Project #: E-21-T12  Cost share #: E-21-319  Rev #: 0  
Center #: R6583-T12  Center shr #: F6583-T12  OCA file #: 128  
Contract#: F30602-88-D-0025-0012  Mod #:  
Prime #:  
Subprojects ?: N  Main project #:  

Project unit:  EE  Unit code: 02.010.118  
Project director(s):  PARIS D T  EE  (404)894-2902  

Sponsor/division names: AIR FORCE / GRIFFISS AFB, NY  
Sponsor/division codes: 104 / 023  

Award period:  890120 to 891231 (performance) 900130 (reports)  
Sponsor amount  
Contract value  99,700.00 
Funded  75,000.00  
Cost sharing amount  8,334.00  

Does subcontracting plan apply ?: Y  

Title: FREQUENCY DOMAIN SIGNAL PROCESSING  

PROJECT ADMINISTRATION DATA  
OCA contact: Brian J. Lindberg  894-4820  
Sponsor technical contact  
STEVEN C. TYLER  
DEPARTMENT OF THE AIR FORCE  
ROME AIR DEVELOPMENT CENTER/DCCD  

Sponsor issuing office  
GERARD J. BROWN/PKRM  
(315)330-7060  
ROME AIR DEVELOPMENT CENTER  
DIRECTORATE OF CONTRACTING (PKRM)  
GRIFFISS AFB, NY 13441-5700  

Security class (U,C,S,TS) : U  
Defense priority rating : DO-A7  
Equipment title vests with: Sponsor  
NONE PROPOSED OR ANTICIPATED  

ONR resident rep. is ACO (Y/N): Y  
GOVT supplemental sheet  
GIT  

Administrative comments -  
DELIVERY ORDER PARTIALLY FUNDS TASK C-8-2402 (RENSSELAER POLYTECHNIC INSTITUTE) THROUGH 9/30/89.
NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 01/16/91

**Project No.** E-21-T12 **Center No.** R6583-T12

**Project Director** JOY E B **School/Lab** ELEC ENGR

**Sponsor** AIR FORCE/GRIFFISS AFB, NY

**Contract/Grant No.** F30602-88-D-0025-0012 **Contract Entity** GTRC

**Prime Contract No.**

**Title** FREQUENCY DOMAIN SIGNAL PROCESSING

Effective Completion Date 900630 (Performance) 900730 (Reports)

<table>
<thead>
<tr>
<th>Closeout Actions Required</th>
<th>Y/N</th>
<th>Submitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Final Invoice or Copy of Final Invoice</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Final Report of Inventions and/or Subcontracts</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Government Property Inventory &amp; Related Certificate</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Classified Material Certificate</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Release and Assignment</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

**Comments**

Subproject Under Main Project No. ________________

Continues Project No. ________________

Distribution Required:

- Project Director **Y**
- Administrative Network Representative **Y**
- GTRI Accounting/Grants and Contracts **Y**
- Procurement/Supply Services **Y**
- Research Property Management **Y**
- Research Security Services **Y**
- Reports Coordinator (OCA) **Y**
- GTRC **Y**
- Project File **Y**
- Other **N**

**NOTE:** Final Patent Questionnaire sent to PDPI.
A. TECHNICAL PROGRESS ACHIEVED ON EFFORT:

The Amiga computer systems have been purchased and installed. Lattice "C" programming language has been installed on the systems.

A means for passing data between the computers via the serial port has been established. The structure of the frequency domain simulator has been defined and broken into a number of programming tasks.
B. TRAVEL:

None

C. PRESENTATIONS AND PUBLICATIONS:

A progress report was presented to visiting RADC monitors on 6/1/89

D. LEVEL OF EFFORT BY EACH CONTRIBUTOR (IN MAN-MONTHS OR MAN-HOURS)

Gary Saulnier 67 hours
Pankaj Das 16 hours
PERIOD COVERED: 6/1/89 - 9/30/89

TITLE: Frequency Domain Signal Processing

PRINCIPAL INVESTIGATOR: Gary Saulnier

INSTITUTION: Rensselaer Polytechnic Institute

OTHER PARTICIPANTS AND TITLES: Pankaj Das, Co-Investigator
Charles Pateros, Ph.D. Student
Joseph Pennisi, Master's Student
John Korecki, Undergraduate

A. TECHNICAL PROGRESS ACHIEVED ON EFFORT:

A spread spectrum frequency domain simulator was implemented in "c" programming in the Amiga 2500 personal computers. The simulator has the following characteristics:

1. Modulator: Direct Sequence or Cyclic Code Shift Spreading

2. Channel: Additive White Gaussian Noise (AWGN), multiple tone jammers, swept tone jammers.

3. Demodulator: Frequency Domain Processing (with jammer suppression) or Time Domain Correlator

4. Parameter Selection: All parameters are selected via menus. Signal parameters are selected prior to execution, channel and demodulator parameters are selectable and changeable during execution.

5. Display: Four system spectra or waveforms can be displayed simultaneously. These are selected from menus and can be changed during execution.
B. TRAVEL:

C. PRESENTATIONS AND PUBLICATIONS:

The simulator was demonstrated at RADC 9/27/89

D. LEVEL OF EFFORT BY EACH CONTRIBUTOR (IN MAN-MONTHS OR MAN-HOURS)

<table>
<thead>
<tr>
<th>Name</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>G. Saulnier</td>
<td>180 hrs</td>
</tr>
<tr>
<td>P. Das</td>
<td>216 hrs</td>
</tr>
<tr>
<td>C. Pateros</td>
<td>127 hrs</td>
</tr>
<tr>
<td>J. Pennisi</td>
<td>550 hrs</td>
</tr>
<tr>
<td>J. Korecki</td>
<td>250 hrs</td>
</tr>
</tbody>
</table>
PERIOD COVERED:  10/1/89 - 12/31/89

TASK NUMBER:  C-8-2402

TITLE:  FREQUENCY DOMAIN SIGNAL PROCESSING

PRINCIPAL INVESTIGATOR:  Gary J. Saulnier

INSTITUTION:  Rensselaer Polytechnic Institute

OTHER PARTICIPANTS AND TITLES:  Pankaj Das  Co-Investigator
                                  Charles Pateros  Ph.D. Student
                                  Joseph Pennisi  Masters Student

A. TECHNICAL PROGRESS ACHIEVED ON EFFORT:

The development of the second generation spread spectrum simulator was
started. This simulator is written in "C" programming language and runs on
a Commodore Amiga 2500. The emphasis is on

1) Modularity - New functions will be easy to implement and add to the
   simulation

2) Portability - Simulator code will be compatable with other computers
   such as Sun Microsystems products.

The simulation is configured prior to execution by creating a file which
provides the system structure. Creation of this file will eventually be
performed by a graphical interface.
B. TRAVEL:

C. PRESENTATIONS AND PUBLICATIONS:

D. LEVEL OF EFFORT BY EACH CONTRIBUTOR (IN MAN-MONTHS OR MAN-HOURS)

  G. Saulnier  39 hrs  
  P. Das  8 hrs  
  C. Pateros  35 hrs  
  J. Pennisi  240 hrs
PERIOD COVERED: 1/1/90 - 3/31/90

TASK NUMBER: C-8-2402

TITLE: Frequency Domain Signal Processing

PRINCIPAL INVESTIGATOR: Gary J. Saulnier

INSTITUTION: Rensselaer Polytechnic Institute

OTHER PARTICIPANTS AND TITLES: Pankaj Das Co-Investigator
Joseph Pennisi Master's Student
Charles Pateros Ph.D. Student

A. TECHNICAL PROGRESS ACHIEVED ON EFFORT:

The problem of signal flow in the spread spectrum simulator was investigated and several possible solutions were identified. The goal is to implement data structures which allow new functions to be added to the simulation with a minimum amount of effort while keeping the simulator structure as simple as possible.
B. TRAVEL:

C. PRESENTATIONS AND PUBLICATIONS:

D. LEVEL OF EFFORT BY EACH CONTRIBUTOR (IN MAN-MONTHS OR MAN-HOURS)

G. Saulnier  20 hrs
J. Pennisi   160 hrs
C. Pateros   6 hrs
## Contract Funds Status Report (DD Form 1586)

**Contract Number:** F30602-88-D-0025  
**Quarter:** May-Jun '88

<table>
<thead>
<tr>
<th>Category</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Current Quarter Funding</strong></td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>Current Quarter Expenditures</strong></td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>Contract Ceiling</strong></td>
<td>$4,200,000.00</td>
</tr>
<tr>
<td><strong>Funding to Date</strong></td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>Pending Commitments</strong></td>
<td>$766,000.00</td>
</tr>
<tr>
<td><strong>Available Funding</strong></td>
<td>$3,434,000.00</td>
</tr>
<tr>
<td><strong>Funding to Date</strong></td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>YTD Expenditures</strong></td>
<td>$0.00</td>
</tr>
<tr>
<td><strong>Outstanding Expenditures</strong></td>
<td>$0.00</td>
</tr>
</tbody>
</table>

### Pending Commitments

- C-8-2120 Westinghouse/Beaudet: $56,000.00
- C-8-2129 Rensselaer/Das: $100,000.00
- E-8-7066 Univ of Penn/Steinberg: $100,000.00
- E-8-7124 Boston College/McFadden: $35,000.00
- E-8-7125 Brandeis Univ/Henchman: $23,000.00
- E-8-7126 Penn State/Castleman: $22,000.00
- A-8-1631 Univ of Penn/Steinberg: $100,000.00
- B-8-3617 Ga Washington Univ/Meltzer: $100,000.00
- B-8-3618 Ga Washington Univ/Berkovich: $100,000.00
- C-8-2492 Ga Tech/Smith: $50,000.00
- A-8-1203 Ga Tech/Hughes: $80,000.00

**Total Pending:** $766,000.00
CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JUL-SEPT '88

**CURRENT QUARTER FUNDING**  
\[
\begin{array}{ll}
\text{DO} & \text{Funding} \\
0001 & 56,000 \\
0002 & 95,141 \\
0003 & 78,854 \\
0004 & 230,000 \\
0005 & 45,561 \\
0006 & 25,000 \\
0007 & 20,000 \\
0008 & 98,374 \\
0009 & 29,403 \\
0010 & 19,701 \\
\hline
\text{Total} & 698,034 \\
\end{array}
\]

**CURRENT QUARTER EXPENDITURES**  
\[
\begin{array}{ll}
\text{Total} & 0.00 \\
\end{array}
\]

**CONTRACT CEILING**  
$4,200,000.00

**FUNDING TO DATE**  
- $698,034.00

**PENDING COMMITMENTS**  
- $426,563.00

**AVAILABLE FUNDING**  
$3,075,403.00

**FUNDING TO DATE**  
- $698,034.00

**YTD EXPENDITURES**  
- $0.00

**OUTSTANDING EXPENDITURES**  
$698,034.00

**INCREMENTS**  
\[
\begin{array}{ll}
\text{DO} & \text{Funding} \\
0001 & 90,729.00 \\
0002 & 66,680.00 \\
0003 & 54,154.00 \\
0004 & 20,000.00 \\
C-8-2400 & 95,000.00 \\
C-8-2402 & 100,000.00 \\
\hline
\text{Total Pending} & 426,563.00 \\
\end{array}
\]
CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: OCT-DEC '88

CURRENT QUARTER FUNDING $120,834.00
DO # 0004 $66,680
0006 $54,154
-------
$120,834

CURRENT QUARTER EXPENDITURES $28,740.82

<table>
<thead>
<tr>
<th>CONTRACT CEILING</th>
<th>$4,200,000.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNDING TO DATE</td>
<td>$818,868.00</td>
</tr>
<tr>
<td>* PENDING COMMITMENTS</td>
<td>$784,729.00</td>
</tr>
</tbody>
</table>

AVAILABLE FUNDING $2,596,403.00

<table>
<thead>
<tr>
<th>FUNDING TO DATE</th>
<th>$818,868.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>YTD EXPENDITURES</td>
<td>$28,740.82</td>
</tr>
</tbody>
</table>

OUTSTANDING EXPENDITURES $790,127.18

<table>
<thead>
<tr>
<th>*</th>
<th>DO #</th>
<th>INCREMENTAL FUNDING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td></td>
<td>$90,729.00</td>
</tr>
<tr>
<td>0007</td>
<td></td>
<td>$20,000.00</td>
</tr>
<tr>
<td>C-8-2400</td>
<td>STATE UNIV OF NY/FAM</td>
<td>$95,000.00</td>
</tr>
<tr>
<td>C-8-2402</td>
<td>RENSSELAER/SAULNER</td>
<td>$100,000.00</td>
</tr>
<tr>
<td>B-9-3592</td>
<td>UNIV OF CA/DAVIS/LEVITT</td>
<td>$60,000.00</td>
</tr>
<tr>
<td>N-9-5514</td>
<td>SOHAR INC./HECHT</td>
<td>$50,000.00</td>
</tr>
<tr>
<td>C-9-2015</td>
<td>NCS/O’NEAL</td>
<td>$100,000.00</td>
</tr>
<tr>
<td>A-9-1120</td>
<td>HITEC, INC./KAZAKOS</td>
<td>$75,000.00</td>
</tr>
<tr>
<td>E-9-7057</td>
<td>UNIV OF TX/ARLINGTON/FUNG</td>
<td>$40,000.00</td>
</tr>
<tr>
<td>E-9-7093</td>
<td>MONTANA STATE/JOHNSON</td>
<td>$34,000.00</td>
</tr>
<tr>
<td>S-9-7552</td>
<td>ALFRED UNIV/SYNDER</td>
<td>$20,000.00</td>
</tr>
<tr>
<td>C-9-2404</td>
<td>STANFORD UNIV/WIDROW</td>
<td>$100,000.00</td>
</tr>
</tbody>
</table>

TOTAL PENDING $784,729.00
CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JAN-MAR '89

CURRENT QUARTER FUNDING $574,457.00

<table>
<thead>
<tr>
<th>DO #</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>$90,729</td>
</tr>
<tr>
<td>0011</td>
<td>$75,000</td>
</tr>
<tr>
<td>0012</td>
<td>$75,000</td>
</tr>
<tr>
<td>0013</td>
<td>$59,989</td>
</tr>
<tr>
<td>0014</td>
<td>$49,989</td>
</tr>
<tr>
<td>0015</td>
<td>$70,000</td>
</tr>
<tr>
<td>0016</td>
<td>$43,750</td>
</tr>
<tr>
<td>0017</td>
<td>$30,000</td>
</tr>
<tr>
<td>0018</td>
<td>$22,000</td>
</tr>
<tr>
<td>0019</td>
<td>$38,000</td>
</tr>
<tr>
<td>0020</td>
<td>$20,000</td>
</tr>
</tbody>
</table>

CURRENT QUARTER EXPENDITURES $86,324.15

| CONTRACT CEILING       | $4,200,000.00 |
| FUNDING TO DATE        | $1,393,325.00 |
| * PENDING COMMITMENTS  | $594,651.00   |

AVAILABLE FUNDING $2,212,024.00

FUNDING TO DATE $1,393,325.00

YTD EXPENDITURES $115,064.97

OUTSTANDING EXPENDITURES $1,278,260.03

* DO #  incremental funding $20,000.00
   0007  INCREMENTAL FUNDING
   0011  INCREMENTAL FUNDING $19,568.00
   0012  INCREMENTAL FUNDING $24,700.00
   0015  INCREMENTAL FUNDING $29,783.00
   0016  INCREMENTAL FUNDING $31,250.00
   0017  INCREMENTAL FUNDING $10,000.00
   0018  INCREMENTAL FUNDING $12,000.00
   0019  INCREMENTAL FUNDING $12,000.00
   N-9-5732 GRIFFIN $25,000.00
   A-9-1476 BOWDOIN COLLEGE/CHONACKY $20,350.00
   E-9-7110 UNIV OF LOWELL/SALES $50,000.00
   S-9-7559 UNIV OF MICHIGAN/ROBINSON $20,000.00
   B-9-3621 SRI/LUNT $20,000.00
   N-9-5308 KAMAN SCIENCES $100,000.00
   E-9-7119 DARTMOUTH COLLEGE/CRANE $100,000.00

TOTAL PENDING $594,651.00
**CONTRACT FUNDS STATUS REPORT (DD FORM 1586)**

**CONTRACT NUMBER** F30602-88-D-0025  
**QUARTER:** APR-JUN '89

### Current Quarter Funding

<table>
<thead>
<tr>
<th>DO #</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0021</td>
<td>$25,000</td>
</tr>
<tr>
<td>0022</td>
<td>$45,000</td>
</tr>
<tr>
<td>0023</td>
<td>$20,350</td>
</tr>
<tr>
<td>0024</td>
<td>$50,000</td>
</tr>
<tr>
<td>0025</td>
<td>$20,000</td>
</tr>
</tbody>
</table>

**Total:** $160,350.00

### Current Quarter Expenditures

**Total:** $318,963.82

### Contract Ceiling

**Funding To Date:** $1,553,675.00

*Pending Commitments:* $718,994.00

**Available Funding:** $1,927,331.00

### Funding To Date

**YTD Expenditures:** $1,553,675.00

**Outstanding Expenditures:** $1,119,646.21

### Incremental Funding

<table>
<thead>
<tr>
<th>DO #</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0007</td>
<td>$20,000</td>
</tr>
<tr>
<td>0011</td>
<td>$19,568.00</td>
</tr>
<tr>
<td>0012</td>
<td>$24,700.00</td>
</tr>
<tr>
<td>0015</td>
<td>$29,783.00</td>
</tr>
<tr>
<td>0016</td>
<td>$31,250.00</td>
</tr>
<tr>
<td>0017</td>
<td>$10,000.00</td>
</tr>
<tr>
<td>0018</td>
<td>$12,000.00</td>
</tr>
<tr>
<td>0019</td>
<td>$12,000.00</td>
</tr>
<tr>
<td>0022</td>
<td>$54,693.00</td>
</tr>
<tr>
<td>B-9-3621</td>
<td>$20,000.00</td>
</tr>
<tr>
<td>N-9-5308</td>
<td>$100,000.00</td>
</tr>
<tr>
<td>E-9-7119</td>
<td>$100,000.00</td>
</tr>
<tr>
<td>N-9-5740</td>
<td>$15,000.00</td>
</tr>
<tr>
<td>N-9-5317</td>
<td>$50,000.00</td>
</tr>
<tr>
<td>S-9-7625</td>
<td>$20,000.00</td>
</tr>
<tr>
<td>N-9-5314</td>
<td>$100,000.00</td>
</tr>
<tr>
<td>N-9-5315</td>
<td>$100,000.00</td>
</tr>
</tbody>
</table>

**Total Pending:** $718,994.00
CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JUL-SEP '89

CURRENT QUARTER FUNDING

<table>
<thead>
<tr>
<th>DO #</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0017</td>
<td>$10,000</td>
</tr>
<tr>
<td>0026</td>
<td>$15,000</td>
</tr>
<tr>
<td>0027</td>
<td>$20,000</td>
</tr>
<tr>
<td>0028</td>
<td>$50,000</td>
</tr>
<tr>
<td>0029</td>
<td>$40,000</td>
</tr>
<tr>
<td>0030</td>
<td>$30,000</td>
</tr>
<tr>
<td>0031</td>
<td>$20,000</td>
</tr>
<tr>
<td>0032</td>
<td>$66,000</td>
</tr>
<tr>
<td>0033</td>
<td>$70,000</td>
</tr>
<tr>
<td>0034</td>
<td>$85,000</td>
</tr>
<tr>
<td>0035</td>
<td>$70,000</td>
</tr>
</tbody>
</table>

$476,000.00

CURRENT QUARTER EXPENDITURES

$415,422.69

CONTRACT CEILING

$4,200,000.00

FUNDING TO DATE

$2,029,675.00

* PENDING COMMITMENTS

$253,994.00

AVAILABLE FUNDING

$1,916,331.00

FUNDING TO DATE

$2,029,675.00

YTD EXPENDITURES

$849,451.48

OUTSTANDING EXPENDITURES

$1,180,223.52

* DO # 0007 INCREMENTAL FUNDING

$20,000.00

0011 INCREMENTAL FUNDING

$19,568.00

0012 INCREMENTAL FUNDING

$24,700.00

0015 INCREMENTAL FUNDING

$29,783.00

0016 INCREMENTAL FUNDING

$31,250.00

0018 INCREMENTAL FUNDING

$12,000.00

0019 INCREMENTAL FUNDING

$12,000.00

0022 INCREMENTAL FUNDING

$54,693.00

N-0-5703 UNIV OF SOUTHERN FLA/WILSON

$50,000.00

TOTAL PENDING

$253,994.00
**CONTRACT FUNDS STATUS REPORT (DD FORM 1586)**

**CONTRACT NUMBER** F30602-88-D-0025

**QUARTER:** OCT-DEC '89

### CURRENT QUARTER FUNDING

<table>
<thead>
<tr>
<th>DO #</th>
<th>Amount</th>
<th>Contract ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>$9,000</td>
<td>C-8-2129</td>
</tr>
<tr>
<td>0011</td>
<td>$19,568</td>
<td>C-8-2400</td>
</tr>
<tr>
<td>0012</td>
<td>$24,700</td>
<td>C-8-2402</td>
</tr>
<tr>
<td>0015</td>
<td>$29,783</td>
<td>C-9-2015</td>
</tr>
<tr>
<td>0016</td>
<td>$31,250</td>
<td>A-9-1120</td>
</tr>
<tr>
<td>0018</td>
<td>$12,000</td>
<td>E-9-7093</td>
</tr>
<tr>
<td>0019</td>
<td>$62,000</td>
<td>C-9-2109</td>
</tr>
<tr>
<td>0022</td>
<td>$54,693</td>
<td>C-9-2404</td>
</tr>
<tr>
<td>0028</td>
<td>$50,000</td>
<td>N-9-5308</td>
</tr>
</tbody>
</table>

**CURRENT QUARTER FUNDING TOTAL:** $292,994.00

### CURRENT QUARTER EXPENDITURES

**CURRENT QUARTER EXPENDITURES TOTAL:** $286,691.16

### CONTRACT CEILING

**FUNDING TO DATE TOTAL:** $2,322,669.00

**PENDING COMMITMENTS TOTAL:** $595,000.00

**AVAILABLE FUNDING TOTAL:** $1,282,331.00

### FUNDING TO DATE

**YTD EXPENDITURES TOTAL:** $1,136,142.64

**OUTSTANDING EXPENDITURES TOTAL:** $1,186,526.36

### TOTAL PENDING

**TOTAL PENDING:** $595,000.00

**WAITING FOR PROPOSALS:**
- P-0-6018 UAH/CAULFIELD
- P-0-6021 GT/SUMNERS
- P-0-6022 CORNELL UNIV/TANG
- B-0-3353 ROCHESTER INST/LASKY
CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JAN-MAR '90

CURRENT QUARTER FUNDING

<table>
<thead>
<tr>
<th>DO #</th>
<th>Funding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0007</td>
<td>$9,000</td>
</tr>
<tr>
<td>0029</td>
<td>$19,568</td>
</tr>
<tr>
<td>0030</td>
<td>$24,700</td>
</tr>
<tr>
<td>0036</td>
<td>$27,783</td>
</tr>
<tr>
<td>0037</td>
<td>$31,250</td>
</tr>
</tbody>
</table>

$114,301.00

CURRENT QUARTER EXPENDITURES

$376,743.62

CONTRACT CEILING
$4,200,000.00
FUNDING TO DATE
- $2,436,970.00
* PENDING COMMITMENTS
- $532,800.00

AVAILABLE FUNDING

$1,230,230.00

FUNDING TO DATE
$2,436,970.00
YTD EXPENDITURES
- $1,512,896.26

OUTSTANDING EXPENDITURES

$924,083.74

* DO# 0034 N-9-5314 INCREMENTAL FUNDING $15,000.00
   0035 N-9-5315 INCREMENTAL FUNDING $30,000.00
   0037 P-0-6011 INCREMENTAL FUNDING $10,000.00
   N-0-5703 UNIV OF SOUTHERN FLA/WILSON $50,000.00
   A-0-1402 UNIV OF CA/SMOOT, BARBER, GT $100,000.00
   C-0-2456 NEW JERSEY INST/BAR-NESS $100,000.00
   P-0-6021 GT/SUMNERS $100,000.00
   P-0-6022 CORNELL UNIV/TANG $30,800.00
   B-0-3353 ROCHESTER INST/LASKY $20,000.00
   F-0-6018 UAH/CAULFIELD $77,000.00

TOTAL PENDING $532,800.00

WAITING FOR PROPOSALS: P-0-6018 UAH/CAULFIELD
                           F-0-6021 GT/SUMNERS
                           P-0-6022 CORNELL UNIV/TANG
                           B-0-3353 ROCHESTER INST/LASKY
Final Report for the Project
entitled
"FREQUENCY DOMAIN SIGNAL PROCESSING"
Submitted to
Georgia Institute of Technology
Subcontract No: E-21-T12-S1
RPI Project No: 5-25705
Period Covered - January 20, 1989 - June 30, 1990

Rensselaer Polytechnic Institute
Troy, New York 12180-3590
Gary J. Saulnier, Pankaj K. Das and Joseph M. Pennisi
October 1, 1990
SPREAD SPECTRUM COMMUNICATIONS

EMULATOR DESIGN

Gary J. Saulnier, Pankaj K. Das,
Joseph M. Pennisi

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

October 1, 1990
# Table of Contents

1. Introduction ............................................................................................................ 1  
   1.1 Program Objective .......................................................................................... 1  
   1.2 Program Goal .................................................................................................. 1  
   1.3 Report Organization ....................................................................................... 3  

2. System Organization and Design ......................................................................... 4  
   2.1 System Organization ...................................................................................... 4  
      2.1.1 Setup Program ......................................................................................... 6  
      2.1.2 Configuration Files ................................................................................. 6  
      2.1.3 Simulation Execution and Observation .................................................... 7  
      2.1.4 Output Files .............................................................................................. 7  
   2.2 Simulator Design .............................................................................................. 8  
      2.2.1 Simulator Structure ................................................................................. 8  
      2.2.2 Simulator Subsections ............................................................................. 11  
      2.2.3 Simulator Data Transfer ........................................................................ 11  

3. The Data Packet ..................................................................................................... 13  
   3.1 Data Packet Structure .................................................................................... 14  
   3.2 Transmitter Structure ..................................................................................... 16  
   3.3 Channel Structure ........................................................................................... 18  
   3.4 Receiver Structure .......................................................................................... 20  
   3.5 Stats Structure ................................................................................................ 22  
   3.6 Address Structure ........................................................................................... 24  
   3.7 Data Structure Listing .................................................................................... 25  

4. The Function Module ............................................................................................... 32  
   4.1 Overall Module Design ................................................................................... 32  
   4.2 Setup Modules for the Function Modules ....................................................... 33  
   4.3 Specific Module Input/Output Requirements .................................................. 34  
       4.3.1 Transmitter Module Requirements ......................................................... 34  
       4.3.2 Channel Module Requirements ............................................................... 35  
       4.3.3 Receiver Module Requirements ............................................................... 35  
   4.4 Example Function and Setup Modules: AWGN Channel Effect ..................... 36  

5. Configuration and Output Files ............................................................................. 39  
   5.1 System Configuration Files ............................................................................ 39  
   5.2 Sample Configuration File: emsys1.sss .......................................................... 42  
   5.3 System Output Files ....................................................................................... 46  
   5.4 Sample Output File: emsys1.out .................................................................... 47  

6. Future Expansion .................................................................................................... 48  
   6.1 Integration of New Modules into the System .................................................. 48  
   6.2 A New Module Integration Example .............................................................. 49  
   6.3 Conclusion and Future Directions ................................................................... 50  

# List of Figures

2.1 Emulator: System Organization ........................................................................... 5  
2.2 Digital Communication System Diagram ......................................................... 8
1. Introduction

1.1 Program Objective

The objective of this program is to develop the underlying structure for an emulator which can be used to study, design and evaluate, through simulation, spread spectrum communication techniques. The emulator is written in "C" and is modular, meaning that the complete emulator would include a library of function modules which can be used to create a spread spectrum system. New spread spectrum techniques can be investigated by creating new modules from discrete-time algorithms and adding them to the library of existing functions.

The purpose of this program is to define the overall structure of the emulator and its components in order to facilitate creation of the basic simulation platform, function modules, and associated input and output files. The organization of this emulator structure centers around three major components:

1) Data Packet Structure: the data structure which links the modules together and facilitates the storing and passing of data between modules.

2) Module Structure: the requirements for a module to interface with the data packet structure.

3) Emulator Control: the control of the emulator's execution, including execution of the simulation portion, initialization of the function modules, and collection of the desired performance data.

This initial framework must be in place before an extensive function module library is created. Presently, only basic modules have been created; future efforts will produce more function modules for transmitter, channel and receiver operations.

1.2 Program Goal

The goal of the program is to develop a communication simulation tool, here titled emulator, that is specifically intended for designing and evaluating spread spectrum techniques, but is not necessarily limited to spread spectrum only. The emulator should maximize flexibility while minimizing program overhead and complexity. In addition, the
emulator must not be host dependent, i.e. it must be portable to various types of host systems which are able to compile and execute C programs.

Minimizing the program complexity and overhead involved in executing the emulator does limit the overall flexibility of the system; however, the end result is a simulation tool which can be expanded easily to include new functions and contains code that is easier to maintain.

The emulator design was performed using a Commodore Amiga 2500 as the standard platform. The simulator code is written in Lattice C, version 5.0. The C code is kept compatible with other C compilers by avoiding any extensions provided exclusively by the Lattice compiler and adhering strictly to the ANSI C standard. As a check, during development all code is ported to a SUN workstation where it is compiled and executed.

Many communication simulators include a graphical interface which allows for the creation and simulation of a desired system to proceed through the use of icons and a graphic pointing device, i.e. mouse. The structure of this emulator allows for inclusion of such an interface; however, it is also structured so the interface is not necessary for execution of the emulator. By structuring the project in this manner, the emulator can be used while the interface is still under development. This will prove important since a graphical interface usually requires a great deal of code and debugging time before it works properly. Another reason for separating the two in this manner is that graphical interfaces are usually host dependent and including these routines in the main structure severely limits the portability. Keeping them separate allows for two options in development. First, separate graphic routines may be written for each host system, while porting the non-graphic routines directly. Second, it still allows for graphic routines to be developed with a standard graphics interface, such as "X Windows".

Finally, the modular structure of the program provides the ability to interface the software modules with actual hardware components. Through the use of computer
interface boards, it will be possible in the future to test an actual hardware component by using the software modules to represent the rest of the communication system.

1.3 Report Organization
Section 2 discusses the organization and structure of the emulator system and the overall design approach, including design tradeoffs. Section 3 discusses the contents and structure of the data packet used to transport data between modules. Section 4 discusses, in detail, the structure of the function modules used to create each of the simulation systems. Section 5 explains the structure of the configuration files and the output files used to interface with the emulator. Finally, section 6 discusses the design schedule for the creation of new modules, and the creation of the setup program and graphic observation routines.
2. System Organization and Design

2.1 System Organization

This emulator system is broken into 4 major parts: the Setup program, the Configuration files, the Simulation Execution and Observation program, and the Output files; these are shown in figure 2.1. In this structure, the Setup program is a graphics-intensive, user friendly set of routines which help the operator setup a spread spectrum communication system to simulate. Once the system is fully specified, the Setup program writes a Configuration file.

The Configuration files have a specific structure, known to both the Setup program and the Simulation Execution and Observation program. They contain all the necessary information for setting up a simulation with the proper parameters.

The Simulation Execution and Observation program contains the heart of the emulator. This is the portion of the emulator which performs all the calculations for each
simulation; it also contains the routines used to observe the bit streams, digital waveforms, bit error rates, etc. The operator chooses a Configuration file with the parameters of the system to be simulated, then this program reads in all the parameters, sets up the simulation, and begins execution. Upon completion of the simulation, this program records the desired results in an operator chosen Output file.

The Output files contain the performance results of the simulation of a particular system. These, too, are arranged in a very specific order so that they can be easily used by other programs in the future.

The emulator was broken down into these four parts for several reasons. First, it is desired that the emulator be graphically oriented as well as portable to many computer systems. By its nature, much of the C program which performs the actual simulation calculations is easily portable to many systems; however, graphics tend to be very system specific. Therefore, keeping these two sections as separate as possible allows more of the program to be ported directly to another machine while only redeveloping the portion of the program which interacts with the graphics.

Second, a project of this size takes a long time to develop and debug; therefore, it is advantageous to break it down into smaller parts which can operate independently of the rest of the system. This design allows development of the Simulation Execution and Observation program to proceed without necessarily having simultaneous development of the graphics in the Setup program. Once the Simulation program is completed and operating properly, it can be used in conjunction with Configuration files which have been input as text. This allows the system to be fully operational for testing while development can continue on the Setup program.

Finally, this type of system keeps a record of previous results as well as the system parameters used to obtain them. This can be very advantageous when the emulator is used over long periods of time because records of all simulations are automatically kept on file for reference whenever they are needed.
2.1.1 Setup Program

The purpose of the Setup program is to make the creation of communication system simulations easier for the operator. The program is graphically oriented and requires keyboard input only when names or numbers are needed. Much of the setup consists of choosing between several options. For example, as part of the setup procedure it is necessary to choose a spread spectrum modulator. This is done by choosing from a list of available modulators with a mouse or other pointing device. Then some parameters - those which have only a limited number of options - such as the type of spreading system (ds, ccsk, etc.) or taps for a maximal length shift register to generate a PN code, can also be chosen with the mouse. Others, such as signal-to-noise ratio (Eb/No) or jammer frequency, must be input directly through the keyboard. This procedure continues until the operator selects all the functions and parameters for the desired emulation; the program then writes the Configuration file.

2.1.2 Configuration Files

As mentioned before, the Configuration files contain all the necessary information for setting up a simulation. This information has to be kept in a strict format so that it can be read into the Simulation program correctly. Although, the data has to be in a strict format, there is room in each file for comments regarding that particular Configuration file. Concessions have also been made for future expansion; included as a parameter in the configuration file is a version number. This version number is used by both the setup program and the Simulation program to determine the exact structure of the Configuration file. This parameter allows the structure to change in the future, and as long as the version number is changed, each program will know exactly what information structure to expect.

Finally, as a general rule, it is a good idea to have one Configuration file for each particular system to be simulated. For example, if a system is developed and it is desired
that this system be tested within a range of signal-to-noise ratios, then it is advantageous to create a different configuration file for each signal-to-noise ratio. This may create a lot of configuration files but it makes execution of the emulator much simpler. The emulator can be setup to call each Configuration file in order, treating them as separate simulations, and write separate Output files for each ratio. When this has been completed, a complete record of each simulation is stored in the files. Therefore, if there are any peculiarities, each run of the simulation has its own Configuration file and Output file independent of all the others. Also, if power is suddenly interrupted and the computer shuts down, only those simulations which have not finished will be lost, the completed simulations will have been saved, thus preventing the unnecessary loss of data or the need to repeat simulations. This feature could be quite significant when a group of simulations is expected to take several days.

2.1.3 Simulation Execution and Observation

This program performs all the necessary calculations for determining the performance of a particular spread spectrum communication system, and allows the operator to view any of the bit streams, waveforms, error rates, etc. The program reads in the operator chosen Configuration file and, upon completion of the simulation, writes the performance data to the operator chosen Output file. The end of the simulation is determined by data stored in the Configuration file. Two variables are used to determine how many bits are to be sent through the system; however, the operator also has control to stop execution at any time. Finally, this is also the program which contains all the function modules written for the emulator.

2.1.4 Output Files

The Output files contain all the required performance information from a simulation run. This includes information relating to the numbers of bits or symbols transmitted and
the number of errors detected. This file will also contain the name of the Configuration file used to generate the results.

2.2 Simulator Design

In developing a communication system emulator such as this one, it is important to make it able to handle as much variety in the simulated systems' structures as possible. The more an emulator can simulate, the more useful it will be to a system designer. The digital communication system model used for the simulator is shown in figure 2.2.

![Digital Communication System Diagram](image)

Figure 2.2 Digital Communication System Diagram

2.2.1 Simulator Structure

This model is grouped into three main sections: Transmitter, encompassing the top five blocks; Channel, encompassing only the digital waveform channel; and Receiver, encompassing the bottom four blocks. Inside the simulator, the main program operates by calling each of these three sections in the proper order: Transmitter, Channel, Receiver.
This simulates source bits passing from the simulated transmitter through the simulated channel and into the simulated receiver. The only operation the main program is concerned with is determining how many total bits to send through the system. This comes from the Configuration file used to setup the system. This approach, keeping the controlling program small, was used in order to keep the emulator as flexible as possible.

Each of the above three sections also has a specific task. Each one is responsible for calling all the subsections below it. For example, the transmitter consists of five subsections or types of devices. The transmitter always calls an information source routine, a data encryption routine, a channel encoder routine, a spread spectrum modulation routine and a baseband modulation routine. These routines are chosen when the system is setup and are read in from the Configuration file. It is possible that one or more of these routines does not change the data at all, i.e. a particular system may not use data encryption, however, a data encryption routine must be included which acts as though it is encrypting the data in order for the next device to work properly. This method of organization was chosen because the transmitter needed to be able to work with a variety of different structures, yet they all had to be able to be simulated by the same program. By forcing each simulated transmitter to call each of the routines listed above, all possible system structures could be simulated by the same program. The only requirement this leaves for the programmer is to include modules which can simulate any transmitter using exactly these five blocks. Similarly, the channel and receiver are also organized on this principle.

Each of the systems above may require different numbers of bits per symbol to operate. For example, a Hamming (7,4) channel encoder will require 4 input bits before it can encode and output its 7 bits. The same may be true of the encryptor routine, the spread spectrum modulator routine, and the baseband data modulator routine. The controlling program would have to be much more complex if it had to determine whether there were enough bits for a device to operate properly. For example, if the devices in the transmitter
needed 4 bits, 4 bits, 3 bits and 2 bits respectively, the controlling program would have to
generate 4 bits in the source before the first device could work. The second device could
operate on those 4 bits out of the encryptor, however, if the second device was a
Hamming(7,4) encoder, then it would produce 7 bits for every 4 bits in. Since the third
device requires 3 bits, the controlling program would have to call this routine twice, using
6 of the 7 available bits, then it would have to remember that there was one bit left over
which would have to go through first next time it came to this device. This approach creates
excessive overhead in the main controlling routine, which we want to avoid.

One option to eliminate this problem is to initially send enough bits so that every
device has an integer number of full symbols to operate on. Referring to the above
example, with a 4-bit data encryptor, a (7,4) encoder, a 32-chip 8-ary ccsk spreader, and a
QPSK baseband modulator, the source module would generate 12 bits for each run. This
allows each device to have an integer number of symbols to operate on - 3 symbols, 3
symbols, 21 symbols (from the Hamming(7,4) encoder) and 112 symbols, respectively.
This solves the problem of complexity because there will never be any bits left over to send
through the system on the next loop through. This grouping of bits (12 bits above) is
called a packet - after the data packet in which they are stored and transferred.

Although this solves the problem of program complexity, it may cause the source
module to create packets with a large number of bits. This may slow the observations in
the emulator down measurably. The whole simulation may require the same amount of
time, though updates that can be observed will come much more slowly. Despite this
drawback, it was determined that the reduced complexity of the main program would have
a greater impact on the system than the frequency of the observation updates.

If the frequency of observation updates become critical, it is possible to write
observation routines which will split the observation of a large group of symbols so that
they may be observed as groups of smaller ones.. For instance, above, the source
generated 12 bits per symbol, but this produced 112 symbols for the modulator to transmit.
Since all the data is available at the end of transmission and reception of the simulated data, an observation routine can be written which splits this block of 112 symbols into smaller blocks. Because of the structure of the program, this decision is completely arbitrary and was designed in as a choice for future expansion.

2.2.2 Simulator Subsections

The subsections of the simulator are where the real simulation work is performed. These are the portions which are varied for different simulations. Each of these subsections is referred to as a 'module'. It is these modules that give the simulator its flexibility. Again, for simplicity, a module has very simple requirements to fit into the structure of the program; all modules are defined in terms of their input and output. In other words, as long as a module accepts its input from the proper spot and places its output in the proper spot, then it will work within the framework of the simulator. Now this is no guarantee that it produces the correct result, only that it operates properly within the simulator's framework. The algorithms used for each module will have to be tested to insure that they produce the desired result. A detailed description of the modules is forthcoming in section 4.

2.2.3 Simulator Data Transfer

All data in the simulator is stored in the structure known as the data packet. Each routine in the simulator - all the subsections, or modules, that is - has very specific guidelines for interacting with the data packet. As will be explained in detail in section 3, the data packet has a great deal of structure to it; each type of module (source, encryptor, modulator, etc.) has specific areas that it is required to interact with. For example, a channel encoder module is required to take the data stored in the data encryptor module, encode it according to its algorithm, and place it in an array reserved specifically for encoded data.
Each of these routines perform their chosen tasks in the proper order, from information source to information sink, adjusting the data packet as required. In this way data is transferred from one routine to another correctly. Also, as these modules perform their tasks on one complete pass through the simulator, none of the original data is overwritten or lost. This design consideration was made so that at the end of one simulation run all the data generated by the system will be available for observation. For the next run, these locations are overwritten by the data for the new pass while the statistical information is updated.
3. The Data Packet

Organization of the data involved in this software project presented a major task in its development. Since the nature of the program is one of testing and observation, rather than streamlined efficiency, and since a great deal of emphasis is placed on future expansion, a major effort was made to make the program as expandable as possible. This expandability means the data has to be organized in a fashion similar to the emulator, providing as much modularity as possible, while still adhering to a strict format so that program complexity could be minimized. After several preliminary attempts and a good deal of discussion, it was decided that the best software design technique to use for this project would be Abstract Data Structures.

An abstract data structure is a way of abstracting all the necessary information for a project and representing it by structures in C. This way, necessary simulation data can relate directly to the physical data it is meant to represent in such a way as to make it obvious in the program code. For example, the data packet consists of sections representing the data used in the transmitter, the data in the channel, and the data in the receiver. They are titled appropriately in the structure, thus making it obvious what each is meant to represent. Also, since C allows the nesting of structures, each of the sections below the transmitter, channel and receiver are also abstracted. This can continue down until the basic data types found in C are encountered, i.e. integers, etc. Not only does this allow for a much better understanding of the data involved, it breaks up the data, encapsulating relevant portions and separating non-relevant ones.

The other advantage to abstract data structures is that, since they completely define all the data necessary for a project, all the program routines are simply written to manipulate that data. Thus for this project, an abstract data structure was defined and called the data packet; all the modules in the emulator simply operate on the data included in the data packet.
Finally, since this is a tool of experimentation and observation, it is desirable to extract as much information as possible from the simulations. Therefore, as the simulation proceeds along, no intermediate results are lost. The program stores all intermediate results within their proper location in the data packet. This way, once one group of bits has been simulated traversing the system from source to sink, all the intermediate information is available and can be examined in detail at the information sink. The information sink is where error statistics are updated and all graphic output of the bit streams, waveforms, transforms, etc. occurs.

3.1 Data Packet Structure

The structure of the data packet follows closely to the breakdown of the emulator system. Several levels of encapsulation exist to make both the program and the data packet itself very modular. On the uppermost level of the structure are the basic components of a digital communication system: the transmitter, the channel, and the receiver. Also included at this level is storage space for performance statistics and storage space for implementing the relatively complex, but necessary, method of program storage space allocation. The last section is used solely by the program and is not directly related to the structure of the simulations.

Below the transmitter, channel and receiver levels are sublevels. These sublevels, in both the transmitter and the receiver, contain structures representing each of the devices used by either the transmitter or receiver. The transmitter section contains structures abstractly representing each of its components: information source (source), data encryptor (encrypt), channel encoder (encode), spread spectrum modulator (spread), and baseband modulator (modulate). There is also an array which contains the devices, or modules, to be used by the transmitter in its simulation (transmit[]). The names in parentheses represent the actual names used in the data structure as can be seen in the actual data structure listing provided at the end of this section.
The receiver also has similar abstract devices within it: data/spread spectrum demodulator (demodulate), channel decoder (decode), data decryptor (decrypt), and information sink (sink). The last field within the receiver structure is an array which contains the devices that the receiver will use during its simulation (receive[]).

The channel is structured somewhat differently because there is not a strict physical analogy present as there was in the transmitter and receiver. Therefore, in the channel, these sublevels represent different types of phenomenon encountered in a communications channel. As a basic platform, there are three types of phenomenon represented: additive white gaussian noise (awgn_noise), single tone jammers (tone_jam), and sweeping tone jammers (sweep_jam). Obviously these are not the only types encountered, and as the program grows, more types of phenomenon, or effects, will need to be added. However, since this structure is so modular they are easily added to the system without disturbing any of the structure already in place. The channel also has an array which contains the list of channel effects to use during the simulation (channel_effects[]). The number of elements in this list is not constant like in the transmitter and receiver because the number of channel effects can vary, while the number of devices in the transmitter and receiver, for the first version of the emulator, remains constant.

Also, it is worthy to note that once a system has been setup with a specific data packet structure it will continue to work perfectly after the data packet has been augmented. For example, a system is setup for simulation, executed and stored with one data packet structure; then the system is augmented to include more channel effects. The simulation will still execute fine as long as none of the original channel effects have been changed. This upward compatibility property was considered very important to the design so that any work done on earlier versions of the emulator would still work on the revisions.

Finally, the method of data storage in the data packet presented some potential problems. Much of the data storage required consists of data structured into arrays, or lists. C has facilities for storing data in just such a manner, however, in order for it to
work properly, the largest size needed has to be allocated at the beginning of execution. So, for every array needed in the emulator, the largest number of elements would have to be determined and then that amount of space could be allocated for the array. This meant that for each simulation run the maximum amount of memory would be allocated for each array, possibly wasting memory that could be used by the operating system for other storage. Therefore, it was necessary to determine some way to sanction off only as much memory as was necessary.

Fortunately, C also has a way of dynamically allocating memory - determining and allocating memory while the program is running. This created a little more overhead in the set up routines of the simulator because each array had to be allocated individually during execution, however, the program now only used exactly as much memory as was necessary for execution of a particular simulation. These new structures in C, known as pointers, could have a certain amount of memory allocated to them, then they could be accessed as though they were arrays. Therefore, once the initial allocation was performed, these pointers were treated and accessed as though they were declared as arrays without any problems.

At the end of a simulation run, all the memory associated to these pointers is deallocated, or freed, so that a different simulation run could use its necessary amount of the same memory.

3.2 Transmitter Structure

The transmitter section of the data packet includes all the storage space necessary for simulating a transmitter. This includes all the necessary storage for the basic platform; however, it is possible to augment this section to include more storage for new transmitter device modules without causing any problems for the existing routines.

The source subsection contains all the necessary information and storage locations for any information source module. Within this subsection are: the number of bits.
generated per symbol (bits_per_sym), and an array to store the generated bit stream
(bits[]). Note: in the program code the array is listed as a C pointer (*bits), however, in C
pointers and arrays are interchangeable; therefore, once memory is allocated to the pointer,
*bits can also be accessed as bits[].

The encrypt subsection contains all the necessary information and storage locations
for any data encryption module to encrypt and store the data from the source module.
Within this subsection are: the minimum number of bits needed for a particular encryption
algorithm (bits_per_encr_sym), and an array to store the encrypted bit stream (bits[]).

The encode subsection contains all the necessary information and storage locations
for any channel encoder module to encode and store the data from the encrypt module.
Within it are: the number of bits per symbol expected out of the encoder for each symbol
input (bits_per_encd_sym), and an array to store the encoded bit stream (bits[]).

The spread subsection contains all the necessary information and storage locations
for any spread spectrum modulator module to spread and store the data from the encoder
module. Within it are: the length of the PN sequence (pn_len), the number of bits required
for each spread symbol (bits_per_sprd_sym), an array containing the actual PN code used
to do the spreading (code[]), an array containing the bit sequence - actually the chip
sequence - of the spread symbols (chips[]), and an enumerated variable containing the type
of spreading system used - {no,ds,ccsk} - (system).

The modulate subsection contains all the necessary information and storage
locations for any baseband modulator module to digitally baseband modulate the data from
the spread spectrum modulator module. Within it are: the number of chips required for
each modulated symbol (chips_per_sym), the number of samples to generate for each
modulated symbol (samp_per_sym), an array containing the double precision samples for
the in-phase channel (data_r[]), an array containing the double precision samples for the
quadrature channel (data_i[]), an array containing the double precision values to be used to
shape the pulse before leaving the transmitter (pulse_shape[]), the normalized energy of the
pulse (norm_energy), and an enumerated variable representing the type of signal in use -
{real,complex} - (signals). Note that although double precision variables are used for all
non-integer functions, they can be converted manually to use less precision in order to test
effects like finite word lengths in system computation.

The final subsection in the transmitter is the array containing the list of devices to
execute during simulation of this particular transmitter. These devices are listed in the order
they are supposed to be executed; null devices, when nothing is supposed to be executed,
are represented by zeros.

As was mentioned before, these represent all the storage needed for implementing
the basic simulation platform. As the emulator's library of functions expands, it may be
necessary to add to this list of locations. Once again, as long as this basic structure stays
intact, any additions to the structure will not affect the existing modules. The procedure for
adding locations to this structure will be explained later in section 6.

3.3 Channel Structure

The structure of the channel varies a little from that of the transmitter or receiver,
because there are not a set number of distinct devices at work in the channel. The number
and type of channel effects can vary greatly from simulation to simulation. However, since
intermediate results are meaningless in the channel - the receiver has access to the final
result of the received signal only - it is not necessary to store these intermediate results.

Presently, the basic structure has three types of channel effects: additive white
gaussian noise, single tone jammers and sweeping tone jammers. Also, in each of the
jammer effects is allowance for multiple jammers of that type, so there could be more than
one single tone or sweeping tone jammers. This list could be easily augmented to include
more channel effects as discrete model algorithms are developed and modules written.

The additive white gaussian noise subsection of the channel contains all the
information necessary for the AWGN module to subject the signal to any level of AWGN.
Within it are: the desired signal to noise ratio - $E_b/N_0$ - (ebno), and the calculated variance for the Gaussian random sequence used to generate the samples (noise_var).

The tone_jam substructure contains all the parameters necessary for operating a multiple number of single tone jammers. By definition in the data packet specification, there can be up to 'MAX_JAMMER_NUMBER' single tone jammers; if more are needed then this number is simply increased to include more jammers. Specification of each jammer includes: a status variable indicating whether the jammer is active or inactive (status), the phase to start the jammer at - this is used to make the phase of the jammer continuous between sample runs of bits through the emulator - (phase), the frequency of the jammer normalized to the chip rate of the transmitter (frequency), the strength of the jammers signal - J/S - in both absolute magnitude (strength) and in decibels (strength_dB), and the cosine and sine of the phase increment - i.e. frequency - used in calculating the jammer's sample values (phase_inc_cos, phase_inc_sin). These last two variables are used because they make calculation of the new jammer sample values proceed much quicker. They take advantage of the trigonometric identities:

$$\cos(A+B) = \cos(A)\cos(B) - \sin(A)\sin(B)$$
$$\sin(A+B) = \sin(A)\cos(B) + \cos(A)\sin(B)$$

The cosines and sines of the sums are then added to the in-phase and quadrature channels of the real data signals, respectively. Since there are only multiplications and additions, this technique saves on the excessive overhead involved with calculating cosines and sines for every sample.

The sweep_jam substructure contains all the information for setting up and simulating a tone jammer that continually sweeps through a range of frequencies. With in it are: the status of the jammer - active or inactive - (status), the phase to start the sweeping jammer at (phase), the starting frequency of the sweeping jammer normalized to the chip rate of the transmitter (phase_increment), the minimum frequency of the sweeping jammer normalized to the chip rate (minimum_frequency), the maximum frequency of the sweeping
jammer normalized to the chip rate (maximum_frequency), the rate at which the frequency changes normalized to the chip rate (sweep_rate), this same rate in terms of data samples (sweep_rate_per_sample), and the strength of the jammer - J/S - in both absolute magnitude (strength) and decibels (strength_dB).

These are just some basic channel effects, as more are developed they can be added to make a more effective simulation model. And, once again, there are no intermediate results needed by the program because they would serve no useful purpose - there is no way to separate them in the physical channel; therefore, the channel simply applies all its effects and places the received signal samples at the receiver.

Finally, the channel structure has within it, an array containing the list of channel effects to be executed during simulation of this channel (channel_effects[]). As was mentioned before, this list can vary in length, depending on the channel being simulated.

3.4 Receiver Structure

This section of the data packet contains all the storage space necessary for simulating a receiver. This includes the four sections representing the devices of the receiver, a section containing parameters for the frequency domain excisers which could be used in the data/spread spectrum demodulator, and the array containing the list of devices to use in simulating the receiver.

The data/spread spectrum demodulator subsection is by far the most complex subsection in the entire system because of the amount of processing that takes place here. As new methods of information retrieval and interference suppression are developed, most of their processing will be incorporated here. Due to this complexity, a great deal of storage space has been dedicated to this device to facilitate indepth observations of the interactions within it. These components of the demodulator are: two arrays to store in-phase and quadrature components of the incoming data samples (data_i[], data_r[]), two arrays to store the real and imaginary components of the FFT of the incoming data samples.
(fft_r[], fft_i[]), two arrays to store the real and imaginary components of the FFT of the
time reversed PN code - used for frequency domain correlations - (pn_fft_r[], pn_fft_i[]),
two arrays to store the result of the multiplication of two frequency domain sequences -
used in frequency domain correlations - (mult_r[], mult_i[]), two arrays to store the output
of either the time domain or the frequency domain correlator (corr_r[], corr_i[]), two arrays
to store the result of modulating the PN code by the modulator chosen for the simulation
(code_r[], code_i[]), the length of the PN code (pn_len), the number of bit per spread
symbol (bits_per_sym), the number of chips per modulated symbol (chips_per_sym), the
number of data samples per modulated symbol (samples_per_sym), the minimum number
of bins needed for each FFT (data_len), and the total number of data samples in each of the
arrays above (samples_total).

Next the decode subsection contains all the information and storage space necessary
for implementing a channel decoder. Note that the data the channel decoder will be
operating on will not be in the form of bits, which is what would be expected since the
channel encoder produced information in the form of bits. The channel decoder operates
on the output of the correlator in the data/spread spectrum demodulator so that either hard
or soft decision decoding can be used in the decoder. Contained within this subsection are:
an array to store the bit stream once the device has decided what stream was sent - i.e. the
stream a detector would have output if it had been between the demodulator and the decoder
- (bits[]), the number of bits per encoded symbol (bits_per_encd_sym), and the total
number of bits that present in the array above.

The decrypt subsection contains all the storage space and information for decrypting
the decoded stream of bits. Included under it are: the array containing the received
encrypted bit stream (bits[]), the number of bits per encrypted symbol
(bits_per_encr_sym), and the total number of encrypted bits in the above array.

The sink substructure contains all the storage space for storing the received
information bits - the bits the receiver has decided were sent. Under this subsection are:
the array containing the stream of bits that the receiver decided had been sent (bits[]), the number of bits per transmitted symbol (bits_per_sym), and the total number of information bits that were sent by the transmitter on this simulation run.

Also included under the receiver structure is all the information regarding the frequency domain excisers. The excise substructure contains all this information so that it can be passed easily to the exciser routines. Presently, the only exciser included is the zeroing exciser - the type which notches to zero any portion of a spectrum above a certain threshold. Included under this are the parameters necessary for specifying such an exciser: the threshold (threshold), and the notchwidth in frequency bins (notchwidth). It is very likely that more types of excisers will be added to this list; when they are they will affect the data/spread spectrum demodulator only - no other device within the receiver has any access to these parameters. In the structure of this emulator, all excision is part of the demodulation process; therefore, it is carried out within the demodulator routine. This means the emulator control does not have to access the excisers at all, all access is performed from within the demodulator devices.

Finally, there is an array which contains the receiver devices chosen for this particular simulation (receive[]). It will always have only four non-zero elements, representing the four devices chosen and is setup in the exact same manner as the transmitter and channel arrays.

3.5 Stats Structure

This section is used to store all the simulation results which are later used to calculate the system performance statistics. Along with these results, there are two variables stored here which are used to determine how to control the main loop of the simulator. They contain the number of bits or errors to wait for before terminating the simulation; this will be described in detail later.
First, there are presently six variables used to store simulation results; this is just for the basic platform, so, like the rest of the system, additions can be made easily without disturbing the rest of the emulator. The six statistics stored are: the number of information bits sent through the system (bit\_count), the number of errors detected in the information bits (bit\_errors), the number of symbols sent from the information source - in case the information naturally occurs as a multi-bit symbol - (symbol\_count), the number of errors in transmitting the symbols (symbol\_errors), the number of bits sent through the channel - after encrypting and encoding - (raw\_bit\_count), and the number of errors detected in these channel bits (raw\_bit\_errors). These last two variables measure the effects of the channel on the transmitted bit stream so that later, a measure can be made of the encoder's error prevention and correction ability by comparing the error rates before and after the encoder/decoder combination.

The final two variables are used to determine how many bits get sent through the simulator for a particular system. Since the bit is the basic unit transmitted in the emulator, all measures will be taken in terms of bits. Therefore, in determining the number of times to perform the main simulation loop, only the number of bits and/or the number of bit errors are used. The two control variables are: the number of bits required to send before termination of the loop (num\_bit), and the number of bit errors required before termination of the loop (num\_err).

The loop can be controlled four different ways with these two variables. Simulation can proceed until a certain number of bits have been transmitted. It can proceed until a certain number of errors have been detected. Sometimes it is desirable for a certain number of errors to be detected but the error rate is so small that it would take too long the simulate to this number of errors, for these cases the simulator can be set up so that it terminates upon reaching either a certain number of bit errors or a certain number of bits transmitted, whichever comes first. Finally, the simulator can be set up so that it is never stopped by the program; the only way to stop it is through operator intervention. This
choice is useful for those occasions when an operator wants to work interactively with the system and not be terminated by the program suddenly.

The way the program determines when to stop the loop depends on the values stored in each of the control variables. The only values for terminating the program that make sense are positive, therefore any negative values can be used as flags without confusion. In this structure, if the variable contains a positive number, then the loop will execute until either the bit count or bit error count reaches that particular positive number. If the variable has a negative number then that variable has no control over the loop and the program will essentially ignore it. Therefore, for example, if an operator wanted to execute a simulation until he had 100,000 bits transmitted, regardless of the number of bit errors, then he would set up the two variables like this:

\[
\text{num\_bit} = 100000 \\
\text{num\_err} = -1
\]

This configuration would cause the loop to operate until at least 100,000 bit have been transmitted.

### 3.6 Address Structure

The address section is used strictly by the program to keep a record of allocated memory locations. No portion of this section relates to the actual simulation systems and the operator need never be concerned with anything from here. As was mentioned before, the program uses a rather complex memory storage structure. Due to the large amount of memory which may be required for simulation, a method of allocating just enough memory for the program to operate had to be developed; this way the program could be sure that enough memory was available before starting the simulation.

Memory allocation is performed dynamically, meaning that memory is allocated after the program has started running. First, the program reads in a Configuration file and initializes a data packet; then, from the information contained within the data packet,
determines how much memory is needed for each array. Once the size of each array is determined, the program allocates enough memory to each pointer/array used by the program through the C function `calloc()`. At the end of a simulation the memory is freed and returned to the operating system through the C function `free()`. This is the purpose of the address section: the locations that have had memory allocated to them need to have it deallocated before running another simulation, so the locations are saved in an array to facilitate easy memory deallocation of all pointers/arrays.

There are two types of arrays used by the program, integer arrays and double precision arrays, therefore there is one array to store the addresses of each type. In the address section there are: an array of pointers to store the addresses of the integer arrays (intlocs[]), an array of pointers to store the addresses of the double precision arrays (doublelocs[]), a counter for the number of integer arrays that have been allocated (addriptr), and a counter for the number of double precision arrays that have been allocated (addrdptr).

The two counters store the number of arrays locations in each array. For example, if addiptr contained a nine, then there were nine integer pointer/arrays allocated for the program so far. These numbers are used to determine how many arrays have been allocated so that all of them can be deallocated correctly at the end of a simulation. This deallocation is important because all the memory has to be available for the next simulation to be able to allocate enough memory for its execution.

### 3.7 Data Structure Listing

/* This file contains the packet structure for the spread spectrum emulator. The packet is the main vehicle for passing information throughout the program. It is broken up into three main sections: transmitter, channel, receiver. Each of these main (level 1) sections is further broken into sub-sections, (i.e. transmitter has these sub-sections: source, encrypt, encode, spread, modulator.) Furthermore, each sub-section has sub-sub-sections (level 3) which contain the basic components used by each module, (i.e. transmitter.source has these sub-sub-sections: *bits,
bits_per_sym.)

Each of the sub-sections is defined as a separate type of variable in order to keep the system very modular. The sections, sub-sections, and sub-sub-sections are defined below. */

/***************************************************************************/
#include <math.h>

typedef unsigned long CPTR; /* CPTR - used to store pointers to different variables types in an array */

typedef short BOOL; /* boolean variables */

#define TRUE 1
#define FALSE 0

#ifndef NULL
#define NULL 0L
#endif

#ifndef PI
#define PI 3.14159265358979323846
#endif

#define MAX_JAMMER_NUMBER 10 /* This constant represents the maximum number of each type of jammer (single tone, and sweeping tone) allowed. If more are needed, this number is just incremented. */

#define NUMBER_ALLOCATED_ADDRESSES 50

#define MAX_NUMBER_FUNCTIONS 10 /* This constant represents the maximum number of elements in both the transmitter and receiver arrays. */

enum signal_type /* types of signals used, real or complex */
{
    real,
    complex
};

enum system_type /* types of spreading systems available */
{
    no,
    ds,
    ccsk
};

enum x_mit_functions /* The present transmitter functions. */


```c
enum channel_functions /* The present channel functions. */
{
    no_channel_fn,
    awgn,
    single_tone_jammer,
    sweeping_tone_jammer
};

enum rcvr_functions /* The routines used in the receiver. */
{
    no_rcvr_fn,
    fd_demod_noex,
    no_decode,
    no_decrypt
};

/**
   @struct src /* information source */
   {
      int *bits; /* list of (bits_total) bits */
      int bits_per_sym; /* (k) no. of bits per symbol */
      int bits_total; /* total no. of bits in array */
   }

   @struct encr /* data encryptor */
   {
      int *bits; /* list of (bits_total) bits */
      int bits_per_encr_sym; /* (k') no. of bits per symbol */
      int bits_total; /* total no. of bits in array */
   }

   @struct encd /* channel encoder */
   {
      int *bits; /* list of (bits_total) bits */
      int bits_per_encd_sym; /* (n) no. of bits per coded symbol */
      int bits_total; /* total no. of bits in array */
   }
*/
```
struct sprd /* spread spectrum spreader */
{
    int *chips;  /* list of (chips_total) chips */
    int pn_len;  /* length of the PN sequence */
    int *code;   /* PN code */
    int bits_per_spread_sym; /* (l) no. of bits per spread symbol */
    enum system_type system; /* type of spreading system */
    int chips_total; /* total no. of chips in array */
};

struct mod /* data modulator */
{
    double *data_r;  /* real data channel, (n*p*s) samples */
    double *data_i;  /* imaginary data channel, (n*p*s) samples */
    int chips_per_sym; /* (r) no. of chips per modulated symbol */
    int samp_per_sym; /* (s) no. of samples per modulated symbol */
    double *pulse_shape; /* contains (s) samples for each pulse */
    double norm_energy; /* contains the normalized energy/pulse */
    /* and equals: SUM((pulse_shape(0..(s-1)])^2) / s; for square */
    /* pulse equals 1 */
    enum signal_type signals; /* real or complex signals */
    int samples_total; /* total no. of samples in arrays */
};

struct trans /* transmitter structure */
{
    struct src source;
    struct encr encrypt;
    struct encd encode;
    struct sprd spread;
    struct mod modulate;
    enum xmit_functions transmit[MAX_NUMBER_FUNCTIONS];
};

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

struct awgn /* additive white gaussian noise source */
{
    double ebno;  /* Eb/No ratio required */
    double noise_var; /* variance of the Gaussian noise */
};

struct tjam /* single tone jammers */
{
    double phase;  /* phase to start tone jammer at */
    double frequency; /* freq. of jammer, normalized to chip rate */
    double strength; /* magnitude of the jammer strength (not dB) */
    double strength_dB; /* strength of the jammer signal (dB) */
}
BOOL status; /* flag for active/inactive status */
double phase_inc_cos; /* phase increment for calculating */
double phase_inc_sin; /* new jammer sample value */

struct sjam /* sweeping tone jammers */
{
    double phase; /* phase to start tone jammer at */
    double phase_increment; /* random starting (freq.) for jammer */
    double minimum_frequency; /* min. normalized frequency for jammer */
    double maximum_frequency; /* max. normalized frequency for jammer */
    double sweep_rate; /* normalized rate at which freq. changes */
    double sweep_rate_per_sample; /* above rate divided by samples/chip */
    double strength; /* mag. of strength of jammer */
    double strength_dB; /* strength of the jammer signal (dB) */
    BOOL status; /* flag for active/inactive status */
};

struct chan /* channel simulator structure */
{
    struct awgns awgn_noise;
    struct tjam tone_jam[MAX_JAMMER_NUMBER];
    struct sjam sweep_jam[MAX_JAMMER_NUMBER];
    enum channel_functions channel_effects[MAX_NUMBER_FUNCTIONS];
};

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

struct zero_exc /* notching exciser - replaces bins above threshold with zero */
{
    double threshold;
    int notchwidth;
};

struct excsr_Parms
{
    struct zero_exc zero;
};

/*****************************/
struct snk /* information sink */
{
    int *bits; /* list of decoded (bits_total) bits */
    int bits_per_sym; /* (k) no. of bits per symbol */
    int bits_total; /* total no. of bits in array */
};

struct decr /* data decryptor */
{
    int *bits; /* list of decoded (bits_total) bits */
    int bits_per_encr_sym; /* (k') no. of bits per symbol */
    int bits_total; /* total no. of bits in array */
};

struct decd /* channel decoder */
{
    int *bits; /* list of decoded (bits_total) bits */
    int bits_per_encd_sym; /* (n) no. of bits per symbol */
    int bits_total; /* total no. of bits in array */
};

struct demod /* data/spread spectrum demodulator (incl. signal proc.) */
{
    double *data_r; /* incoming data samples */
    double *data_i;
    double *fft_r; /* FFT of incoming samples */
    double *fft_i;
    double *pn_fft_r; /* FFT (time reversed) of PN sequence */
    double *pn_fft_i;
    double *multi_r; /* result of multiplication of FFT sequences */
    double *multi_i;
    double *corr_r; /* Correlator output */
    double *corr_i;
    double *code_r; /* modulated PN code */
    double *code_i;
    int pn_len; /* length of PN code */
    int bits_per_sym; /* (1) no. of bits per spread symbol */
    int chips_per_sym; /* (r) no. of chips per modulated symbol */
    int samp_per_sym; /* (s) no. of data samples per modulated symbol */
    int data_len;
    /* N = 2^m: m = smallest int s.t. m >= log2((n+1)(p+1)s) */
    int samples_total; /* total no. of samples in arrays */
};

struct recv /* receiver structure */
{
    struct snk sink;
    struct decr decrypt;

30
struct decd decode;
struct demod demodulate;
struct excsr_parms excise; /* exciser parameters */
enum rcvr_functions receive[MAX_NUMBER_FUNCTIONS];
);  

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

struct error_stats /* main error statistics for the emulator */  
{
    unsigned long bit_count; /* number of bits sent so far */
    unsigned long bit_errors; /* number of bit errors detected */
    unsigned long symbol_count; /* number of symbols sent so far */
    unsigned long symbol_errors; /* number of symbol errors detected */
    unsigned long raw_bit_count; /* no. of channel bits sent so far */
    unsigned long raw_bit_errors; /* no. of channel bit errors detected */
    long num_bit; /* number of bits required to send before termination */
    long num_err; /* number of errors needed before termination */
};

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

struct addr /* addresses of all the allocated memory locations */  
{
    int *intlocs[NUMBER_ALLOCATED_ADDRESSES];
    double *doublelocs[NUMBER_ALLOCATED_ADDRESSES];
    int addriptr;
    int addrdfptr;
};

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

struct packet_structure  
{
    struct trans transmitter;
    struct chan channel;
    struct recv receiver;
    struct error_stats stats;
    structaddr address;
};
4. The Function Module

4.1 Overall Module Design

From an overall system viewpoint, the structure of each of the modules is very simple; each module is defined in terms of its input and its output. In order for a module to correctly fit within the structure of the program it is required to retrieve its input from a certain location in the data packet and place its output in another data packet location. If a module correctly performs this task then it will fit within the structure of the emulator.

Earlier it was explained that there are some definite functions associated with each of the parts of the simulator system, 5 functions in the transmitter, 1 within the channel section and 4 in the receiver section. Each of these functions has specific modules associated with it that are responsible for specific input/output functions. For example, the channel encoder routine within the transmitter is responsible for taking the data from the data encryptor subsection of the data packet, encoding it if necessary, and storing it within the channel encoder subsection of the data packet. Every channel encoder must perform this task in order to work properly within the confines of the emulator.

There are several major advantages to constructing the function modules in this way. First, since specification concerns only the input and the output, the program is not affected by what happens inside each module, thus keeping the program very modular. Routines which perform the same type of function can be exchanged easily even though they may manipulate the data in vastly different ways.

Next, all the modules are self contained. Each module has access to all the data it needs coming through the emulator; however, some modules may need to remember some of the past data (i.e. for non-memoryless modules). In order to facilitate this type of function, each module can have its own separate memory locations within it. If arrays are needed, they can be allocated normally (statically) or if necessary, dynamically, like the rest of the arrays in the data packet. All this initialization is performed in the setup routine for each module.
Finally, by placing only the input/output restrictions on the modules, this structure, if necessary, can accommodate multiple functions within a certain block. Every effort was made to make the emulator as complete as possible, however, if in the future a need arises that requires a new type of function be researched, then a module can be created which incorporates this module within another existing type of function module. Once again, this is not foreseen in the future, yet the capability exists if it is ever required.

4.2 Setup Modules for the Function Modules

Many of the function modules in the emulator require certain parameters or arrays to be initialized upon startup. Therefore, a method of initializing all these pre-operation parameters was developed.

This method includes execution of a separate setup routine for each function module in the simulation. A setup routine exists for each function module in the emulator. If the system being simulated requires a particular device, then before the emulator starts execution of the simulation calculations, it calls the setup routines of each of the devices or channel effects in this simulation. These setup routines perform tasks such as initializing device parameters, allocating memory for a function module (in addition to that in the data packet), or initializing an array of information. Essentially any task which must be performed before the function module can operate properly is placed in its setup module.

In order to keep the program modular, every function module is required to have a setup module, even if it has nothing to initialize. This increase in overhead is insignificant when compared to the decrease in complexity of the program. If a module has nothing to initialize its setup module is simply an empty function with the proper name. All setup modules are named after the function modules they initialize; they simply have the same name preceded by the characters 'SU_', to signify setup module. During the emulator's execution, they are called in the same order as their function module counterparts, and they are called just once, before execution of the main simulation loop.
As new modules are developed, they must have a setup module created concurrently. This process, as well as the integration of new modules into the basic platform, will be described, along with an example, in section 6.

4.3 Specific Module Input/Output Requirements

Below is a list of all the input and output requirements of the function modules for the transmitter, the channel and the receiver. There are further descriptions of each in the program comment listings located at the end of this report.

4.3.1 Transmitter Module Requirements

The input/output requirements of the transmitter modules are as follows. Note that terms in italics signify actual variables within the data packet.

1) Information Source - these routines are required to place

   \texttt{transmitter.source.bits\_total} bits into the array \texttt{transmitter.source.bits[]}.

2) Data Encryptor - these routines are required to take the

   \texttt{transmitter.source.bits\_total} bits present in the array

   \texttt{transmitter.source.bits[]} , encrypt them by an encryption method (or leave them unencrypted, if necessary) resulting in \texttt{transmitter.encrypt.bits\_total} bits, and place them into the array \texttt{transmitter.encrypt.bits[]}.

3) Channel Encoder - these routines are required to take the

   \texttt{transmitter.encrypt.bits\_total} bits in the array \texttt{transmitter.encrypt.bits[]},

   encode them by a channel encoding method (or not, if desired) resulting in

   \texttt{transmitter.encode.bits\_total} bits, and place them into the array

   \texttt{transmitter.encode.bits[]}.

4) Spread Spectrum Modulator - these routines are required to take the

   \texttt{transmitter.encode.bits\_total} bits in the array \texttt{transmitter.encode.bits[]},

   spread them using the chosen spreading technique - resulting in
transmitter.spread.chips_total chips, and place them into the array transmitter.spread.chips[].

5) Baseband Data Modulator - these routines are required to take the transmitter.spread.chips_total chips in the array transmitter.spread.chips[], modulate them using the chosen baseband modulation scheme - resulting in transmitter.modulate.samples_total complex data samples, and place them into the arrays transmitter.modulate.data_r[] and transmitter.modulate.data_i[] - the complex signal's in-phase and quadrature portions, respectively.

4.3.2 Channel Module Requirements

The input/output requirements of the channel module is as follows:

1) Digital Waveform Channel - the routines used to simulate channel effects are somewhat different from the rest of the routines. All the routines in this section operate on the same group of data samples. First, a routine transfers the transmitter.modulate.samples_total complex data samples in each of the arrays transmitter.modulate.data_r[] and transmitter.modulate.data_i[] to the arrays receiver.demodulate.data_r[] and receiver.demodulate.data_i[], then each routine in the list of appropriate channel functions operates on these two arrays placing the results back into these arrays.

4.3.3 Receiver Module Requirements

The input/output requirements of the receiver modules are as follows:

1) Data/Spread Spectrum Demodulator - these routines perform a great deal of calculations in order to take the receiver.demodulate.samples_total in the arrays receiver.demodulate.data_r[] and receiver.demodulate.data_i[], remove both the baseband and spread spectrum modulations - resulting in
receiver.demodulate.data_len values in the correlation output arrays, receiver.demodulate.corr_r[] and receiver.demodulate.corr_i[] - the real and imaginary portions of the complex correlation respectively.

2) Channel Decoder - these routines use the receiver.demodulate.data_len values in the correlation output arrays, receiver.demodulate.corr_r[] and receiver.demodulate.corr_i[] to determine the incoming bit stream using either hard or soft decision decoding, place the incoming bit stream into the array receiver.decode.bits[], decode the receiver.decode.bits_total bits using the appropriate decoding method, and place the resulting receiver.decrypt.bits_total bits into the array receiver.decrypt.bits[].

3) Data Decryptor - these routines decrypt the receiver.decrypt.bits_total in the array receiver.decrypt.bits[] using the appropriate decryption algorithm, and place the resulting receiver.sink.bits_total bits into the array receiver.sink.bits[].

4) Information Sink - these routines do not affect the data as the array receiver.sink.bits[] already contains the received bit stream that the receiver has determined had been sent. The routines in this section are concerned with determining errors and updating the error statistics, and (eventually) displaying the operator chosen waveforms, bit streams, transforms, etc.

4.4 Example Function and Setup Modules: AWGN Channel Effect

This is a module taken from the available list of channel effects in the basic platform. Included are both the setup module and the function module for an additive white gaussian noise channel source. Notice how the setup routine has the same name, preceeded by 'SU_'. Also notice how the function module interacts with the data packet. Finally, notice the comments in the beginning of the routine, this documentation will help in maintaining this routine as well as in creating new noise modules.
Noise_ss.c - contains the routines which take care of adding noise to the data samples.

Additive White Gaussian Noise:
- adds A.W.G.N. to each data sample passed to the receiver

<awgn> - enumerated value
void AWGN(data_packet)
struct packet_structure *data_packet;
requires: Eb/No and noise variance have been defined and stored in...
= transmittermodulatordata_r[] and ...data_i[]
- data samples have been transferred to the receiver structure by the transfer data samples routine
results: creates sample values of a Gaussian random sequence with zero mean and variance equal to ...noise_var and adds them to the data samples stored for the receiver:
...demodulatordata_r[] += Gaussian sample values;
and if modulator also uses imaginary channel:
...demodulatordata_i[] += Gaussian sample values;
- note: noise variance is determined by Eb/No, spread spectrum processing gain, and whether complex data samples are used.

(place abstract specs. for additional noise models here)

#include "new_packet.c" /* external declarations and packet structure */

void SU_AWGN(data_packet)
struct packet_structure *data_packet;
{
  double value1, value2, ebn; sig_energy;
  int pn_length;

  /* initialize local variables for ease of code maintenance */
  pn_length = data_packet->transmitter.spreadpn_len;
  ebn = data_packet->channel.awgn_noise.ebno;
  sig_energy = data_packet->transmitter.modulate.norm_energy;

  /* calculate noise variance */
  value1 = 10.0 * log10((double)pn_length * 0.5) - ebn;
  value2 = sig_energy * pow((double)10.0,(double)(value1 / 10.0));

  /* store noise variance */
data_packet->channel.awgn_noise.noise_var = value2;

}

void AWGN(data_packet)
struct packet_structure *data_packet;
{
    /* routine to add AWGN to the data signal */
    double drand48(); /* Lattice library page L44 */
    double number, rand1, rand2, awgn_value;
    double variance;
    int count;
    struct demod incoming;
    enum signal_type signal;

    /* setup local variables to point to portions of the data packet */
    total_number_samples = data_packet->transmitter.modulate.samples_total;
    incoming = data_packet->receiver.demodulate;
    signal = data_packet->transmitter.modulate.signals;
    variance = data_packet->channel.awgn_noise.noise_var;

    /* add noise to all data samples */
    for (count = 0; count < total_number_samples; count++) {
        while((number = drand48()) == 0.0);
        rand1 = sqrt(-2.0 * variance * log(number));
        rand2 = 2.0 * PI * drand48();
        awgn_value = rand1 * cos(rand2);

        incoming.data_r[count] += awgn_value;
    }
    if (signal == complex) {
        for (count = 0; count < total_number_samples; count++) {
            while((number = drand48()) == 0.0);
            rand1 = sqrt(-2.0 * variance * log(number));
            rand2 = 2.0 * PI * drand48();
            awgn_value = rand1 * cos(rand2);

            incoming.data_i[count] += awgn_value;
        }
    }
}
5. Configuration and Output Files

5.1 System Configuration Files

The configuration file for the Spread Spectrum Emulator contains all the information needed to setup a complete system to emulate. This information is organized into a strict format so that it will be read in correctly. The present file structure also allows for expansion; this expansion, however, must retain compatibility with the previous file structures. This file structure is described below along with the procedure for future expansion of the structure.

The file structure is composed of four major components: setup information, transmitter information, channel information, and receiver information. Each of these blocks is started by a specific piece of information, and in the case of the last three, a header string.

The file begins with 20 unused lines reserved strictly for comments; this space is never used by the emulator, so any comments may reside here. These lines are followed by a floating point number representing the version of the configuration file. This allows for different versions of configuration files, thus providing the opportunity for future expansion. New configuration file versions must remain compatible with previous versions, but may contain additional information. The version number is followed by 7 integers representing various parameters of the system. These integers, respectively, represent the parameters: k, k', n, p, l, r, s; where k is the minimum number of input bits to the encoder, k' is the minimum number of input bits needed for the data encryptor, n is the number of output bits from the encoder given k bits were entered, p is the length of the PN sequence, l is the number of bits input to the spreader, r is the number of chips input to the data modulator, and s is the number of samples required per modulated symbol. These parameters are followed by an integer list of '0's and '1's. This list represents the PN sequence to be used by the spreader with the '0's representing '1's and the '1's representing '-1's coming out of the spreader. This mapping of '0's and '1's is arbitrary.
and was chosen from a standard mapping in digital communications. There are at most 15 chips per input line, thus there will be a variable number of lines depending on the length of the PN sequence used. Because of this structure, these lines are not allowed to contain any comments. This length is given above in the parameter 'p', so the number of lines which need to be read can be predetermined from this constant.

Following the list of chips in the PN sequence, there is another list of double precision constants. This list of constants represents the partial response pulse shape required with one sample per line. In the first version this response is limited to a one chip window, however, later versions may include pulse shaping over several chip windows, allowing for testing of effects such as intersymbol interference. The number of lines can be determined from the parameter 's' which represents the number of samples per modulated symbol. Following this list there are 2 more integers. The first is the ordinal value of the enumerated variable 'system_type' which can accept the values (0 [no spreader], 1 [DS spreader], 2 [CCSK spreader]). The second is the ordinal value of the enumerated variable 'signal_type' which can accept the values (0 [real signals] or 1 [complex signals]).

Finally, there are the 2 variables used to determine how many bits are sent through a particular system. As described in section 3.5, these variables contain the number of bits and/or bit errors the emulator must transmit or encounter before terminating the simulation. They are the last numbers to be read in the setup information section of a version 1.0 configuration file. If in the future, more parameters are required, they must be placed after these. By keeping the present parameters in this order, a configuration file of this version can still be read in by an updated version of the emulator. Also, if it is necessary, an older version of the emulator will be able to read in a configuration file developed for a newer version. This flexibility can occur because the setup portion of the emulator will read in all the information that is required to be there by virtue of the version, then it will skip lines until it comes upon a line containing the string "_transmitter_". This string represents the
beginning of the transmitter setup section. By looking for this string, any new setup information can be skipped without causing any errors.

The transmitter setup section contains the remaining data necessary for setting up the transmitter portion of the emulator. For version 1.0, this consists of the contents of the array 'transmitter[]'. The values in here represent the devices which need to be called for the proper implementation of the transmitter. Their order in the array indicates the order in which they must be called in the program and the number of elements in the array is given by the constant 'MAX_NUMBER_FUNCTIONS' defined in the file "new_packets.c". These integers represent the ordinal values of the enumerated variable 'x_mit_functions'. This is all that is required to setup the remainder of the transmitter now; however, if in the future more information is required it will have to be placed after this list.

Following the transmitter information is the channel information. The start of the channel information is indicated by the string "_channel_". Following this string is the contents of the array 'channel_effects[]'. This array is the list of routines, from the enumerated list 'channel_functions', which are to be executed during simulation of the channel. They are to be performed in the order in which they appear. Note, as in the transmitter, the maximum number of functions is the same.

After this list there is a double precision floating point number which represents the desired Eb/No for the A.W.G.N. generator. Following this, the next 40 lines represent the data for the single tone jammers. These are arranged into groups of 4 lines for each of the 10 jammers allowed of this type. The first piece of data in the group is a BOOL(short) variable representing the status of the jammer. The remaining 3 are double precision constants which represent the remaining parts of the tone jammer structure.

At the end of the tone jammer information, there is a blank line followed by the information for all 10 sweeping tone jammers. These are arranged in the same fashion except that there are 7 fields in each group which follow exactly from the sweep jammer
structure defined in "new_packets.c". As the number of modules increases, any additional information needed for the channel must be placed after the sweep jammer information.

The start of the receiver information is indicated by the string "_receiver_". The only new information required for the receiver at this point is the list of routines to perform and the exciser information. The list is similar in structure to the two previous lists and includes functions from the enumerated list 'rcvr_functions'. For the zeroing exciser - an exciser which replaces the frequency bins in a signal's transform, which are above threshold, with zero's - a threshold and a notchwidth are required. The threshold is represented by a double precision constant and the notchwidth is represented by an integer constant. This is the section which will probably undergo the most modifications due to the amount of receiver processing that will be tested, therefore, great care must be taken in modifying this section to insure compatibility still exists with previous versions.

5.2 Sample Configuration File: emsys1.sss

Below is a sample configuration file for a system using a (7,4) Hamming code encoder, a 32 chip PN code, 4 level CCSK, BPSK modulation, -2.5 dB AWGN, no jammers, and no excisers in a transform domain processing demodulator.
1.0 /* Version number (double) */
4 /* (k) number of bits into encoder */
1 /* (k') minimum number of bits to input to encryptor */
7 /* (n) number of bits out of encoder */
32 /* (p) PN code length */
2 /* (l) number of bits into spreader */
1 /* (r) number of chips into modulator */
1 /* (s) number of samples out of modulator */
1111100001101110
1010000100101111
00
1.0 /* Pulse Shaping (double) */
2 /* System type (no,ds,ccsk) */
0 /* Signal type (real,complex) */
50000 /* (num_bit) number of bits to xmit before end (integer) */
-1 /* (num_err) number of errors to detect before end (integer) */

_transmitter_
1 /* Symbols_rand */
2 /* No_encrypt */
9 /* No_encode */
6 /* ccsk_spread */
7 /* bpsk_mod */
0 /* no_xmit_fn */
0 /* no_xmit_fn */
0 /* no_xmit_fn */

_channel_
1 /* awgn */
0 /* no_channel_fn */
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 /* no_channel_fn */

-2.5 /* Eb/No (dB) (double) */
0 /* tone jammer 1 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength_dB (double) */
0 /* tone jammer 2 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* tone jammer 3 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* tone jammer 4 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* tone jammer 5 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* tone jammer 6 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* tone jammer 7 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* tone jammer 8 status (FALSE) (short) */
0.0 /* phase (double) */
0.28 /* frequency (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 1 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 2 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 3 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 4 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 5 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 6 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 7 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 8 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 9 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */
0 /* sweep jammer 10 status (FALSE) (short) */
0.0 /* phase (double) */
0.45 /* phase_increment (double) */
0.25 /* min_frequency (double) */
0.75 /* max_frequency (double) */
0.002 /* sweep_rate (double) */
20.0 /* strength dB (double) */

_receiver_
1 /* fd_demod_noex */
4 /* no_decode */
3 /* no_decrypt */
0 /* no_rcvr_fn */
0 /* no_rcvr_fn */
5.3 System Output Files

As the emulator finishes a simulation, it has final statistics for that particular system's performance. In this basic platform, these statistics consist of 6 separate numbers. They are: information bit count, information bit error count, symbol count, symbol error count, channel bit count, and channel bit error count. The information bits are the bits generated by the information source module. The symbols are the symbols formed by the bits from the information source. This measure is used to determine error rates if the basic unit of information sent is represented by symbols rather than bits, i.e. ascii characters. Finally, channel bits are the bits sent through the channel; those generated by the channel encoder. This allows performance checks on the channel encoders by providing the actual bit error rate of the channel; this can then be compared to the information bit error rates to determine the effect of the encoder.

Also included in the output files is the name of the configuration file used to generate these performance statistics. This was required in order to organize the information obtained from the emulator. This allows the operator to see what parameters and devices were used to generate these statistics.

5.4 Sample Output File: emsys1.out

This is a simple output file; however, there is no limit to the amount of information that can be stored in these files. If in the future, more information is required to be saved, it can be appended to this file. As long as this structure remains intact and in this order,
previously written observation modules - modules which display emulator information -
can interface with these new output files easily.

This is what a typical output file would look like. This basic structure must be
maintained when augmenting the output file structure.

```
emsys1.sss
bit count = 52248
bit errors = 157
symbol count = 13062
symbol errors = 81
raw bit count = 91434
raw bit errors = 439
```
6. Future Expansion

6.1 Integration of New Modules into the System

Integration of new modules requires modification of a few sections of the emulator. Below is a list of the changes that have to be made in order for a new module to be included into the system.

1) Create the new function module and setup module - The first step in integrating a new module is to develop the software model for the module, determine the memory requirements, and write the code for the function module and its initial setup routine. The new function and setup modules need to follow the guidelines included with the sample module in section 4.

2) Add to the list of functions in new_packet.c - Once the routines have been written, they are given a name (different from the procedure's name in the code to prevent any confusion by the compiler) to be used in the enumerated variable list (for examples, see the data_packet listing in section 3.7). This name is placed into the proper enumerated list, depending on what type of function this module performs (i.e. transmitter, channel or receiver function). This is the name that will be used to reference this routine throughout the case statements in the code.

3) Augment the case statements - The final step in integrating a new module into the system is to create the proper calling commands. These commands are located in the routines: transmitter(), Setup_transmitter(), channel(), Setup_channel(), receiver() and Setup_receiver(). Depending on what function the new module performs, it, along with its setup routine, will be placed in a pair of these calling routines. The actual commands will be part of the selections in the appropriate case statement; an example of which will follow in the next section.
These are the only steps that have to be followed in order to have a newly created module interact properly with the existing emulator. The next section contains sample commands for integrating new modules into the existing structure.

6.2 A New Module Integration Example

For this section assume that a new module has been created which performs the function of encoding data as a Hamming(7,4) encoder. Also, assume that its setup routine has been created too. They've been named `hamming_7_4()` and `Setup_hamming_7_4()`. Therefore, the next step in integrating these modules is to augment the enumerated variable `x_mit_functions` (since the is a transmitter routine). For this, `hamm_7_4` will be added to the list. Note, by using the enumerated variable in this way, this new value can be placed anywhere in the list after the `no_xmit_fn`, allowing similar routines to be grouped near each other.

Next, the routines `transmitter()` and `Setup_transmitter()` need to be updated to include calls to these new functions. Another case statement of the main switch function has to be added for the main routine, `hamming_7_4()`. This case statement looks like this:

```c
    case hamm_7_4: {
        hamming_7_4(data_packet);
        break;
    }
```

Similarly, in `Setup_transmitter()`, these lines need to be added:

```c
    case hamm_7_4: {
        Setup_hamming_7_4(data_packet);
        break;
    }
```

Once again, since the functions to perform are chosen from the enumerated variable list, these case statements can be placed anywhere in the list of existing statements.

Now, assuming that the routines work properly (and an appropriate decoded has been created and added to the receiver section similarly), the emulator can be set up to use
this Hamming(7,4) coder along with the other modules chosen for the remainder of the system.

6.3 Conclusion and Future Directions

With the completion of the basic platform, the emulator will be able to simulate relatively simple communication systems; however, a sound structure will be in place which will allow for a great deal of expansion in the future. This structure has been created with the concept of future expansion as a main focus so that the procedure required for expanding the system is relatively simple. This fact should soon make this tool a valuable resource for quickly creating simulations to test new devices or concepts under current investigation.

While development of new modules proceeds, simultaneous development of the graphic interface can also occur. The structure of the emulator allows it to run without the graphic interface so that it can be used to test new routines immediately; however, the complete emulator is envisioned to include an extensive graphic interface which will allow it to (1) display all information graphically, in the form of waveforms, transforms, bit error curves, etc. and (2) have all simulation systems set up graphically through the use of a graphic pointing device such as a mouse. The latter attribute will make the emulator more "user-friendly", while the former will provide valuable insight into the inner workings of the system under investigation by providing the opportunity to observe of the data as it passes through the system. Both of these facets will help to make the emulator more valuable as a research tool.