commands for updating, linking, and compiling.
the finished program. *Makefiles* are run from a command shell like MS-DOS and when executed, automate the creation of the executable binary program for the ipEngine.

**b. Sample SMART MakeFile**

Here is a sample *makefile* (Figure 5.2) included in the IP Engine software samples. It is located in the folder: BSE\pKernel\samples\hello directory. The file name is “Makefile” and it is written to compile a simple C program for the IP Engine called “hello.c”.

```makefile
OBJS = hello.o
include ../../../Makefile.top
all: hello.bin
```

*Figure 5.2 Makefile for Hello World Program.*

```c
int main()
{
    sysInt();
    printf ("hello world\n");
}
```

*Figure 5.3 Hello.c.*

Figure 5.3 above lists the C program referenced in the hello.c *makefile*. Figure 5.4 shows the relative locations of the important files on the host computer in order to compile the “hello.c” program.
Relative locations of the files within the directories are important because GNU Make has to locate each of them and execute the commands to update, link, compile etc... as directed by the makefile. Typically, the best way to arrange a makefile to compile a program is to locate the makefile and the source code in the same directory. Otherwise, GNU Make will have to be instructed where to find the code.

The first line of the "Makefile" in Figure 5.2 declares a variable called OBJJS and assigns to it the object file "hello.o." The second line uses a directive called include that instructs GNU Make to suspend reading of "Makefile" and execute a special makefile, called "Makefile.top" before continuing. "Makefile.top" (see Appendix D) is called the 'master makefile' and was specifically created for the ipEngine software development package. As shown in Fig 5.4, "Makefile.top" is located in the BSE directory, two levels above the sample subdirectory where the "hello.c" code is located.
The location of "Makefile.top" with reference to "Makefile" is defined within the include statement. "Makefile.top" is important because it sets up the GNU Make software to locate all the necessary components it needs to automate the compiling of the SMART program. Therefore, ipEngine programming is designed to have "Makefile.top" included within all the SMART program makefiles. When GNU Make finishes executing the commands of "Makefile.top", it then jumps back into the "Makefile" and finishes executing the code. The final line is a directive all. The all command is a target that directs GNU Make command to create the binary file called "hello.bin" from "hello.c."

For more on makefiles see Appendix C or the GNU Make manual [Ref. 17] included with the BSE software.

C. SMART SOFTWARE INTERFACING

To properly program the ipEngine, several steps need to be addressed. These include: setting up the Network parameters, building a C program, loading the program onto the ipEngine and executing the program.

1. ipEngine Network Setup

To quickly integrate the ipEngine into a computer network environment, one of the first things that must be done is to set up its Internet protocol network parameters. The following parameters need address information assigned to them:

1. IP address of the ipEngine
2. Network Mask
3. Host name of the TFTP server
4. Gateway IP address (optional)
In the ipEngine experimentation, the laboratory desktop computer serves as the TFTP server. Its IP address is: 131.120.101.75. To define the IP parameters for the ipEngine, open a terminal emulator connection to the ipEngine and type the following.

```
>fset myip 131.120.101.70
>fset netmask 255.255.255.0
>fset serverip 131.120.101.75
>fset gateway 131.120.96.1
```

The actual address values above reflect the setup within the SP-111 robotics laboratory and will change depending on the future network configurations.

2. Building the Program from the MS-DOS Command Shell

The MS-DOS command window is the shell I used to direct the compilation of all the SMART code. To begin the compiling of a simple program like "hello.c" the program environment must be set up correctly. This is automatically done with a batch file called "setu.bat" or "setup.bat" in the root directory of the BSE software package.

```
@ECHO OFF
SET BSEDEF=c:\bse
SET BSEDEFU=c:/bse
SET BSEROOT=e:\BSE
SET BSEROOTU=e:/BSE
SET MAKE_MODE=unix
SET GDBTK_LIBRARY=%BSEROOT%\gnutools\share\gdbtcl
SET PATH=%BSEROOT%\gnutools\H-i586-cygwin32\bin;%PATH%
SET HOME=%BSEROOT%
```

Figure 5.5 setu.bat.

The setup batch file initializes the host's computers environment so that it can find all the Software Development tools. At the MS-DOS command prompt go to the BSE root directory and type setu (or setup). To build a program you must be within the
directory that the program and the makefile are in. Do this by changing directories within MS-DOS. Once in that directory type: make to execute the makefile within the directory. This will run the "makefile" program, build the binary file and place a downloadable binary file in the download directory of pKernel. Figure 5.6 shows the execution of these commands.

```
E:\bse\>yes
E:\bse\>all \setu
E:\BSE\pkernel\samples>cd hello
E:\BSE\pkernel\samples\hello>touch hello.c
E:\BSE\pkernel\samples\hello>make
E:\BSE\gnutools\lib\-i586-cygwin32/bin/powerpc-eabi-gcc "-mcpu=860" -nostdinc -le:$/BSE/pkernel\include -g -O2 -ffunction-sections -fdata-sections -fno-rtti -fno-exceptions -futable-ye -finit-priority -c hello.c -o hello.o
E:\BSE\gnutools\lib\-i586-cygwin32/bin/powerpc-eabi-gcc "-mcpu=860" -nostdinc -fdata-sections -ffunction-sections -o hello.x e:\$/BSE/pkernel\lib/powerpc-eabi-crt0.o hello.o -le:$/BSE/pkernel\lib\-util\-lnet\-lkern \-lc \-lutil \-lnet \-lkern \-lc \-lutil\-lnet \-lkern \-lc \-lgcc -le:$/BSE/pkernel\lib\-ipengine\-ld\-nostdlib
E:\BSE\gnutools\lib\-i586-cygwin32/bin/powerpc-eabi-size hello.x
text data bss dec hex filename
359204 8282 87396 454802 6f892 hello.x
e:\$/BSE\gnutools\lib\-i586-cygwin32/bin/powerpc-eabi-objcopy hello.x -0 binary hello.bin
cp hello.bin e:\$/BSE/pkernel\download\hello.bin
E:\BSE\pkernel\samples\hello>
```

Figure 5.6 Building hello.bin from MS-DOS Command Window.

The first command entered in Figure 5.6 is yes. I have written two batch files that simplify the command sequence within the MS-DOS window. "Go.bat" automates the calls to setu and changes directory to pkernel\samples (a source directory for running BSE sample programs) and "yes.bat" automates the call to setu and changes the directory to pkernel\src\smart (a source directory for saving SMART code). Therefore when I type go or yes from the BSE directory the file executes the setup batch file and places me in
the directory I would like to be in for programming. This is simply a time automation of the start up.

```
Call setu
cd pkernel\samples
```

Figure 5.7  yes.bat.

```
Call setu
Cd pkernel\src\smart
```

Figure 5.8  go.bat.

Note the use of the command **touch** in Figure 5.6. This simply updates the time stamp of the “hello.c” program so it appears to GNU Make that the program has been changed between compiles. This is only necessary if you have not changed the program; yet desire GNU Make to re-compile. Also notice that the second to last line in the window shows that “hello.bin” has been placed into the e:\BSE\pkemel\download directory where it can be accessed for FTP download to the ipEngine.

3. **Loading the Program**

Once the binary file it loaded into the BSE\pkemel\download directory, the next goal is to load the program onto the ipEngine. Here are the steps for configuring the FTP application (PumpKIN), the terminal emulator (Tera Term Pro), and then loading an executable program onto the ipEngine.

**a. PumpKIN FTP Setup**

1. Open the PumpKIN program. The following screen should appear.
2. Select **Options** and ensure the root download path is correct for the source location where the program is stored. For most **SMART** programming the default location prearranged by the GNU software is: `BSE\pkernel\download`. Also ensure that the Read Request Behavior is checked as **"Give all files"** and that the Write Request Behavior is set to **"Deny all requests."**

3. Ensure there is an Ethernet connection between the host computer and the ipEngine.
4. Keep PumpKiN (or another FTP program) open any time you are trying to FTP a file to the ipEngine.

b. Tera Term Pro Setup

The commands for downloading the program to the ipEngine are issued from a terminal emulation program such as Tera Term. Communication with the ipEngine is through Serial port 1 during program downloads. Therefore, ensure that the serial port is connected from the host computer to the ipEngine.

1. Open Tera Term. The following screen should appear.

![Tera Term Connection to Serial Port 1.](image)

2. Select Serial Port COM1.

3. Select Setup, Serial Port and configure the serial port as follows.
c. **FTP a Program to the ipEngine**

1. Open a terminal emulator program (Tera Term), ensure the host computer serial port is connected to the ipEngine serial port, select a serial connection, and reset the power button on the ipEngine. This should result in the following response from the ipEngine in the Tera Term window:


   >

2. At the command prompt, type the command `bload` followed by the program name and the memory location address.

   `>bload hello.bin 4000`

   The `bload` command loads the binary file `hello.bin` into memory location 0x4000 on the ipEngine. The bload command uses TFTP protocol to copy the file from the host computer to the ipEngine. In order for this to work, you must have a working Ethernet connection between the host computer and the ipEngine. Remember the FTP server (PumpKIN) must be up and running.
Figure 5.13 Commands for Loading hello.bin into the ipEngine.

3. Once this command is received, the PumpKIN FTP will transfer the file from the download directory onto the ipEngine via the Ethernet connection. Successful completion of this process will be confirmed by a “done” response in the terminal emulation window and a PumpKIN response of: “Transfer of ‘serialtest.bin’ has successfully been completed” (‘serialtest.bin’ will be replaced by the program name).

Figure 5.14 Pumpkin FTP Response for Successful Program Loading.
4. At the command prompt in the terminal emulator type: `go 4000` to execute the program. Figure 5.13 show the command sequence for hello.bin in the Tera Term Window.

![Figure 5.13](image)

Figure 5.15 Host Computer Screen With All Three Applications Open

D. **SMART I/O EXPERIMENTATION**

The focus of the software experimentation was to write C code that would simulate some basic I/O interfacing with the ipEngine. To do this, four separate functional command interfaces were investigated. These interfaces will become the foundational building blocks for the SMART platform software development. The four functional areas were: serial communication, telnet, sockets, and a web server interface. A building block approach was used to understand and test each I/O technique. Once each passed initial testing, they were integrated into one overall SMART control program. The individual testing of the interfaces aided the understanding of the multiple C commands that were required. The two I/O standards used for the simulation were
serial and sockets. Using these communication interfaces, four different I/O links were established simulating a typical SMART robotic application.

1. **Sockets Test**

The first test program was designed to simulate the possibility of using a standard socket connection to pass information on a network hub. The program “socketstest.c” created a simulated robot command window that accepted user inputs to drive a robot. The commands were simple 1 byte characters (f - forward, b - back, l - left, r - right) that the ipEngine would accept on socket port 2000, check it to see if it was a legal command for the robot, and return a movement status message back to the user on the same socket (port 2000). The movement responses transmitted by the ipEngine simply stated which direction the robot was moving based on the user’s input or, in the case of an unrecognized command, that the ipEngine did not understand the command. Appendix G. graphically depicts the command window and lists the C code for “socketstest.c.” Future uses for socket communication include: wireless terminal emulation interfacing, debugging code through telnet protocol, command and control interfacing, FTP upload/downloads of files, and on-board transfer of information from devices connected to a network hub. An introduction into sockets is located in Appendix E.

2. **Serial I/O Test**

The second test program built on the previous “socketstest.c” program and included the additional task of using the serial I/O capability of the ipEngine. Using the same robotic command driver window connected to port 2000, the serial test program modified what the ipEngine did with the user single character command (f, b, l, r). This time, in addition to transmitting the movement status of the robot to the user on socket
port 2000, the program sent a user's movement command out the serial port as well.

Appendix H. graphically depicts this transfer of information and lists the C code associated with “serialtest.c.” The future uses of serial communication are numerous and include: sending motor control strings to a DC motor controller and various interfacings with serial sensor devices.

3. **Web Server**

The last test program developed, ‘webserv.c’, investigated the ability to successfully load a simulated robotic control web page onto the ipEngine. Using the “webserv.c” sample program supplied by BSE, modifications were made to the sample index page on the ipEngine Apache web server. This capability was required to experiment with the possibility of a web control interface for the **SMART** platform. Appendix I. graphically depicts the simulated web page interface downloaded onto the ipEngine and lists the C code associated with “webserv.c.” This experimentation was one of the more frustrating ones because of repeated failures of the software to successfully run. After weeks of troubleshooting, it was determined that the underlying cause of the failures was simply corrupted software (a bad zip.exe file). Once this was discovered, the program executed perfectly. The future uses of the SMART control web page include: web control of the **SMART** platform, sensor data streaming for autonomous and semi-autonomous vehicles and FTP uploading of software.

E. **SMART CONTROL INTERFACE PROGRAM**

The final challenge was to integrate all three of these test programs into one single SMART control program called “smartcontrol.c.” The intent was to show the capabilities that the ipEngine has and therefore, the flexibility this gives the user. Figure 5.16 shows the final interfacing techniques set up for this experiment. When the program
was executed the ipEngine was able to communicate via four different multi-threaded methods at the same time. These methods were:

1. Terminal emulator telnet session on port 2000
2. Terminal emulator default telnet session on port 23
3. Web Browser connection on port 80
4. Serial Communication on serial port 1.

![Diagram showing different methods of communication](image)

**Figure 5.16** I/O Interfacing With the ipEngine

The "smartcontrol.c" program written for the control interface simulation takes the previous "serialtest.c" program and includes the web server addition. The program and the individual communication displays are outlined in Appendix J. Seeking to apply the capabilities of the ipEngine as both a platform and sensor controller, the mission that the SMART control interface program sought to validate was the operation of the platform from remote distances.

The program enabled the ipEngine to connect with a Philips 80C552 microcontroller via a serial cable and remotely drive the Lemming vehicle. The Philips
80C552 was sent with the Lemming from Coastal Systems Station as the original controller for the platform and is show in Figure 5.17.

Figure 5.17  Original Philips 80C552 Microcontroller and Robotic Circuitry.

The interface software sent along with the Philips controller provided a specific option of vehicle control that allowed a user to input individual characters into a terminal emulator and drive the Lemming. To quickly test the remote I/O of the ipEngine, the "smartcontrol.c" program was devised to interface directly with the established electrical configuration of the Lemming. This setup would allow the user to transmit character from a wireless telnet session (from a desktop to the ipEngine) and send them to the Phillips controller via a serial connection from the ipEngine. This avoided the necessity of rebuilding the DC motor control hardware interface and other required circuitry to actually drive the Lemming. However, upon finalized testing of the program interface,
an error occurred with the original embedded code loaded into the Philips microcontroller. Since no other copies of the original code for this variant of the Lemming were available, the testing was completed via computer simulation only.

Figure 5.18  SMART Control Remote Interface Modeled in “Smartcontrol.c.”

Figure 5.18 shows the screen capture of the finalized SMART control program interface. The upper left corner shows the futuristic web control page running off port 80. The bottom left corner shows the simple robot driver control user interface running off port 2000. The user has entered several commands into the driver control window and it displays the mission status messages transmitted back from the ipEngine in the same panel. The top right window is the Tera Term telnet session running off port 23. This window simulates a remote debugging of the on-board computer system. Commands are entered here that check the ipEngine’s memory status and its IP address parameters as a simple example of debugging capabilities. The bottom right window is
the serial port connection. The first couple of lines in the serial window show the sequence for the loading and execution of the "smartcontrol.bin" program. The last line of the serial window displays the start of the pKernel shell as annotated by the # prompt. In addition, that final line shows the transmission of the user's command characters that were entered in the robot driver control window (bottom left). The drive commands displayed are only sent to the serial port if they are valid commands. For example, when a user enters a character 'f' the command is interpreted as 'forward' by the ipEngine and the status message is transmitted back to the user in the user interface window (bottom left). In addition, the character 'f' is transmitted to the serial port (bottom right) as well. If the command is unrecognized, the status message declares that the robot is confused and doesn't transmit a character to the serial port. This is the serial port connection that would have gone to the Philips 80C552 microcontroller to drive the Lemming had the program not failed.
Figure 5.19   SMART Control Mission Simulation.
VI. CONCLUSIONS /SMART FUTURE

Autonomous robotic systems will certainly play an increasing role in future conflicts. NPS and its Combat Systems, Science and Technology Curriculum intend to play a significant role in the advancement of the technologies necessary to support this expanding area of warfare. The SMART initiative is a great step toward achieving this goal. The vision of this thesis was to create that ongoing research effort within the CSS&T Curriculum that engages in a forward-looking application of small robotic technology for military employment. The basic foundation of this vision was laid with the successful completion of multiple goals of this research. These included:

- The successful research and procurement of a Lemming tracked vehicle.
- The careful selection of the BSE ipEngine, a robust, network enabled, real-time microcontroller capable of fully autonomous operations.
- The initial investigation into Differential GPS as a future autonomous navigation system.
- The development of the software environment for integration of the ipEngine with the Lemming robotic vehicle.
- The establishment of a baseline set of data transfer techniques including:
  - Wireless networking using TCP/IP socket connections
  - Wireless Serial Communication
  - Wireless programming updates using TFTP.
  - Wireless User interfacing with Telnet and a Web Browser.

Another significant accomplishment of this thesis was the academic engagement with other military research organizations engaged in identical robotic pursuits. The primary two: SPAWAR Systems Center - San Diego and Coastal Systems Station - Panama City, are eager to continue this cooperative research initiative.

The Adaptive Systems Branch of SPAWAR is currently working on technology for a Man Portable Robotic System (MPRS). The MPRS program goal is to develop
lightweight, man-portable mobile robots for operation in urban environments (indoor, outdoor and underground). These small robotic systems are used to evaluate areas in close proximity to the location of the Soldier or Marine and provide valuable tactical sensor data. [Ref. 3] The NPS SMART platform will provide valuable opportunities to work with SPAWAR in future cooperative projects including: autonomous surveillance sensing techniques, chemical detection, and non-lethal warfare.

Coastal Systems Station, Panama City is actively engaged in developing autonomous robotic platforms for expeditionary reconnaissance missions. The SMART initiative will allow NPS to collaborate with them in many areas including: communications and control of multiple platforms, mine detection sensing systems, and techniques of for navigation, communication, and autonomous robotic control. [Ref. 4]

There are many areas with the SMART project that need further research and development and to prepare the Lemming platform for mission experimentation. These include, but are not limited to:

- FPGA integration and analysis
- Precision Navigation requirements and testing
- Web Browser based interface techniques
- Modular sensor architecture development

There are plans developed within the Combat Systems Department to investigate and complete each of these items.

Finally, a cooperative research project will be conducted, within the CSS&T Curriculum, to test a Lemming-mounted seismic sonar to detect buried mines. This sensor research, sponsored by the Office of Naval Research, has been an ongoing research project at NPS [Ref. 20] and requires developmental testing for military utility.
The initial concept would use the Lemming as a platform to test a synthetic aperture seismic sonar array capable of detecting buried mines. Figure 6.1 shows a concept array detecting a buried mine in the sand. The demo would use the SMART platform’s mobility to create the required synthetic array spacing and then analyze data from this promising seismic sensing system.

![Synthetic Aperture Seismic Sonar Array Concept](image)

Figure 6.1  Synthetic Aperture Seismic Sonar Array Concept. [From Ref. 21]

The SMART initiative at NPS has the potential to contribute greatly to the robotic capability of our expeditionary forces. As missions such as surveillance, mine sweeping and chemical detection become increasingly dangerous, continued research in the area of autonomous sensing platforms will be a cornerstone in our preparation for future conflict.
Figure 6.2 Conceptual Autonomous Robotic Mine Reconnaissance Mission. [From Ref. 21]
APPENDIX A. IP ENGINE DIAGRAMS

A. MEMORY LOCATION

The following chart displays the memory address range of

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000.0000 – 00FF.FFFF</td>
<td>16 MB DRAM</td>
</tr>
<tr>
<td>FC00.0000 – FC7F.FFFF</td>
<td>8 MB FPGA SPACE</td>
</tr>
<tr>
<td>FE00.0000 – FE3F.FFFF</td>
<td>4 MB FLASH</td>
</tr>
<tr>
<td>FF00.0000 – FF00.3FFF</td>
<td>MPC823 On-Chip Registers</td>
</tr>
<tr>
<td>FF01.0000 – FF01.0000</td>
<td>FPGA Config Register</td>
</tr>
<tr>
<td>FF02.0000 – FF02.0000</td>
<td>Clock Synth Reg</td>
</tr>
</tbody>
</table>

Table A.1   ipEngine Memory Location. [From Ref. 13]

B. POWER CONFIGURATIONS

<table>
<thead>
<tr>
<th>Power Requirements</th>
<th>Maximum Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option 1. 7-18 V DC</td>
<td>7-18 V DC @ 2 Watts Typical @ 18.26 Watts Max</td>
</tr>
<tr>
<td>Option 2. 5 Volts</td>
<td>+5 V DC ± 5% 800mA Typical</td>
</tr>
<tr>
<td>Option 3. 5 Volts and 3.3 Volts</td>
<td>+5 V DC ± 5% 500mA Typical +3.3 V DC ± 5% 400mA Typical</td>
</tr>
<tr>
<td>Operating Temperature</td>
<td>0°C to 70°C</td>
</tr>
<tr>
<td>Storage Temperature</td>
<td>-40°C to +85°C</td>
</tr>
<tr>
<td>Relative Humidity</td>
<td>10% to 90% (no-condensing)</td>
</tr>
</tbody>
</table>

Table A.2   Power/Operating Condition Requirements. [From Ref. 13]
### Connecting a Power Source

<table>
<thead>
<tr>
<th>Input Power</th>
<th>Why?</th>
<th>JP1 Jumper</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Unregulated 7-18 V DC</strong></td>
<td>To use the on-board switching power supply</td>
<td>Remove all jumpers from JP1</td>
<td>Ground to J3-4 &amp; J3-5, + DC to J3-5 &amp; J3-6</td>
</tr>
<tr>
<td><strong>2. Regulated 5 V DC</strong></td>
<td>If the system already has a regulated 5 V power source, and 7 V DC is not available. The on-board power supply generates 3.3 V from the 5 V input</td>
<td>Connect pins 1-2, Disconnect pins 3-4</td>
<td>Ground to J3-4 &amp; J3-5, +5 V to J3-1, J3-5, J3-6</td>
</tr>
<tr>
<td><strong>3. Regulated 5 V and 3.3 V DC</strong></td>
<td>Use this configuration if the switching noise is a concern, and you wish to completely disable the on-board power supply</td>
<td>Disconnect pins 1-2, Connect pins 3-4</td>
<td>Ground to J3-4 &amp; J3-5, +5V to J3-1, +3.3 V to J3-5, Don’t connect J3-5 &amp; J3-6</td>
</tr>
</tbody>
</table>

Table A.3  Engine Connection Configurations. [From Ref. 13]
C. IPENGINE SPECIFICATIONS

Figure A.1  ipEngine Mechanical Specifications. [From Ref. 13]
### Port IP and Port A Pin Definitions

<table>
<thead>
<tr>
<th>MPC823 Pin Name</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP_B1</td>
<td>FPGA CONF DONE</td>
</tr>
<tr>
<td>IP_B2</td>
<td>FPGA nSTATUS</td>
</tr>
<tr>
<td>PA15/USBRXD</td>
<td>USB RXD</td>
</tr>
<tr>
<td>PA24</td>
<td>USB CE</td>
</tr>
<tr>
<td>PA13</td>
<td>Ethernet RXD</td>
</tr>
<tr>
<td>PA12</td>
<td>Ethernet TXD</td>
</tr>
<tr>
<td>PA9</td>
<td>Serial 2 RXD</td>
</tr>
<tr>
<td>PA8</td>
<td>Serial 2 TXD</td>
</tr>
<tr>
<td>PA7</td>
<td>Ethernet TX Clock</td>
</tr>
<tr>
<td>PA6</td>
<td>Ethernet RX Clock</td>
</tr>
<tr>
<td>PA5</td>
<td>VCLK</td>
</tr>
<tr>
<td>PA4</td>
<td>BCLK</td>
</tr>
</tbody>
</table>

### Port B Pin Definitions

<table>
<thead>
<tr>
<th>MPC823 Pin Name</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB31</td>
<td>LCD_A</td>
</tr>
<tr>
<td>PB30</td>
<td>VCLKEN</td>
</tr>
<tr>
<td>PB29</td>
<td>Ethernet Enable</td>
</tr>
<tr>
<td>PB28</td>
<td>RS-232 Enable</td>
</tr>
<tr>
<td>PB7</td>
<td>I2CSDA</td>
</tr>
<tr>
<td>PB6</td>
<td>I2CSCCL</td>
</tr>
<tr>
<td>PB25</td>
<td>Serial 1 TX</td>
</tr>
<tr>
<td>PB24</td>
<td>Serial 1 RX</td>
</tr>
<tr>
<td>PB23</td>
<td>SDACK1</td>
</tr>
<tr>
<td>PB22</td>
<td>SDACK2</td>
</tr>
<tr>
<td>PB19</td>
<td>LCD_B</td>
</tr>
<tr>
<td>PB18</td>
<td>Ethernet TX Enable</td>
</tr>
<tr>
<td>PB17</td>
<td>LCD_C</td>
</tr>
<tr>
<td>PB16</td>
<td>Ethernet Full-Duplex Enable</td>
</tr>
</tbody>
</table>

Figure A.2   Pin Connections for Port A and Port D. [From Ref 13]
## Port C Pin Definitions

<table>
<thead>
<tr>
<th>MPC823 Pin Name</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC15</td>
<td>DREQ1</td>
</tr>
<tr>
<td>PC14</td>
<td>DREQ2</td>
</tr>
<tr>
<td>PC13</td>
<td>nCONFIG</td>
</tr>
<tr>
<td>PC12</td>
<td>USBSPD</td>
</tr>
<tr>
<td>PC11</td>
<td>USBRXP</td>
</tr>
<tr>
<td>PC10</td>
<td>USBRXN</td>
</tr>
<tr>
<td>PC9</td>
<td>Ethernet Collision</td>
</tr>
<tr>
<td>PC8</td>
<td>Ethernet Carrier Detect</td>
</tr>
<tr>
<td>PC7</td>
<td>USBTXP</td>
</tr>
<tr>
<td>PC6</td>
<td>USBTXN</td>
</tr>
<tr>
<td>PC5</td>
<td>PDN</td>
</tr>
<tr>
<td>PC4</td>
<td>Ethernet Loopback</td>
</tr>
</tbody>
</table>

## Port D Pin Definitions

<table>
<thead>
<tr>
<th>MPC823 Pin Name</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PD15</td>
<td>LD8</td>
</tr>
<tr>
<td>PD14</td>
<td>LD7</td>
</tr>
<tr>
<td>PD13</td>
<td>LD6</td>
</tr>
<tr>
<td>PD12</td>
<td>LD5</td>
</tr>
<tr>
<td>PD11</td>
<td>LD4</td>
</tr>
<tr>
<td>PD10</td>
<td>LD3</td>
</tr>
<tr>
<td>PD9</td>
<td>LD2</td>
</tr>
<tr>
<td>PD8</td>
<td>LD1</td>
</tr>
<tr>
<td>PD7</td>
<td>LD0</td>
</tr>
<tr>
<td>PD6</td>
<td>LCD_AC</td>
</tr>
<tr>
<td>PD5</td>
<td>VSYNC</td>
</tr>
<tr>
<td>PD4</td>
<td>HSYNC</td>
</tr>
<tr>
<td>PD3</td>
<td>LCDCLK</td>
</tr>
</tbody>
</table>

Figure A.3 Pin Connections for Port C and Port D. [From Ref. 13]
Figure A.4 Connector Pin-outs for inEngine. [From Ref. 13]
### J1 - I/O Port

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RX-</td>
</tr>
<tr>
<td>2</td>
<td>RX+</td>
</tr>
<tr>
<td>3</td>
<td>GND</td>
</tr>
<tr>
<td>4</td>
<td>TX-</td>
</tr>
<tr>
<td>5</td>
<td>TX+</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
</tr>
<tr>
<td>7</td>
<td>I2CSCL</td>
</tr>
<tr>
<td>8</td>
<td>I2CSDA</td>
</tr>
<tr>
<td>9</td>
<td>RSTX1</td>
</tr>
<tr>
<td>10</td>
<td>RSRX1</td>
</tr>
<tr>
<td>11</td>
<td>RSTX2</td>
</tr>
<tr>
<td>12</td>
<td>RSRX2</td>
</tr>
<tr>
<td>13</td>
<td>MRST</td>
</tr>
<tr>
<td>14</td>
<td>USBD+</td>
</tr>
<tr>
<td>15</td>
<td>USBD-</td>
</tr>
</tbody>
</table>

Figure A.5  J1 Connector (I/O Port) for the ipEngine. [From Ref. 13]

### J2 Debug Port

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>DSDI</td>
</tr>
<tr>
<td>7</td>
<td>HReset</td>
</tr>
<tr>
<td>6</td>
<td>VCC</td>
</tr>
<tr>
<td>5</td>
<td>DSDO</td>
</tr>
<tr>
<td>4</td>
<td>DSCK</td>
</tr>
<tr>
<td>3</td>
<td>GND</td>
</tr>
<tr>
<td>2</td>
<td>Flash WP</td>
</tr>
<tr>
<td>1</td>
<td>FRZ</td>
</tr>
</tbody>
</table>

Figure A.6  J2 Connector (Debug Port) for the ipEngine. [From Ref. 13]
Connector J3 – DC Power

Connector J3 supplies the DC power for the board.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+5 Volts</td>
</tr>
<tr>
<td>2</td>
<td>+3.3 Volts</td>
</tr>
<tr>
<td>3</td>
<td>GND</td>
</tr>
<tr>
<td>4</td>
<td>GND</td>
</tr>
<tr>
<td>5</td>
<td>DC In</td>
</tr>
<tr>
<td>6</td>
<td>DC In</td>
</tr>
</tbody>
</table>

Notes:

1. When using a 7V-18V DC input on pins 5 and 6, the on-board, switching power supply provides 5 Volts on pin 1 and 3.3 Volts on Pin 2.

2. If you do not provide a single 5 Volt supply to the board, disable the on-board 5 Volt supply via jumper 1 and connect 5 Volts to pins 1, 5, and 6.

3. If you are not using the on-board, switching power supply:
   a. Do not connect pins 5 and 6.
   b. Disable the switching supply via jumper JP1.
   c. Supply 5 Volts to pin 2.
   d. Supply 3.3 Volts to pin 1.

Figure A.7 J3 Connector (DC Power) for the ipEngine. [From Ref. 13]
**Connectors J10 & J11 - Virtual Interface**

### J11 I/O CONNECTOR

<table>
<thead>
<tr>
<th>Function</th>
<th>Pin</th>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>1</td>
<td>2</td>
<td>GND</td>
</tr>
<tr>
<td>VCC5</td>
<td>3</td>
<td>4</td>
<td>VCC5</td>
</tr>
<tr>
<td>ZVCC</td>
<td>5</td>
<td>6</td>
<td>ZVCC</td>
</tr>
<tr>
<td>DCIN</td>
<td>7</td>
<td>8</td>
<td>DCIN</td>
</tr>
<tr>
<td>***</td>
<td>9</td>
<td>10</td>
<td>***</td>
</tr>
<tr>
<td>RSTX1</td>
<td>11</td>
<td>12</td>
<td>RSTX1</td>
</tr>
<tr>
<td>RSTX2</td>
<td>13</td>
<td>14</td>
<td>RSTX2</td>
</tr>
<tr>
<td>IB00</td>
<td>15</td>
<td>16</td>
<td>IB01</td>
</tr>
<tr>
<td>IB02</td>
<td>17</td>
<td>18</td>
<td>IB03</td>
</tr>
<tr>
<td>IB04</td>
<td>19</td>
<td>20</td>
<td>IB05</td>
</tr>
<tr>
<td>IB06</td>
<td>21</td>
<td>22</td>
<td>IB07</td>
</tr>
<tr>
<td>IB08</td>
<td>23</td>
<td>24</td>
<td>IB09</td>
</tr>
<tr>
<td>IB10</td>
<td>25</td>
<td>26</td>
<td>IB11</td>
</tr>
<tr>
<td>IB12</td>
<td>27</td>
<td>28</td>
<td>IB13</td>
</tr>
<tr>
<td>IB14</td>
<td>29</td>
<td>30</td>
<td>IB15</td>
</tr>
<tr>
<td>IB16</td>
<td>31</td>
<td>32</td>
<td>IB17</td>
</tr>
<tr>
<td>IB18</td>
<td>33</td>
<td>34</td>
<td>IB19</td>
</tr>
<tr>
<td>IB20</td>
<td>35</td>
<td>36</td>
<td>IB21</td>
</tr>
<tr>
<td>IB22</td>
<td>37</td>
<td>38</td>
<td>IB23</td>
</tr>
<tr>
<td>IB24</td>
<td>39</td>
<td>40</td>
<td>IB25</td>
</tr>
<tr>
<td>IB26</td>
<td>41</td>
<td>42</td>
<td>IB27</td>
</tr>
<tr>
<td>IB28</td>
<td>43</td>
<td>44</td>
<td>IB29</td>
</tr>
<tr>
<td>IB30</td>
<td>45</td>
<td>46</td>
<td>IB31</td>
</tr>
<tr>
<td>IB32</td>
<td>47</td>
<td>48</td>
<td>IB33</td>
</tr>
<tr>
<td>IB34</td>
<td>49</td>
<td>50</td>
<td>IB35</td>
</tr>
<tr>
<td>IB36</td>
<td>51</td>
<td>52</td>
<td>IB37</td>
</tr>
<tr>
<td>CPUCLK</td>
<td>53</td>
<td>54</td>
<td>VCLK</td>
</tr>
<tr>
<td>UCLK</td>
<td>55</td>
<td>56</td>
<td>BCLK</td>
</tr>
<tr>
<td>MRSF</td>
<td>57</td>
<td>58</td>
<td>RST</td>
</tr>
<tr>
<td>USBD+</td>
<td>59</td>
<td>60</td>
<td>USBD+</td>
</tr>
<tr>
<td>GND</td>
<td>61</td>
<td>62</td>
<td>GND</td>
</tr>
<tr>
<td>RX+</td>
<td>63</td>
<td>64</td>
<td>TX+</td>
</tr>
<tr>
<td>RX-</td>
<td>65</td>
<td>66</td>
<td>TX-</td>
</tr>
</tbody>
</table>

### J10 I/O CONNECTOR

<table>
<thead>
<tr>
<th>Function</th>
<th>Pin</th>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBAT</td>
<td>1</td>
<td>2</td>
<td>GND</td>
</tr>
<tr>
<td>LDO</td>
<td>3</td>
<td>4</td>
<td>LD1</td>
</tr>
<tr>
<td>LD2</td>
<td>5</td>
<td>6</td>
<td>LD3</td>
</tr>
<tr>
<td>LD4</td>
<td>7</td>
<td>8</td>
<td>LD5</td>
</tr>
<tr>
<td>LD6</td>
<td>9</td>
<td>10</td>
<td>LD7</td>
</tr>
<tr>
<td>LD8</td>
<td>11</td>
<td>12</td>
<td>LCD AC</td>
</tr>
<tr>
<td>VSYNC</td>
<td>13</td>
<td>14</td>
<td>HSYNC</td>
</tr>
<tr>
<td>LCDCLK</td>
<td>15</td>
<td>16</td>
<td>LCD A</td>
</tr>
<tr>
<td>LCD B</td>
<td>17</td>
<td>18</td>
<td>LCD C</td>
</tr>
<tr>
<td>12CSDA</td>
<td>19</td>
<td>20</td>
<td>12CSL</td>
</tr>
<tr>
<td>IA00</td>
<td>21</td>
<td>22</td>
<td>IA01</td>
</tr>
<tr>
<td>IA02</td>
<td>23</td>
<td>24</td>
<td>IA03</td>
</tr>
<tr>
<td>IA04</td>
<td>25</td>
<td>26</td>
<td>IA05</td>
</tr>
<tr>
<td>IA06</td>
<td>27</td>
<td>28</td>
<td>IA07</td>
</tr>
<tr>
<td>IA08</td>
<td>29</td>
<td>30</td>
<td>IA09</td>
</tr>
<tr>
<td>IA10</td>
<td>31</td>
<td>32</td>
<td>IA11</td>
</tr>
<tr>
<td>IA12</td>
<td>33</td>
<td>34</td>
<td>IA13</td>
</tr>
<tr>
<td>IA14</td>
<td>35</td>
<td>36</td>
<td>IA15</td>
</tr>
<tr>
<td>IA16</td>
<td>37</td>
<td>38</td>
<td>IA17</td>
</tr>
<tr>
<td>IA18</td>
<td>39</td>
<td>40</td>
<td>IA19</td>
</tr>
<tr>
<td>IA20</td>
<td>41</td>
<td>42</td>
<td>IA21</td>
</tr>
<tr>
<td>IA22</td>
<td>43</td>
<td>44</td>
<td>IA23</td>
</tr>
<tr>
<td>IA24</td>
<td>45</td>
<td>46</td>
<td>IA25</td>
</tr>
<tr>
<td>IA26</td>
<td>47</td>
<td>48</td>
<td>IA27</td>
</tr>
<tr>
<td>IA28</td>
<td>49</td>
<td>50</td>
<td>IA29</td>
</tr>
<tr>
<td>IA30</td>
<td>51</td>
<td>52</td>
<td>IA31</td>
</tr>
<tr>
<td>IA32</td>
<td>53</td>
<td>54</td>
<td>IA33</td>
</tr>
<tr>
<td>IA34</td>
<td>55</td>
<td>56</td>
<td>IA35</td>
</tr>
<tr>
<td>IA36</td>
<td>57</td>
<td>58</td>
<td>IA37</td>
</tr>
<tr>
<td>IA38</td>
<td>59</td>
<td>60</td>
<td>IA39</td>
</tr>
<tr>
<td>IA40</td>
<td>61</td>
<td>62</td>
<td>IA41</td>
</tr>
<tr>
<td>IA42</td>
<td>63</td>
<td>64</td>
<td>IA43</td>
</tr>
<tr>
<td>GND</td>
<td>65</td>
<td>66</td>
<td>GND</td>
</tr>
</tbody>
</table>

*** - Connector Polarization Pins (Male Pins should be removed from mating connector)

Figure A.8  J10 & J11 Connector (Virtual Interface) for the ipEngine. [From Ref. 13]
## Virtual Interface Pin Function

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>PIN</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA00-IA43</td>
<td>J10-21 .. J10-64</td>
<td>FPGA Virtual I/O</td>
</tr>
<tr>
<td>IB00-IB37</td>
<td>J11-15 .. J11-52</td>
<td>FPGA Virtual I/O</td>
</tr>
<tr>
<td>LD0-LD8</td>
<td>J10-3 .. J10-18</td>
<td>LCD/TV Interface</td>
</tr>
<tr>
<td>LCD A,B,C</td>
<td></td>
<td>All/Any pin can be used as a discrete I/O</td>
</tr>
<tr>
<td>LCD DAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCD CLK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I2CSDA</td>
<td>J10-19</td>
<td>I2C Interface</td>
</tr>
<tr>
<td>I2CSI</td>
<td>J10-20</td>
<td></td>
</tr>
<tr>
<td>CPU CLK</td>
<td>J11-53</td>
<td>CPU Clock</td>
</tr>
<tr>
<td>V CLK</td>
<td>J11-54</td>
<td>Clock Synthesizer Output</td>
</tr>
<tr>
<td>USER CLK</td>
<td>J11-55</td>
<td>User Clock (Input)</td>
</tr>
<tr>
<td>MRST</td>
<td>J11-57</td>
<td>Manual Reset Switch Input</td>
</tr>
<tr>
<td>RST</td>
<td>J11-58</td>
<td>Active Low Reset Output</td>
</tr>
<tr>
<td>USB D+, USB D-</td>
<td>J11-59, J11-60</td>
<td>USB Interface</td>
</tr>
<tr>
<td>RX+, RX-</td>
<td>J11-63, J11-65</td>
<td>Ethernet 10BT Receive Pair</td>
</tr>
<tr>
<td>TX+, TX-</td>
<td>J11-64, J11-66</td>
<td>Ethernet 10BT Transmit Pair</td>
</tr>
<tr>
<td>RST1, RSTX2</td>
<td>J11-11, J11-13</td>
<td></td>
</tr>
<tr>
<td>RSRX1, RSRX2</td>
<td>J11-12, J11-14</td>
<td></td>
</tr>
<tr>
<td>VCC 5</td>
<td>J11-3, J11-4</td>
<td>+5 Volts</td>
</tr>
<tr>
<td>VCC</td>
<td>J11-5, J11-6</td>
<td>+3.3 Volts</td>
</tr>
<tr>
<td>DC IN</td>
<td>J11-7, J11-8</td>
<td>DC Power Input 7V-18V</td>
</tr>
<tr>
<td>VB AT</td>
<td>J10-1</td>
<td>Real-Time clock battery supply</td>
</tr>
<tr>
<td>GND</td>
<td>J10-2.65,66</td>
<td>System Ground</td>
</tr>
<tr>
<td></td>
<td>J11-1,2,61,62</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.9  Virtual Interface Pin Functions for the ipEngine. [From Ref. 13]
### Virtual Interface Pin Function

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>PIN</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA00-JA43</td>
<td>J10-21..J10-64</td>
<td>FPGA Virtual I/O</td>
</tr>
<tr>
<td>IB00-JB37</td>
<td>J11-15..J11-52</td>
<td>FPGA Virtual I/O</td>
</tr>
<tr>
<td>LD0-LD8</td>
<td>J10-3..J10-18</td>
<td>LCD/TV Interface</td>
</tr>
<tr>
<td>LCDAC</td>
<td></td>
<td>All/Any pin can be used as a discrete I/O</td>
</tr>
<tr>
<td>LCDAC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSYNC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LCDCLK</td>
<td>J10-19</td>
<td>I2C Interface</td>
</tr>
<tr>
<td>I2CSA</td>
<td>J10-20</td>
<td></td>
</tr>
<tr>
<td>CPUCCLK</td>
<td>J11-53</td>
<td>CPU Clock</td>
</tr>
<tr>
<td>VCLK</td>
<td>J11-54</td>
<td>Clock Synthesizer Output</td>
</tr>
<tr>
<td>UCLK</td>
<td>J11-55</td>
<td>User Clock (Input)</td>
</tr>
<tr>
<td>MRST</td>
<td>J11-57</td>
<td>Manual Reset Switch Input</td>
</tr>
<tr>
<td>RST</td>
<td>J11-58</td>
<td>Active Low Reset Output</td>
</tr>
<tr>
<td>USBP+,USBP-</td>
<td>J11-59,J11-60</td>
<td>USB Interface</td>
</tr>
<tr>
<td>RX+,RX-</td>
<td>J11-63,J11-65</td>
<td>Ethernet 10BT Receive Pair</td>
</tr>
<tr>
<td>TX+,TX-</td>
<td>J11-64,J11-66</td>
<td>Ethernet 10BT Transmit Pair</td>
</tr>
<tr>
<td>RSTX1,RSTX2</td>
<td>J11-11,J11-13</td>
<td></td>
</tr>
<tr>
<td>RSRX1,RSRX2</td>
<td>J11-12,J11-14</td>
<td></td>
</tr>
<tr>
<td>VCC5</td>
<td>J11-3,J11-4</td>
<td>+5 Volts</td>
</tr>
<tr>
<td>VCC</td>
<td>J11-5,J11-6</td>
<td>+3.3 Volts</td>
</tr>
<tr>
<td>DCIN</td>
<td>J11-7,J11-8</td>
<td>DC Power Input 7V-18V</td>
</tr>
<tr>
<td>VBAT</td>
<td>J10-1</td>
<td>Real-Time clock battery supply</td>
</tr>
<tr>
<td>GND</td>
<td>J10-2,65,66</td>
<td>System Ground</td>
</tr>
<tr>
<td></td>
<td>J11-1,2,61,62</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.10  Virtual Interface Pin Functions for the ipEngine. [From Ref. 13]
**Jumper JP1**

Configure jumper JP1 to disable the on-board power supply completely or to disable only the 5 Volt supply.

![JP1 Connector Diagram](image)

**Notes:**

1. Insert jumper across pins 1 and 2 to disable the on-board power supply (both 3.3 Volts and 5 Volts).
2. Insert jumper across pins 3 and 4 to disable only the 5 Volt supply.

*Figure A.11  JP1 Connector to Enable/Disable of on-board Power Supply. [From Ref. 13]*
### D. SOFTWARE DIRECTORIES

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bse/</td>
<td>Root directory for the ipEngine</td>
</tr>
<tr>
<td>ipEngine/</td>
<td>Contains all ipEngine-specific items</td>
</tr>
<tr>
<td>doc/</td>
<td>Documents that describe the ipEngine hardware</td>
</tr>
<tr>
<td></td>
<td>IpEngine Hardware Reference Manual</td>
</tr>
<tr>
<td>fpga/</td>
<td>Files required to configure the Altera FPGA. Contains the subdirectories vhdl and tdf</td>
</tr>
<tr>
<td>pkernel/</td>
<td>Contains all pKernel-specific items (Makefile.top is located here).</td>
</tr>
<tr>
<td>Include/</td>
<td>Header files. The header files define the interface to the kernel. There are also the subdirectories arpa, gcc, net and sys.</td>
</tr>
<tr>
<td>Lib/</td>
<td>Library files</td>
</tr>
<tr>
<td>Doc/</td>
<td>Documents that describe the pKernel</td>
</tr>
<tr>
<td>pkref.pdf</td>
<td>PKernel Programmer’s Guide</td>
</tr>
<tr>
<td>sdkug.pdf</td>
<td>Software Developer’s Kit User Guide</td>
</tr>
<tr>
<td>samples/</td>
<td>Sample programs.</td>
</tr>
<tr>
<td>src/</td>
<td>Source Directory for SMART programs</td>
</tr>
<tr>
<td>download/</td>
<td>Directory to store binary program images for downloading to the ipEngine.</td>
</tr>
<tr>
<td>gnu tools/</td>
<td>The GNU tool set, including executable programs and documentation.</td>
</tr>
<tr>
<td>doc/</td>
<td>Documentation files for the GNU tool set.</td>
</tr>
<tr>
<td>html/</td>
<td>GNU documents in HTML format</td>
</tr>
<tr>
<td>pdf/</td>
<td>GNU documents in PDF format.</td>
</tr>
</tbody>
</table>

Table A.4 Directory Location of Software Components. [From Ref. 9]
APPENDIX B. GLOBAL POSITIONING SYSTEM

The Global Positioning System (GPS) is a worldwide radio-navigation system formed from a constellation of 24 satellites, each orbiting every 12 hours at a height of about 11,000 nautical miles, and their ground stations. Using simple "trilateration" techniques, based on time of flight for uniquely coded spread-spectrum radio signals transmitted by the satellites, GPS uses these "man-made stars" as reference points to calculate positions on earth accurate to a matter of meters. In fact, with advanced forms of GPS you can make measurements to better than a centimeter. Knowing the exact distance of the ground receiver to three of the satellites theoretically allows for calculation of receiver latitude, longitude and altitude. Although conceptually very simple, this design introduces at least four obvious technical challenges. [Ref. 17]

1. Time synchronization between individual satellites and GPS receivers.
2. Precise real-time location of satellite position.
3. Accurate measurement of signal propagation time.
4. Sufficient signal-to-noise ratio for reliable operation in the presence of interference and possible jamming. [Ref. 22]

A. GEOMETRIC VISUALIZATION:

Suppose we measure our distance from three separate satellites and find it to be 11,000, 12,000 and 13,000 miles. If you draw a sphere of radius 11,000 miles around the first and 12,000 miles around the second, the two spheres will intersect in space forming a circle. The intersection of the 13,000 mi sphere, measured from the third satellite, with the other two spheres narrows the position location down to two points on that circle. To decide which one is our true location we could make a fourth measurement, but usually one of the two points is a ridiculous answer, either too far from Earth or moving at an impossible velocity, and can be rejected without a measurement
The estimated ranges to each satellite intersect within a small region when the receiver clock bias is correctly estimated and added to each measured relative range.

Figure B.1   Geometric Intersection of Four Satellite Range Rings. [From Ref. 18]

B.   SATELLITE TRANSMISSIONS:

Each GPS satellite transmits a periodic, pseudo-random digital code on two different carrier frequencies (designated L1 and L2) in the internationally assigned navigational frequency band. The L1 carrier frequency is 1575.42 MHz and carries both a system status message and the pseudo-random code for timing. The L2 carrier frequency is 1227.6 MHz and is used for a more precise military pseudo-random code. [Ref. 17]
C. PSEUDO-RANDOM CODES

There are two types of pseudo-random code. The first is called the Coarse Acquisition (C/A) code. It modulates the L1 carrier. It repeats every 1023 bits and modulates at a 1MHz rate. Each satellite has a unique pseudo-random code. The C/A code is the basis for civilian GPS use. The second pseudo-random code is called the Precise (P) code. It repeats on a seven-day cycle and modulates both the L1 and L2 carriers at a 10MHz rate. This code is intended for military users and can be encrypted. When it's encrypted it's called "Y" code. Since P code is more complicated than C/A it's usually more difficult for receivers to acquire.

There are several good reasons for the complexity of the transmitted pseudo-random code. Since each satellite has its own unique Pseudo-Random Code, this complexity guarantees that the receiver won't accidentally pick up another satellite's signal. It also allows all the satellites to use the same frequency without jamming each other, and makes it more difficult for a hostile force to jam the system. In fact, the Pseudo Random Code gives the DOD a way to control access to the system. However, there's another reason for the complexity of the Pseudo Random Code, a reason that's crucial to making GPS economical. The codes make it possible to use digital "information theory" to "amplify" the GPS signal. This "amplification" will not be expanded upon here, but that is why GPS receivers do not need big satellite dishes to receive the GPS signals. In addition to the pseudo-random code, there is a low frequency signal added to the L1 codes that gives information about the satellite's orbits, their clock corrections and additional system status. [Ref. 17]
D. MEASURING DISTANCE:

To accurately measure a position on the earth, the exact time required for signal propagation from an individual satellite to a receiver station must be accurately measured. This is accomplished by generating an identical pseudo-code sequence in the GPS receiver on the ground and comparing it to the received code from the satellite. The locally generated code is shifted in time during this comparison process until maximum correlation is observed, at which point the induced delay represents the time of arrival as measured by the receiver’s clock. The problem then becomes establishing the relationship between the atomic clock on the satellite and the inexpensive quartz-crystal clock employed in the GPS receiver. [Ref. 17]

E. ACCURATE TIME MEASUREMENT

Accurate time measurement is the key to measuring the receiver’s distance from a satellite. That problem is addressed through the use of atomic clocks onboard each of the satellites. The satellites generate time ticks at a frequency of 10.23 MHz, relying on the vibration period of the cesium atom as a time reference. Each satellite generates the two different transmission frequencies, L1 and L2, by multiplying the cesium-clock time ticks by 154 and 128, respectively. The individual satellite clocks are monitored by dedicated ground tracking stations operated by the Air Force, and continuously advised of their measured offsets from the ground master station clock. High precision in this regard is critical since electro-magnetic radiation propagates at the speed of light, roughly 0.3 meters per nanosecond. The clocks on the receivers, however, are not highly accurate and must be corrected to ensure accurate distance measurement. [Ref. 17]
F. EXTRA MEASUREMENT TO CORRECT TIMING OFFSET

If our receiver's clocks were perfect, then all four of the measured satellite ranges would intersect at a single point. However, when a receiver takes a measurement to a fourth satellite the imperfect GPS clock causes the last measurement to misalign with the other three. When the receiver's computer notices the discrepancy, it realizes that its clock is not perfectly synchronized with universal time. Since any time error will corrupt all other measurements, the receiver calculates a correction factor that it can subtract from all its timing measurements that will cause the position measurements to intersect at a single point. This timing correction automatically adjusts the receiver's clock back into synchronization with universal time. Once it that correction is applied to the rest of its measurements the receiver can calculate a precise position. One consequence of this principle is that any decent GPS receiver will need to have at least four channels so that it can make the four measurements simultaneously. However, for the triangulation calculation to work we not only need to know accurate distance measurements to the satellites, we also need to know the exact position of the satellites in space. [Ref. 17]
The Global Positioning System

Measurements of code-phase arrival times from at least four satellites are used to estimate four quantities: position in three dimensions (X, Y, Z) and GPS time (T).

Figure B.2 Four Satellite Signal Measurements allow for positioning in X, Y, Z, and T. [From Ref. 18].

G. GPS SATELLITE ARRANGEMENT

The GPS satellites are placed into a precise orbit at 11,000 miles above the earth. The spacing of the satellites is arranged so that a minimum of five satellites is in view from every point on the globe. On the ground all GPS receivers have an almanac programmed into their computers that tells them where in the sky each satellite is located. The basic orbits are quite exact, but just to make things perfect the GPS satellites are constantly monitored by the Department of Defense through various GPS ground stations.
These stations monitor the GPS satellites, checking both their operational health and their exact position in space. The master ground station transmits corrections for the satellite's "ephemeris", or orbital, constants and clock offsets back to the satellites themselves. The satellites can then incorporate these updates in the signals they send to GPS receivers. There are five of these ground monitor stations: Hawaii, Ascension Island, Diego Garcia, Kwajalein, and Colorado Springs. They use very precise radar to check each satellite's exact altitude, position and speed. These ephemeris errors are caused by gravitational pulls from the moon and sun and by the pressure of solar radiation on the satellites. The errors are usually very slight but for precise positioning they must be taken into account. Once the DOD has measured a satellite's exact position, they relay that information back up to the satellite itself. The satellite then includes this new corrected position information in the L1 timing signals it is broadcasting. [Ref. 17]

H. OTHER SOURCES OF GPS ERROR

1. Atmospheric Signal Delays

Since GPS positioning depends on the speed of propagation of the satellite signal through the atmosphere, any atmospheric condition that affects the speed of the transmitted signal will induce timing errors. As a GPS signal passes through the charged particles of the ionosphere and then through the water vapor in the troposphere it gets slowed down a bit, and this creates the same kind of error as bad clocks. There are a couple of ways to minimize this kind of error. The first is to use atmospheric models to predict the propagation speed of the signal. Unfortunately atmospheric conditions are rarely consistent and make modeling less than mathematically perfect. The second method to reduce this error is to compare the relative speeds of two different satellite signals. By comparing the delays of the two different carrier frequencies of the GPS
signal, L1 and L2, we can deduce what the atmospheric conditions are, and we can correct for them. Unfortunately this requires a very sophisticated receiver since at the present time only the military has access to the signals on the L2 carrier. Civilian companies have worked around this problem with sophisticated strategies not covered here. [Ref. 17]

2. Multipath error

The whole concept of GPS relies on the idea that a GPS signal flies straight from the satellite to the receiver. Unfortunately, multipath signal propagation results in a barrage of signals arriving at the receiver: first the direct one, then a bunch of delayed reflected ones. This creates a messy signal. Sophisticated receivers use a variety of signal rejection techniques to ensure they only consider the direct signal arriving first to minimize this problem. [Ref. 17]

3. Geometric Errors

Typically there are more satellites available than a receiver needs to fix a position, so the receiver picks a few and ignores the rest. If the receiver picks satellites that are close together in the sky, the intersecting circles that define a position will cross at very shallow angles. This increases the “gray area” or error margin around a position. If it picks satellites that are widely separated, the circles intersect at nearly right angles and minimize the error region. Good receivers determine which satellites will give the lowest amount of this error called Geometric Dilution of Precision (GDOP). [Ref. 17]
Summary of Typical GPS Error Sources

<table>
<thead>
<tr>
<th>Typical Error in Meters</th>
<th>(per satellite)</th>
<th>Standard GPS</th>
<th>Differential GPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satellite Clocks</td>
<td>1.5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Orbit Errors</td>
<td>2.5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Ionosphere</td>
<td>5.0</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>Troposphere</td>
<td>0.5</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Receiver Noise</td>
<td>0.3</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>Multipath</td>
<td>0.6</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>Selective Availability</td>
<td>30</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Typical Position Accuracy

<table>
<thead>
<tr>
<th></th>
<th>Horizontal</th>
<th>Vertical</th>
<th>3-D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50</td>
<td>78</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>2.0</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table B.1 Summary of GPS/DGPS Source Errors. [From Ref. 17]

I. DIFFERENTIAL GPS (DGPS)

Differential GPS is a way to correct the various inaccuracies in the normal GPS system. DGPS can yield measurements accurate to a couple of meters in moving applications and even better in stationary situations. For a standard GPS system each of the timing measurements from the four measurements are figured into the position calculation and the compounded error of each of these signals translates into an overall positioning error. However, the satellites are so far out in space that the distances traveled here on earth are proportionally insignificant. So, if two receivers are fairly close to each other, say within a few hundred kilometers, the signals that reach both of them will have traveled through virtually the same slice of atmosphere, and so will have virtually the same errors. The Differential GPS uses this concept to correct for error by comparing the received signals of two GPS units, one that's stationary and another that's
roving. The stationary receiver is the key because it ties all the satellite measurements into a solid local reference. [Ref. 17]

1. **Eliminating Common Errors**

Differential GPS eliminates all errors that are common to both the reference receiver and the roving receiver. These include everything except multipath errors and any unique receiver errors. The idea behind differential GPS is to have one receiver measure the timing errors and then provide correction information to the receivers that are roving around. That way, virtually all errors can be eliminated from the system, even the Selective Availability error that the DOD puts in on purpose. The idea is simple. Put the reference receiver on a point that's been very accurately surveyed and keep it there. This reference station receives the same GPS signals as the roving receiver, but instead of working like a normal GPS receiver, it attacks the equations backwards by using its known position to calculate timing. It figures out what the travel time of the GPS signals should be, and compares it with what they actually are. The difference is an "error correction" factor. The stationary receiver then transmits this error information to the roving receiver so it can use it to correct its measurements. Since the reference receiver has no way of knowing which of the many available satellites a roving receiver might be using to calculate its position, the reference receiver quickly runs through all the visible satellites and computes each of their errors. Then it encodes this information into a standard format and transmits it to the roving receivers. The roving receivers get the complete list of errors and apply the corrections for the particular satellites they're using. [Ref. 17]
2. **DGPS Error Transmissions**

Error transmissions not only include the timing error for each satellite, they also include the rate of change of that error as well. This enables the roving receiver the ability to interpolate its position between updates. In the early days of GPS, only a few private companies, who had big projects demanding high accuracy, established permanent reference stations. The United States Coast Guard and other international agencies are currently establishing reference stations all over the place, especially around popular harbors and waterways. These stations often transmit on the radio beacons that are already in place for radio direction finding (usually in the 300kHz range). Anyone in the area can receive these corrections and greatly improve the accuracy of their GPS measurements. [Ref. 17]

However, for military applications, we cannot necessarily rely on an established reference station for GPS error corrections. Therefore, the SMART project intends to experiment with various methods of distributing error corrections to multiple platforms. One possible way of doing this would be a wireless Internet connection. For example, consider a fleet of SMART vehicles that we would like to pinpoint on a beachhead with very high accuracy. To obtain this accuracy without equipping each vehicle with expensive differential receivers, an "inverted DGPS" system could be used. With an inverted DGPS system, the SMART robots would be equipped with standard GPS receivers and a transmitter and would send standard GPS positions back to a tracking center. At the tracking center the corrections would be applied to the received positions. It requires a computer to do the calculations and a transmitter to send the data but it
enables accurate positioning for a fleet of vehicles for the cost of one reference station, a computer and a lot of standard GPS receivers. [Ref. 17]
APPENDIX C. MAKEFILES

The knowledge of how to compile and build programs is directed from within GNU Make by a file called the makefile. A makefile code lists each of the source and non-source files, describes their relationship to each other and provides commands for updating, linking, and compiling the finished program. All SMART C code is compiled using these makefiles to direct the automation of the GNU Make utility to create the executable binary program for the robot.

A. MAKEFILE STRUCTURE

Makefiles are organized in a particular manner consisting of code lines called rules and commands. It is these rules and commands that instruct GNU Make on how to link and compile the multi-source code. A makefile may contain other text besides rules, but the simple makefile necessary for SMART programming need only contain rules. Rules can be subdivided into three general parts: targets, dependencies and commands. Here is a common format for a rule:

    target ... : dependencies ...
    command
    ...
    ...

Each rule in a makefile tells the GNU Make utility how to execute a series of commands in order to build a target file from multiple source files. Examples of targets are executable or object files generated from C code source files. Every rule also specifies a list of dependencies of the target file. Dependencies are usually C code text and header files developed by the programmer and are used as inputs to create a target. A typical target is often constructed of several dependency files. Each dependency list
should include all files, whether source files or even other targets, which are required as inputs to the *command* lines. [Ref. 16]

Finally, a *command* is an action that GNU Make carries out. Usually *commands* serve to create the updated *target* file if any of the *dependencies* are changed. Typical *command* lines used in SMART *makefiles* are instructions to the GNU CC compiler to create one type of file (object file) from another (C code text file). A *rule* may have more than one executable *command*, but each must be on its own line and a tab character must begin the line of text in order for GNU Make to recognize it as a *command*.

GNU Make uses *makefiles* to figure out which *target* files need to be compiled by observing each files’ time stamp and determining which of them actually have changed and need to be updated. If a *target* file is newer than all of its *dependencies*, then it is already up to date, and it does not need to be regenerated. If a *target* file is older than its *dependencies* then GNU Make regenerates the *dependencies* first, then updates the *target*. Normally, updating files could be complicated when a *target* has *dependencies* that are defined as *targets* with other *dependencies* etc... However, GNU Make solves this by automatically figuring out the correct order in which to update and build each target file. When you run GNU Make, you can specify particular *targets* to update; otherwise, its simply updates the first *target* listed in the *makefile* and proceeds sequentially through each *target* listed in the file. [Ref. 16]

The following is an less generic example of a *rule* construction:

```
main.o : main.c main.h
    cc -c main.c
```
This rule consists of the target ‘main.o’ (a object file to be created called ‘main.o’) and the dependency files: ‘main.c’ (a text file of C code) and ‘main.h’ (a header file for ‘main.c’). The command that instructs GNU Make on what to do with these files is ‘cc -c main.c.’ This command will direct GNU Make to compile ‘main.c’ into ‘main.o’.

B. SIMPLE MAKEFILE EXAMPLE

Here is a straightforward makefile taken from the GNU Make manual. It describes an executable file called ‘edit’ that depends on eight separate object files. Each of the object files, in turn, depends on eight C source and three separate header files.

```make
edit: main.o kbd.o command.o display.o \
     insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \ 
       insert.o search.o files.o utils.o

main.o : main.c defs.h
       cc -c main.c

kbd.o : kbd.c defs.h command.h
       cc -c kbd.c

command.o : command.c defs.h command.h
           cc -c command.c

display.o : display.c defs.h buffer.h
            cc -c display.c

insert.o : insert.c defs.h buffer.h
           cc -c insert.c

search.o : search.c defs.h buffer.h
           cc -c search.c

files.o : files.c defs.h buffer.h command.h
          cc -c files.c

utils.o : utils.c defs.h
          cc -c utils.c

clean :
    rm edit main.o kbd.o command.o display.o \ 
       insert.o search.o files.o utils.o [Ref. 16]
```

For the example, each long line is split into two lines using backslash-new line. This is observed by GNU Make as one continuous line, but makes it easier for the
programmer to read. The targets include the executable file ‘edit’, and the object files such as ‘main.o’ and ‘kbd.o’. The dependencies are files such as ‘main.c’ and ‘defs.h.’ In fact, each object file is both a target and a dependency. The GNU Make utility commands include: ‘–cc –c main.c’ and ‘cc –c kbd.c.’ These are automatically recognized as instructions to compile the files ‘main.c’ and ‘kbd.c.’ [Ref. 16]

C. HOW GNU MAKE PROCESSES THIS MAKEFILE

When you run GNU Make, you can specify particular targets to update otherwise, GNU Make begins with the first target listed and sequentially works its way down through the makefile updating targets according to the command lines. In organizing the code within a makefile the first target listed is called the default goal. For the SMART programming, the default goals are the executable robot program files that the programmer desires to update and load onto the robot. Listing default goals as the first line in the makefile creates a consistent method of documenting the makefile and aids in debugging. In the simple example above, the default goal is to update the executable program edit; therefore, that rule is placed first.

The actual command used to run this makefile and create the executable file called edit is simply:

```
make
```

It is important to remember that when the command is given within a shell, like the MS-DOS Command Shell, the GNU looks for the makefile within the directory you are in. To ensure that the correct makefile is executed for the program that you desire to compile, keep them together in a unique directory and run GNU make from within that directory.
In the previous 'edit' example, the first instruction GNU Make executes is the
rule is for re-linking edit. However, before Make can fully process this rule, it must
process the rules for each of the files that 'edit' depends on, which in this case are the
object files. Each of these files is processed according to its own rule. These rules are
instructions to update each object file by compiling its source file. The recompilation
must be done if the source file, or any of the header files named as dependencies, is more
recent than the object file, or if the object file does not exist.

Thus, if we change the file 'insert.c' and run Make, it will compile that file to
update 'insert.o' and then link 'edit.' If we change the file 'command.h' and run Make, it
will recompile the object files 'kbd.o', 'command.o' and 'files.o' and then link the file
'edit.' [Ref. 16]

D. SIMPLIFYING MAKEFILES USING VARIABLES

In our example, we had to list all the object files twice in the rule for edit:

```
edit: main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o [Ref. 16]
```

To eliminate the risk of error and simplify the makefile a variable can be used.
Variables allow a text string to be defined once and substituted in multiple places later. It
is standard practice for every makefile to have a variable assigned as a listing of all the
object file names. While the name for that variable is typically: objects, OBJECTS, objs,
OBJJS, obj, or OBJ; for all SMART programs the variable name OBJS will be used to
define the object files (for standarization). A variable line defining a list of object files
may look something like this:
OBJS = main.o kbd.o command.o display.o \ 
insert.o search.o files.o utils.o

Then, each place we want to put a list of the object file names, we can substitute
the variable by writing `$(objects)`. Here is how the simplified `makefile` looks when you
use a variable for the object files:

OBJS = main.o kbd.o command.o display.o \ 
insert.o search.o files.o utils.o
edit : $(OBJS)
cc -o edit $(OBJS)
main.o : main.c defs.h
 cc -c main.c
kbd.o : kbd.c defs.h command.h
 cc -c kbd.c
command.o : command.c defs.h command.h
 cc -c command.c
display.o : display.c defs.h buffer.h
 cc -c display.c
insert.o : insert.c defs.h buffer.h
 cc -c insert.c
search.o : search.c defs.h buffer.h
 cc -c search.c
files.o : files.c defs.h buffer.h command.h
 cc -c files.c
utils.o : utils.c defs.h
 cc -c utils.c
clean :
 rm edit $(OBJS) [Ref. 16]

E. SIMPLIFYING MAKEFILES BY IMPLICIT RULES

It is not necessary to spell out the commands for compiling the individual C
source files into object files because Make will recognize the relationship if properly
formatted. Make has an implicit rule for updating a `.o' file from a correspondingly
named `.c' file using a `cc -c' command. For example, it will use the command `cc -c
main.c -o main.o to compile `main.c' into `main.o'. We can therefore omit the compile commands from the rules for the object files.

By using this shortcut format any `.c' file is also automatically added to the list of dependencies for the target object file. We can therefore omit the `.c' files from the dependencies. Here is the entire example, with both of these changes, and a variable OBJS as suggested above:

```
OBJS = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
$(OBJS)
cc -o edit $(OBJS)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h edit buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
.PHONY : clean
clean :
-rm edit $(OBJS) [Ref. 16]
```

This is how we would write the makefile in actual practice. Because implicit rules are so convenient they are used frequently. Note the use of the goal clean. This is considered a phony target that deletes everything except source files at the request of the user. To execute the clean command, simply type make clean from inside the makefile's directory within the command shell. [Ref. 16]

F. NAMING MAKEFILES

By default, when GNU Make looks for a makefile to execute, it tries the following names, in order: `GNU makefile', `makefile' and `Makefile'. Normally you should call
your makefile either `makefile' or `Makefile'. If you want to use a nonstandard name for your makefiles, you can specify the name with the `-f' or `--file' option. The arguments `-f name' or `--file=name' tell GNU Make to read the file name as a makefiles. If you use more than one `-f' or `--file' option, you can specify several makefiles to execute. All the makefiles are effectively executed in the order specified. The default names `GNU makefile', `makefile' and `Makefile' will not be automatically checked if you specify a `-f' or `--file'. The command to run a non-standard makefile named `hello.mak' from within a MS-DOS shell is:

```
make -f hello.mak
```

G. SAMPLE SMART MAKEFILES

Here is a sample makefile included in the IP Engine software samples. It is located in the folder: BSE\pKernel\samples\hello directory. The file name is `Makefile' and it is written to compile a simple C program for the IP Engine called "hello.c".

```
OBJS = hello.o
include ../..\Makefile.top
all:hello.bin
```

The first line of `Makefile' declares a variable called OBJS and assigns to it the object file `hello.o.' The second line uses a directive called include that instructs GNU Make to suspend reading of `Makefile' and execute a special makefile, called `Makefile.top' before continuing. `Makefile.top' (see Appendix D) is called the `master makefile' and was specifically created for the ipEngine software development package. As shown in Fig G.1, `Makefile.top' is located in the BSE directory, two levels above the sample subdirectory where the `hello.c' code is located.
The location of ‘Makefile.top’ with reference to ‘Makefile’ is defined within the include statement. ‘Makefile.top’ is important because it sets up the GNU Make software to locate all the necessary components it needs to automate the compiling of the SMART program. Therefore, ipEngine programming is designed to have ‘Makefile.top’ included within all the SMART program makefiles. When GNU Make finishes executing the commands of ‘Makefile.top’, it then jumps back into the ‘hello.c’ makefile and finishes executing the code. The final line is a directive all. The all command is a target that directs GNU Make command to create the binary file called ‘hello.bin’ from ‘hello.c.’

Figure G.1 shows the relative locations of the important files on the host computer in order to compile the “hello.c” program. The position of the files within the directories is important because GNU Make has to locate each of them and execute the commands to update, link, compile etc... as directed by the makefile. Typically, the best way to
arrange a `makefile` to compile a program is to locate the `makefile` and the source code in the same directory. Otherwise, GNU Make will have to be instructed where to find the code.

Finally, here is another sample Makefile. This is taken from the SPAWAR Man-Portable Robotic System vehicle. It is called ‘driver.mak’ and controls the compiling of ‘driver.c’. Note the ability to control the type of file to download to the ipEngine. ‘driver.bin’ for RAM operation and ‘driver.zbin’ for FLASH operation.

The command to run ‘driver.mak’ for RAM operation is:

```
make -f driver.mak all
```

The command to run ‘driver.mak’ for RAM operation is:

```
make -f driver.mak auto
```

# SMART Driver makefile.
#
# For RAM operation type:
# make -f driver.mak all
#
# For FLASH operation type:
# make -f driver.mak auto

SMART = netio.o list.o clock.o config.o debug.o
OBJS = driver.o $(SMART)

include ../../../Makefile.top

all: driver.bin

auto: driver.zbin

rbffile.s: smartfpga.rbf
  $(TOOLS)/f2bin rbffile smartfpga.rbf > rbffile.s

For more on `makefiles` see the GNU Make manual [Ref 16] included with the BSE software.
APPENDIX D. MAKEFILE.TOP

.SUFXES: .o .c .s .S

.PRECIOUS: %,x

# base of pkemnel distribution
BASE=$(SEROOT)/pkemel

# directory for target downloads
DOWNLOAD=$(BASE)/download
#vpath %.bin $(DOWNLOAD)
#vpath %.zbin $(DOWNLOAD)

# path to GNUTOOLS .exe's
GCCBIN=$(SEROOT)/gnutools/H-1586-cygwin32/bin

# path to BSE tools
TOOLS=$(GCCBIN)

# target architecture
ARCH2=powerpc-eabi

# target gcc options
GCCMOPTS="-mcps=8660" -nostdinc

# default libraries to link
LLIBS += -lutil
LLIBS += -lnet
LLIBS += -lkern
LLIBS += -lc

# default library dependencies
LIBLIST += $(BASE)/lib/libkem.a
LIBLIST += $(BASE)/lib/libutil.a
LIBLIST += $(BASE)/lib/libc.a
LIBLIST += $(BASE)/lib/libnet.a

# exec names for compiler tools
CXX=$(GCCBIN)/$(ARCH2)-g++ $(GCCMOPTS)
CC=$(GCCBIN)/$(ARCH2)-gcc $(GCCMOPTS)
LD=$(GCCBIN)/$(ARCH2)-ld
AS=$(GCCBIN)/$(ARCH2)-as
AR=$(GCCBIN)/$(ARCH2)-ar
RANLIB=$(GCCBIN)/$(ARCH2)-ranlib
OBJCOPY=$(GCCBIN)/$(ARCH2)-objcopy

# linker command file
LCMD=$(BASE)/lib/ipenginel1.ld
# compile/link flags
DBGFLAGS := -g -O2
LANGFLAGS := -ffunction-sections -fdata-sections
CXXLANGFLAGS := -fno-rtti -fno-exceptions -fvtabe-gc -finit-priority
INCLUDES := -I$(BASE)/include
CFLAGS= $(INCLUDES) $(DBGFLAGS) $(LANGFLAGS) $(CXXLANGFLAGS)
#LFLAGS = -Wl,--gc-sections -Wl,-static
LFLAGS += $(BASE)/lib $(LIBS) $(LIBS) $(LIBS) $(LIBS) $(LIBS) -lgcc -T$(LCMD)
   -nostdlib

# crt0 startup
CRT0 = $(BASE)/lib/$(ARCH2)-crt0.o

# LIBRARY target
LIBRARY=$BASE/lib/$LIBNAME

ALL: all

# build pkernel src
src:
   $(MAKE) -C $(BASE)/src

# build library
$(BASE)/lib/$LIBNAME: $(OBJS)
   $(AR) $(AR) $@ $(OBJS)

# convert executable to bootable binary image
%.zbin: %.x
   $(OBJCOPY) $< -0 binary $*.bin
   gzip -c $*.bin > $*.bin.gz
   $(TOOLS)/mkz $(BASE)/lib/zload.bin $*.bin.gz $@ 4000 ""
   cp $@ $(DOWNLOAD)/$@

# convert executable to binary image
%.bin: %.x
   $(OBJCOPY) $< -0 binary $@
   cp $@ $(DOWNLOAD)/$@

# build executable
%.x: $(OBJS) $(CRTC) $(LIBLIST) $(LCMD)
   $(CC) -fdata-sections -ffunction-sections -o $@ $(CRTC) $(OBJS) $(LFLAGS)
   $(GCCBIN)/$(ARCH2)-size $@

clean:
   rm -f *.o *.x *.map *.bin *.zbin *.bin.gz zimage $(CLEANX)
cleano:
    find . -name "*.o" -print -exec rm {} \\

# generate dependencies
#%.d: %.c
# $(CC) -M $(CFLAGS) $< > $@
#
#include $(OBJS:.o=.d)
APPENDIX E. SOCKETS

A. BACKGROUND

A socket is one of the most fundamental technologies of computer networking. Sockets allow applications to communicate using standard mechanisms built into network hardware and operating systems. A socket is essentially one endpoint of a two-way communication link between two software programs running on the network. Much like the telephone allows one person to speak to another, sockets allow one computer process to speak to another and share information.

Normally, an application or program running on a specific computer and has a socket that is assigned or “bound” to a specific computer port number. The port number is simply the address within the computer that identifies a specific socket connection. This address system is necessary to route packets of information to specific applications on various computer systems. Normally all port numbers below 1024 are reserved for specific computer application. Unless they are being used by another program, socket numbers above 1024 up through 65535 are free for use. The default socket for web browsing is 80 and for telnet applications is 23. Sockets are bi-directional, meaning that either side of the connection is capable of both sending and receiving data. [Ref. 23]

Multiple web browsers simultaneously communicating with a single web server are a great example of the socket connection process, but sockets can also be used to communicate locally (inter-process) on a single computer. The web server/client model helps to visualize what a socket is. On a network the “server computer” listens to the socket port 80 for a “client” to make a connection request. The client knows the hostname, or ip address, of the machine on which the server is running and the port
number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.

![Figure D.1 Connection Request from Client to Server. [From Ref. 24]](image1)

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs to assign a new socket (and consequently a different port number) so that it can continue to listen to the original socket, port 80, for connection requests while tending to the needs of the connected client.

![Figure D.2 Connection Accepted by Server. [From Ref. 24]](image2)

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not necessarily bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running.

The client and server can now communicate by writing to or reading from their sockets. The network hardware can identify the address of the specific application that a data packet needs to be sent when a socket is created and bound to that specific computer.
port number. This TCP protocol layering method of data transfer is used to correctly match the information packet to the application address. This information system directs the accurate transfer of packet of information around a network.

There are many types of sockets. The type of socket used with the ipEngine is the Stream Socket. Stream sockets are reliable two-way communication streams that guarantee if data is sent to a socket in sequential order it will arrive in sequential order at the opposite end. The data will also be error free. Typical applications that use stream sockets are Telnet and Web browsers. [Ref. 23]
APPENDIX F. BSE PROGRAMMING EXAMPLES

All the commands and programs in this appendix are found in the BSE Software Developer’s Kit [Ref. 9]. Included here are some important programming examples that amplify programming techniques applied to SMART programming. More details on the programs and the specific command can be found in the BSE Software Developer’s Kit User’s Guide [Ref. 9] and the pKernel Programmer’s Reference Guide [Ref. 26].

A. IMPORTANT COMMANDS

These are a few basic command that are included in all SMART programming:

- **SysInt()**
  - Runs the initialization for the pKernel

- **StartNetwork()**
  - This starts the network operation on the ipEngine. It starts the following threads:
    - **netTask**
      - This task handles the TCP/IP protocol processing
    - **telnet**
      - The telnet daemon. This is a server task that manages requests for telnet sessions.

- **Pthread_create()**
  - This creates a new thread and provides a thread identification object (referenced by the third argument passed to the function)

B. SAMPLE PROGRAMS

1. Auto Boot

The ‘autoboot’ sample program illustrates the method of creating of a self-loading, compressed image that can be loaded into Flash memory and booted automatically on power-up. The sample code itself is just a simple Hello World example. However, auto boot sequences were used in every SMART program written for this thesis. When executed, the image decompresses itself to RAM and executes the
‘helloworld.c’ program. This is an important technique that is required for all SMART programming.

a. Autoboot Sample Makefile

```
OBJJS = autoboot.o /*creates a variable called OBJJS, defined by the object file autoboot.o*/
include ../..Makefile.top /*includes the makefile.top*/
all: autoboot.zbin /*Default goal is to create a autoboot binary file to copy into Flash, not RAM. Upon start it will decompress into RAM*/
```

b. Auto Boot Hello World

```
#include <stdio.h> /* printf */

int main() {
    sysInit(); /* initialize pKernel */
    printf("autoboot says hello world\n");
}
```

c. Key Commands for Autoboot

- In the command shell simply type make to compile this specific program. All of the SMART programs allow the option to compile the program into RAM or Flash. The typical command in to compile into Flash is make auto (see SMART program examples).

- The command sequence to execute an auto boot within the terminal emulator is:
  ```
  > bload autoboot.zbin FE010000
  > set bootcmd "go FE010000"
  ```
  now just cycle the power on the ipEngine and the program will run.

- To cancel autoboot press escape (Esc) w/in 3 sec of cycling power on the ipEngine.

- Reset the bootcmd within the terminal emulator by typing set bootcmd
2. Command Shell

The command shell allows you to control the ipEngine via a pKernel shell command line interface. The shell can be set up to communicate over the RS-232 serial port connections, or across the network using a terminal emulator. Each of these techniques were demonstrated in the SMART programming programs.

a. Key Commands for Using the pKernel Command Shell

```
startshell (0,1,2);
```

b. Shell Makefile

```
OBJS = shell.o
include ../Makefile.top
all: shell.bin
```

/*creates a variable called OBJS, defined by the object file autoboot.o*/

/*includes the makefile.top*/

/*Default goal is to create a autoboot binary file to copy into Flash, not RAM. Upon start it will decompress into RAM*/
c. **Shell C Program**

```c
#include <stdio.h>

int thread1Entry (int arg) {
    printf("Hello from thread 1, arg = %d\n", arg);
    /* This thread now exits */
}

int main () {
    sysInit (); /* initialize pKernel */
    startNetwork (); /* start network stack */
    startShell (0, 1, 2); /* spawn shell on stdin, out, err */

    /* Create a thread for thread1Entry, pass 5: */
    pthread_create (NULL, NULL, thread1Entry, 5);

    /* The "main" thread will now exit */
}
```

d. **Shell Example Windows**

![Tera Term Telnet Connection (Port 23)](image)

Figure F.2 Tera Term Telnet Connection (Port 23).
Figure F.3  Tera Term Telnet Command Window (pKernel Shell Running).

Figure F.4  Commands to Create a Telnet Session from MS-DOS Window.
3. Sockets

a. **Outline Key Commands for Creating and Using Sockets**

```c
struct sockaddr_in name
int socket (AF_INET, SOCK_STREAM, 0)
name.sin_family = AF_INET;
name.sin_addr.s_addr = INADDR_ANY;
name.sin_port = htons (2000);
bind (sock, &name, sizeof name);
listen (sock, 5);
int msgsock = accept (sock, 0, 0);
pthread_create (NULL, NULL, echoThread, msgsock);
```

b. **BSE Sample Socket Makefile**

```makefile
OBJS = sockets.o  #creates a variable called OBJS, defined
         by the object file autoboot.o*/
include ../Makefile.top  #includes the makefile.top*/
all: sockets.bin  #Default goal is to create a autoboot binary file
to copy into Flash, not RAM. Upon start it will
decompress into RAM*/
```
#include <string.h> /* strlen */
#include <sys/types.h>
#include <sys/socket.h> /* sockaddr_in */
#include <netinet/in.h> /* AF_INET, etc. */
#include <pthread.h> /* pthread_create */

int echoThread (int sockfd) {
    /* Loop reading chars from socket, write echo msg back */
    int m;
    char s[25];
    strcpy (s, "You typed _\n\r");
    m = strlen (s);
    while (1) {
        int n = read (sockfd, &s[10], 1);
        if (n != 1)
            return 0;
        write (sockfd, s, m);
    }
}

int main() {
    int sock;
    struct sockaddr_in name;

    sysInit(); /* initialize pKernel */
    startNetwork(); /* start network stack */

    /* Make the socket, and bind to port 2000: */
    if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0)
        return -1;

    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = htons (2000);

    if (bind (sock, &name, sizeof name))
        return -2;

    /* Loop accepting connections on the socket: */
    listen (sock, 5);
while (1) {
    int msgsock = accept (sock, 0, 0);
    if (msgsock == -1)
        return -3;
    pthread_create (NULL, NULL, echoThread, msgsock);
}
APPENDIX G. SOCKETS TEST

A. SOCKETS TEST OVERVIEW

The socket test program was designed to simulate the possibility of using a standard socket connection to pass information on a network hub. The program "socketstest.c" created a simulated robot command window that accepted user inputs to drive a robot. The commands were simple 1 byte characters (f - forward, b - back, l - left, r - right) that the ipEngine would accept on socket port 2000, check to see if it was a legal command for the robot, and return a movement status back to the user on socket port 2000. The movement responses transmitted by the ipEngine simply stated which direction the robot was moving based on the user's input or, in the case of an unrecognized command, that the ipEngine did not understand the command.

Figure G.1 shows the information routing within the 'socketstest.c' program. The two methods information is passed are via Ethernet on socket ports 2000 and 23 (telnet default). The web page and serial communication are inactive for this program.

Figure G.2 shows the Tera Term command window (connection on port 2000). The user inputs the character commands for the robot and the ipEngine responds with a message simulating the direction the robot would be moving.
Figure G.1  Information Routing for Sockets Test.

Figure G.2  Socket Test Command Window (Port 2000).
B. SOCKET TEST MAKEFILE

OBJS = socketstest.o
include ../..\Makefile.top
all: socketstest.bin
auto: socketstest.zbin

C. SOCKETSTEST. C

#include <string.h>        /* strlen(), strcat() */
#include <sys/types.h>
#include <sys/socket.h>    /* sockaddr_in */
#include <netinet/in.h>    /* AF_INET, etc. */
#include <pthread.h>       /* pthread_create() */

#define IPPORT 2000         /* Defines my socket port*/
#define BACKLOG 5            /* max queue number*/

int cmdThread (int sockfd) { /*Simulate a Loop to read
direction commands for robot */

    int m,u,len;
    char cmmd;
    char feedback[55];
    char dir[40];
    char *welcome = "Welcome to the Robot Driver \n\r";    /*welcome is a pointer to
the welcome string*/

    char *msg = "\n\r The character cmmd is "; /*test message to check if
character is received via the
socket*/

    len = strlen(msg);
    u = strlen (welcome);
    write(sockfd,welcome,u);            /*Writes the welcome message*/
while (1) {
    int n = read (sockfd,&cmmd,1);
    if (n != 1)
        return 0;
    strcpy(feedback, "\n\r Robot is moving ");
    m = strlen (feedback);
    write(sockfd,msg,len);
    write(sockfd,&cmmd,1);

    switch(cmmd)
    {
    case 'f':
        strcpy(dir,"forward");
        break;
    case 'b':
        strcpy(dir,"backward");
        break;
    case 'l':
        strcpy(dir,"left");
        break;
    case 'r':
        strcpy(dir,"right");
        break;
    default:
        strcpy(dir,"confused");
    }
    strcat (feedback, dir);
    m = strlen (feedback);
    write (sockfd,feedback,m);
}
/* reads from socket 1 character*/
/* copies "Robot" string into feedback, not used now*/
/* writes test message to echo cmd back*/
/* echo's cmd character back to user for debug*/
/* checks the cmd variable against directions for robot*/
/* copies the robot direction into dir string*/
/* user typed in something other than f,b,l,r*/
/* places the dir string onto the end of the feedback string*/
/* gets the new length of feedback in order to send it via a socket*/
/* sends the feedback command to the user, telling what the robot is doing*/
int main() {
    int sock;
    struct sockaddr_in ipEngine; /*defines the socket structure, ipEngine*/
    sysInit(); /*initialize pKernel*/
    startNetwork(); /* start network stack*/

    /* Make the socket, and bind to port 2000: */
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) /*sock is the file descriptor returned by socket*/
        return -1;
    ipEngine.sin_family = AF_INET; /*host byte order*/
    ipEngine.sin_addr.s_addr = INADDR_ANY; /*automatically chooses the Engine's IP Address*/
    ipEngine.sin_port = htons(IPPORT); /*short network byte order*/
    if (bind(sock, &ipEngine, sizeof ipEngine)) /*binds socket 2000 to ipEngine*/
        return -2;

    listen(sock, BACKLOG); /* Loops accepting connections on the socket: */
    /*listens on port 2000 for incoming connections*/

    while (1) {
        int msgsock = accept(sock, 0, 0); /*accepts incoming connections on 2000*/

        if (msgsock == -1)
            return -3;
        pthread_create(NULL, NULL, cmdThread, msgsock); /*create the thread that will execute robot commands*/
    }
}
APPENDIX H. SERIAL TEST

A. SERIAL TEST OVERVIEW

The second test program built on the previous “socketstest.c” program and included the additional task of using the serial I/O capability of the ipEngine. Using the same robotic command driver window connected to Port 2000, the serial test program modified what the ipEngine did with the user single character command (f, b, l, r). This time, in addition to transmitting the movement status of the robot to the user on socket port 2000, the program sent the user’s movement command out the serial port as well.

Figure H.1 shows the information routing within the “socketstest.c” program. The two methods information is passed are via Ethernet on socket ports 2000 and 23 (telnet default) and on serial port 1. The web page is inactive for this program.

Figure H.2 shows the Tera Term command window (connection port 2000). Again the user inputs the character commands for the robot and the ipEngine responds with a message simulating the direction the robot would be moving. However, this time in addition to replying on port 2000 with a robot status message the character is transmitted over the serial COM port 1 [Figure H.3] as if it is connected to a DC servomotor controller. Note only valid commands (f, b, l, r) are transmitted to the serial command window.
Robot Interface Application Communication Interface

- Web Page
  - Explorer
  - 80
  - Ethernet

- Robot Driver
  - Tera Term
  - 2000
  - Ethernet
  - ipEngine

- Serial To Motor
  - Tera Term
  - Com 1
  - Ethernet

- Debug Program
  - Tera Term
  - 23
  - Information routing
  - No Information routed

Figure H.1 Information Routing for Serial Test Program.

Figure H.2 Serial Test Robot Command Window (Port 2000).
One significant lesson learned from this program dealt with the reading and transmitting of characters over the serial port. Special attention must be made to how the New-line character is treated. Figure H.4 shows the New-line setting for Tera Term that corresponds to this program.

Figure H.3 Serial Test Tera Term Window (Serial Com 1).

Figure H.4 Tera Term Setup for New Line Recognition.
B. SERIALTEST MAKEFILE

OBJS = serialtest.o
include ../..//Makefile.top
all: serialtest.bin
auto: serialtest.zbin

C. SERIALTEST.C

#include <string.h> /* strlen(), strcat() */
#include <sys/types.h> /* sockaddr_in */
#include <sys/socket.h> /* AF_INET, etc. */
#include <netinet/in.h> /* AF_INET, etc. */
#include <pthread.h> /* pthread_create() */
#include <stdio.h>

#define IPPORT 2000 /* Defines my socket port */
#define BACKLOG 5 /* queue number */

int msgsock, pfd;
char *dnames[] = {
    "",
    "device://smc/0", /* serial port 1 */
    "device://smc/1", /* serial port 2 */
};

int cmdThread (int sockfd) {
    /* Simulate a Loop to read direction commands for robot */

    int m,u,len;
    char cmdm;
    char feedback[55];
    char lbuf[64];
    char dir[40];

    FILE *sockf = fdopen(sockfd,"r");
    char *welcome = "Welcome to the Robot Driver \n\r"; /* welcome is a pointer to the welcome string */
char *msg = "\n\r The character cmd is "; /* test message to check if character is received via the socket*/

len = strlen(msg);

u = strlen(welcome);
write(sockfd,welcome,u); /* Writes the welcome message*/

while (1) {
    int n = read(sockfd,&cmd,1); /* reads from socket 1 character*/
    if (n != 1)
        return 0;
    strcpy(feedback, "\n\r Robot is moving "); /* copies "Robot" string into feedback, not used now*/
    m = strlen(feedback);
    write(sockfd,msg,len); /* writes test message to echo cmd back*/
    write(sockfd,&cmd,1); /* echo's cmd character back to user for debug*/
    switch(cmd) /* checks the cmd variable against directions for robot*/
    {
        case 'f':
            {strcpy(dir,"forward"); /* copies the robot direction into dir string*/
            write (pfd, &cmd, 1); /* sends the direction character out port pfd, which is #1*/
            break;}
        case 'b':
            {strcpy(dir,"backward"); /* sends the direction character out port pfd, which is #1*/
            write (pfd, &cmd, 1);
            break;}
        case 'l':
            {strcpy(dir,"left"); /* sends the direction character out port pfd, which is #1*/
            write (pfd, &cmd, 1);
            break;}
        case 'r':
            {strcpy(dir,"right"); /* sends the direction character out port pfd, which is #1*/
            write (pfd, &cmd, 1);
            break;}
    }
}
break;
    default:
        strcpy(dir,"confused"); /*user typed in something other than f,b,l,r*/
    }
    strcat(feedback, dir); /*places the dir string onto the end of the feedback string*/
    m = strlen(feedback); /*gets the new length of feedback in order to send it via a socket*/
    write(sockfd,feedback,m); /* sends the feedback command to the user, telling what the robot is doing*/
}
}

/***************************/

portThread (int portid) /*thread to create serial port/s */
{
    int length, e, num, i;
    char active[256];
    char buf;
    char newline = 10;
    pfd = open(dnames[1]); /*opens port*/
    if (pfd<O) {return -1;}
    /*pfd is an int, read returns a -1 if unable to open port*/
    sprintf(active,"n port %d active\n\r",pfd); /*stores quotation into variable active*/
    write(pfd,&active,strlen(active));
    write(sockfd,buf,strlen(buf)); /*read characters from port 1, write them to socket*/
    while (1) {
        read(pfd, &buf, 1);
        write(pfd, &buf, 1);
        /*read into buf a string from port 1*/
        /*write buf to port 1*/
        if (buf==13)
            write(msgsock,newline,1);
        /*write buf to socket 2000*/
        write(msgsock, &buf, 1);
    }
    sched_yield();
}

/***************************/
int main() {
    int sock;
    struct sockaddr_in ipEngine; /*defines the socket structure, ipEngine*/
    sysInit(); /*initialize pKernel*/
    startNetwork(); /* start network stack*/

    /* Make the socket, and bind to port 2000: */
    if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0) /*sock is the file descriptor returned by socket*/
        return -1;
    ipEngine.sin_family = AF_INET; /*host byte order*/
    ipEngine.sin_addr.s_addr = INADDR_ANY; /*automatically chooses the Engine's IP Address*/
    ipEngine.sin_port = htons (IPPORT); /*short network byte order*/
    if (bind (sock, &ipEngine, sizeof(ipEngine)) /*binds socket 2000 to ipEngine*/
        return -2;
    pthread_create (NULL, NULL, portThread, 1); /*thread to create port*/

    /* Loop accepting connections on the socket: */
    listen (sock, BACKLOG); /*listens on port 2000 for incoming connnections*/

    while (1) {
        msgsock = accept (sock, 0, 0); /*accepts incoming connections on 2000*/
        if (msgsock == -1)
            return -3;
        pthread_create (NULL, NULL, cmdThread, msgsock); /*create the thread that will execute robot commands*/
    }
}
APPENDIX I. WEB PAGE TEST

A. WEB PAGE TEST OVERVIEW

The partial program, ‘webserv.c’, investigated the ability to successfully load a simulated robotic control web page onto the ipEngine. Using the “webserv.c” sample program supplied by BSE, a modified index page was loaded onto the ipEngine Apache web server to demonstrate the possibility of a web control interface for the SMART platform. Figure I.1 shows the information routing within the ‘sockettest.c’ program. The two methods of information control are initiated via Ethernet on socket ports 80 (standard for web browsers) and 23 (telnet default). The port 2000 and serial COM 1 are inactive at this time.

![Diagram of information routing for web page test](image)

Figure I.1 Information Routing for Web Page Test.
Hot Damn It Worked!

SMART ROBOT CONTROLLER

Figure 1.2 SMART Control Simulated Web Page.

B. KEY INSTRUCTIONS FOR USING THE APACHE WEB SERVER

- To modify a index page
  - Save your html page in the htdocs directory
  - include all the necessary Apache files (filesys folder, apache...) in the directory you are programming in
  - ensure when the makefile compiles the program that the filesys.zip is recreated. Otherwise type: make clean fs filesys.zip and retry make (Figure 1.3)
  - Figure 1.4 shows the filesys.zip that is loaded onto the ipEngine
Figure I.3  Remaking the Filesys.zip.

Figure I.4  List of files in Filesys.zip.
C. WEBPAGE MAKEFILE

OBJS = webserv.o filesys.o
include .././Makefile.top
LLIBS += -lapache
all: webserv.bin

# Create assembly file object from zipped filesystem:
filesys.s: filesys.zip
$(TOOLS)/f2bin filesys filesys.zip > filesys.s

# Create zipped filesystem.
# If you add FORCE as a dependencey of filesys.zip
# the zip file will be rebuilt everytime you run make
filesys.zip:
  (cd filesys; $(TOOLS)/zip -r ../filesys.zip .)

# After you have changed the file system you can
# type "make fs" to recreate the zip file.
fs: cleanfs filesys.zip

cleanfs: FORCE
  rm filesys.zip

FORCE:

D. WEBPAGE.C

#include <stdio.h> /* sprintf() */

extern char filesys[]; /* memory block holding filesys*/
extern int filesys_size; /* size of block */

int main() {
  char name[255];
sysInit(); /* initialize pKernel */
startNetwork(); /* start network stack */

/* Unpack the web server's file system: */
sprintf (name, "mimage://%x/%d/**", filesys, filesys_size);
unpackzip (name);
startShell(0,1,2);
startApache ("/apache"); /* start web server */
printf("oops apache exited\n");
}
APPENDIX J. SMART CONTROL TEST

A. SMART CONTROL TEST OVERVIEW

The final challenge was to integrate all three of these test programs into one single SMART control program called "smartcontrol.c." The intent was to show the capabilities that the ipEngine has and therefore, the flexibility this gives the user. When the program was executed the ipEngine was able to communicate via four different multi-threaded methods at the same time. These methods were:

- Terminal emulator telnet session on port 2000
- Terminal emulator default telnet session on port 23
- Web Browser connection on port 80
- Serial Communication on serial port 1.

The "smartcontrol.c" program written for the control interface simulation takes the previous "serialtest.c" program and includes the web server addition. Figure G.1 shows the information routing within the 'socketstest.c" program. The all methods are used to pass information with the ipEngine.
Figure J.1  Information Routing for Web Page Test.

Figures J.2 through J.5 show the individual command window interfaces that interact with the ipEngine.

Figure J.2  TeraTerm Window (Serial Port 1).
The character cmd is f
Robot is moving forward
The character cmd is b
Robot is moving backward

Figure J.3 Tera Term Window (Socket Port 2000).

Figure J.4 Tera Term Telnet Window (Socket Port 23).
Hot Damn It Worked!

SMART ROBOT CONTROLLER

Figure J.5 SMART Control Page (Socket Port 80).

Figure J.6 shows the entire SMART control program interface, each of the screens captured on the desktop.
Figure J.6 Four Panel Display of Each Communication Window.

The upper left corner shows the futuristic web control page running off port 80. The bottom left corner shows the simple robot driver control user interface running off port 2000. The user has entered several commands into the driver control window and it displays the mission status messages transmitted back from the ipEngine in the same panel. The top right window is the Tera Term telnet session running off port 23. This window simulates the remote debugging of the on-board system. Here the user has entered commands to check the ipEngine’s memory status and its IP address parameters as an example of debugging possibilities. The bottom right window is the serial COM port 1 connection. The first couple of lines in the serial window show the loading and execution of the “smartcontrol.bin” program. The last line of the serial window displays
the start of the pKernel shell as annotated by the # prompt. In addition, that final line shows the transmission of user’s command characters that were entered in the robot driver control window. The drive commands displayed are only sent to the serial port if they are valid commands.

**B. SMARTCONTROL MAKEFILE**

```
OBJS = smartcontrol.o filesys.o
include ../../Makefile.top
LLIBS += -lapache
all: smartcontrol.bin

# Create assembly file object from zipped filesystem:
filesys.s: filesys.zip
   $(TOOLS)/f2bin filesys filesys.zip > filesys.s

# Create zipped filesystem.
# you add FORCE as a dependency of filesys.zip
# the zip file will be rebuilt every time you run make
filesys.zip:
   (cd filesys; $(TOOLS)/zip -r ../filesys.zip .)

# After you have changed the file system you can
# type "make fs" to recreate the zip file.
fs: cleanfs filesys.zip

cleanfs: FORCE
   rm filesys.zip

FORCE:
```

152
C. SMARTCONTROL.C

#include <string.h>    /* strlen(), strcat() */
#include <sys/types.h>
#include <sys/socket.h> /* sockaddr_in */
#include <netinet/in.h> /* AF_INET, etc. */
#include <pthread.h>   /* pthread_create() */
#include <stdio.h>     /* sprintf() */

extern char filesys[]; /* memory block holding filesys */
extern int filesys_size; /* size of block */

#define IPPORT 2000       /* Defines my socket port */
#define BACKLOG 5          /* queue number */
char name[255];
int msgsock, pfd;
char *dnames[] = {
    "", /* serial port */
    "device://smc/0","device://smc/1", /* serial port */
};

int cmdThread (int sockfd) {
    /* Simulate a Loop to read direction commands for robot */
    int m,u,len;
    char cmdm;
    char feedback[55];
    char lbuf[64];
    char dir[40];

    FILE *sockf = fdopen(sockfd,"r");
    char *welcome = "Welcome to the Robot Driver \n\r"; /* welcome is a pointer to the welcome string */

    char *msg = "\n\r The character cmdm is "; /* test message to check if character is received via the socket */

    len = strlen(msg);
    u = strlen(welcome);
    write(sockfd,welcome,u); /* Writes the welcome message */
}

153
// pfld = open(dnames[1]); /*opens port 1*/
// if (pfld<0) {return -1;} /*pfld is an int, read returns a -1 if unable
to open port*/

// portf = fdopen(pfld,"r+");
// fgets(&lbuf,sizeof(lbuf),sockf);
// fputs(&lbuf,sockf);

while (1) {
    int n = read (sockfd,&cmmd,1); /*reads from socket 1 character*/
    if (n != 1)
        return 0;
    strcpy(feedback, "\n\tRobot is moving "); /*copies "Robot" string into
feedback, not used now*/
    m = strilen (feedback);
    write(sockfd,msg,len); /*writes test message to echo cmd
    back*/
    write(sockfd,&cmmd,1); /*echo's cmd character back to user for
    debug*/
    switch(cmmd) /*checks the cmd variable against
directions for robot*/
    {
    case 'f':
        {strcpy(dir,"forward"); /*copies the robot direction into dir
        string*/
         write (pfld, &cmmd, 1); /*sends the direction character out port
         pfld, which is #1*/
         break;}
    case 'b':
        {strcpy(dir,"backward"); /*sends the direction character out port
         pfld, which is #1*/
         write (pfld, &cmmd, 1);
         break;}
    case 'l':
        {strcpy(dir,"left"); /*sends the direction character out port
         pfld, which is #1*/
         write (pfld, &cmmd, 1);
         break;}
    case 'r':
        {strcpy(dir,"right"); /*sends the direction character out port
         pfld, which is #1*/
         break;}
}
write (pfdf, &cmd, 1);  /*sends the direction character out port
pfd, which is #1*/
break;
}
default:
strncpy(dir,"confused");  /*user typed in something other than
f,b,l,r*/
}
strcat(feedback, dir);  /*places the dir string onto the end of the
feedback string*/
m = strlen (feedback);  /*gets the new length of feedback in order
to send it via a socket*/
write (sockdfd,feedback,m);  /* sends the feedback command to the
user, telling what the robot is doing*/
}
}

personaThread (int portid)  {  /*thread to create serial port/s */
  int length, e, num, i;
  char active[256];
  char buf;
  // char newline = 10;
  pfd = open(dnames[1]);  /*opens port 1*/
  if (pfd<0) {return -1;}  /*pfd is an int, read returns 
a-1 if unable
to open port*/
  sprintf(active,"\n port %d active\n\r",pfd);  /*stores quotation into variable
active*/
write(pfd,&active,strlen(active));
// write(sockfd,buf,strlen(buf));
while (1) {
  read(pfd, &buf, 1);  /*read into buf a string from port 1*/
  write(pfd, &buf, 1);  /*write buf to port 1*/
  // if (buf==13)
  // write(msgsock,newline, 1);
  write(msgsock, &buf, 1);  /*write buf to socket 2000*/
}
sched_yield();
}
apacheThread(char filename[255]) {
    /* Unpack the web server's file system: */
    sprintf(name, "mimage://%x/%d/", filesys, filesys_size);
    unpackzip (name);
    startShell(0, 1, 2);
    startApache("/apache"); /* start web server */
    printf("oops apache exited\n");
}

int main() {
    int sock;
    struct sockaddr_in ipEngine;

    sysInit(); /* initialize pKernel */
    startNetwork(); /* start network stack */

    pthreadcreate (NULL, NULL, *(*apacheThread), name);

    /* Make the socket, and bind to port 2000: */
    if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0) /*sock is the file
descriptor returned by
socket*/
        return -1;

    ipEngine.sin__family = AF_INET; /*host byte order*/
    ipEngine.sin_addr.s_addr = INADDR_ANY; /*automatically chooses the
Engine's IP Address*/
    ipEngine.sin_port = htons (IPPORT); /*short network byte order*/

    if (bind (sock, &ipEngine, sizeof ipEngine)) /*binds socket 2000 to ipEngine*/
        return -2;

    pthreadcreate (NULL, NULL, portThread, 1); /*thread to create port*/

    /* Loop accepting connections on the socket: */
    listen (sock, BACKLOG); /*listens on port 2000 for
incoming connections*/
while (1) {
    msgsock = accept (sock, 0, 0); /*accepts incoming connections on 2000*/
    if (msgsock == -1)
        return -3;
    pthread_create (NULL, NULL, cmdThread, msgsock); /*create the thread that will execute robot commands*/
}


LIST OF REFERENCES


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center ................................................................. 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library ......................................................................................... 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Director, Training and Education ................................................................. 1
   MCCDC, Code C46
   1019 Elliot Road
   Quantico, VA 22134-5027

4. Director, Marine Corps Research Center .................................................. 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

5. Marine Corps Representative ................................................................. 1
   Naval Postgraduate School
   Code 037, Bldg. 330, Ingersoll Hall, Room 116
   555 Dyer Road
   Monterey, CA 93943

6. Marine Corps Tactical Systems Support Activity ........................................ 1
   Technical Advisory Branch
   Attn: Librarian
   Box 555171
   Camp Pendleton, CA 92055-5080

7. Richard M. Harkins, Code PH/HR ................................................................. 2
   Department of Physics
   Naval Postgraduate School
   Monterey, CA 93943-5101

8. Thomas J. Hofler, Code PH/HF ................................................................. 1
   Department of Physics
   Naval Postgraduate School
   Monterey, CA 93943-5101
9. Chairman, Code PH
   Department of Physics
   Naval Postgraduate School
   Monterey, CA 93943-5101

10. Curricular Officer, Code 34
    Engineering and Technology
    Naval Postgraduate School
    Monterey, CA 93943-5107

11. Mr. William Warner Ferry
    1006 Parkstead
    Seabrook, TX 77586

12. Capt. Todd W. Ferry
    Weapons and Systems Engineering Dept.
    United States Naval Academy
    105 Maryland Ave.
    Annapolis, MD 21402

13. Mr. Robin T. Laird (PL-SS)
    SPAWARSYS CEN D371
    53406 Woodward Rd. Rm. 108
    San Diego, CA 92152-7383

14. Mr. Chuck Bernstein
    Coastal Systems Station R23
    6703 W. Hwy 98
    Panama City, FL 32407-7001

15. Mr. Xiaoping Yun, Code EC/YX
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, CA 93943