Technical Report CS/E-88-029 August 1988

ANNE USER MANUAL

Casey S. Bahr
Oregon Graduate Center
Dept. of Computer Science & Engineering
Beaverton, Oregon 97006-1999
(503) 690-1151

ANNE
USER MANUAL

Version 1.0
August 4, 1988

TABLE OF CONTENTS

LIST OF FIGURES

User Manual for

ANNE - Another Neural Network Emulator:

A program to simulate neural networks on the iPSC

1. General Description of Usage

ANNE is one of two simulators in the CAP (Cognitive Architecture Project) group's neural network development environment that run on the iPSC [Jag88][May88][BHJ88].[1] The development of a neural network simulation with ANNE is composed of the following general steps:

(1) A network structure must be described in a standardized network specification format known as **BIF** (for Beaverton Intermediate Format). The easiest way to create a BIF file is through the use of **NDL**, CAP's Network Description Language (pronounced "noodle"). NDL creates connectivity graphs of basic network objects known as connection nodes, or **CNs**, which can then be translated into BIF as well as other useful formats [Joh88b][Joh88a].

(2) A BIF file must have its CNs mapped to the physical processors of the target machine, in this case the iPSC. The node processors of the iPSC are called **HNs** (for hypercube nodes) in this manual. **Mapper** is a CAP program used for mapping CNs to HNs.

(3) A mapped BIF file (called **MIF**) is further processed by a utility called **bifsplit** that allows ANNE to parse a network specification in parallel.

(4) User code, which describes the algorithm of the network to be simulated, must be linked with ANNE's image prior to runtime. The user code consists of host and node portions.

(5) Finally, usual iPSC procedures are followed to load the cube nodes and start ANNE's host code.

The details of this somewhat lengthy procedure will become clearer in later sections of this manual. Other manuals should be referenced that describe the use of NDL, Mapper, and the iPSC. For a thorough description of ANNE's design and implementation see [Bah88].

2. BIF Network Models

A BIF file represents the connectivity specification of a neural network. BIF attempts to give the user both generality and compactness in modelling neural networks with one standardized format. BIF was originally derived from data structures used for a neural network simulator done at Univ. of Rochester [SSB83][Fan86]. This file contains the "raw" structure of the user's network in terms of three basic objects: CNs, sites, and links. Unlike Rochester's version, functions associated with these objects are not specified in BIF. Such functions are explicitly declared in the user code linked with ANNE. Only a cursory description of BIF is given here to

[1] ANNE's source code may also be compiled to run on the iPSC/2, but this version is not optimized for that machine.

make subsequent sections of this manual more understandable in terms of the data model used in simulations. A later section on BIF grammar gives a more detailed specification of what a BIF file actually contains.

2.1. CNs, Sites, and Links

The basic network object is the CN. Each CN may have one or more **sites**, and each site has one or more **links**. A CN is, thus, made up of three sub-parts. The main CN fields pertain to the global state of an entire CN. A second sub-part is the site, which groups the links and holds intermediate values derived from the values received on those links. Links are actually shared between CN structures. That is, the link specification attached to a particular CN is just one terminus of an entire link. Each link terminus holds the "address" of its other end and identical weight and value information. Thus, links can easily be used in a bidirectional manner. Figure 2.1 shows a pictorial model of a CN.

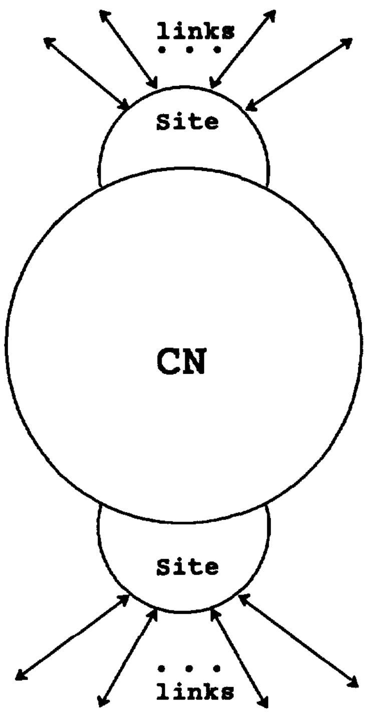

Figure 2.1: Conceptual structure of a CN and its sub-components.

2.2. BIF Files

A BIF file has two parts. The first contains a listing of the **CNgroups**, each of which consists of a unique group index, a string name, and two initialization values for CNs belonging to the group. Each CN carries an index corresponding to the group to which the CN belongs. The group name allows the user to address groups of CNs symbolically. Each BIF file used with ANNE must have two special CN groups named "input" and "output". These groups designate the CNs used for global I/O operations.

The second part of a BIF file consists of individual CN records. These records are composed of a hierarchy of CNs, sites, and links. Sites nest within CNs, and links nest within sites. Input or output sites are not listed in any particular order. Neither sites nor links are explicitly indexed in a BIF file.

Sites happen to be specified in BIF as being either input or output. This designation is ignored in ANNE, where sites are bidirectional. In a typical case for an input site (or bidirectional site being used for input), a site's link values are summed together and the resulting intermediate value is used in internal CN calculations. An output site may be used to broadcast the CN's output value, if any, along the links attached to the output site.

As noted previously, each link specification within a single site is the specification of one of two ends of a single connection between two CNs. Each link terminus contains the index of the CN, site, and link of the corresponding terminus at the other end of the link. All links have a *weight* that can be used to modify values passing along the link.

3. Runtime Simulation Model

3.1. Simulation Clocking

Two alternative synchronization methods were considered for ANNE. Both alternatives are designed for distributed event-driven simulations. These alternatives were Misra's algorithm [Mis86] and Time Warp [JeS82]. Both are designed for general-purpose simulations that require exact timing synchronization. ANNE relaxes the constraints imposed by these complex, communication-intensive algorithms by exploiting the fault-tolerance of connectionist networks that employ a distributed representation of their data [MRH86].

Rather than event-driven, ANNE utilizes *message-driven* timing and synchronization. This scheme yields minimal message overhead, faster simulations, and no possibility of deadlock. This technique gives the user the ability to tune the parameters of synchronization to suit the particular network model at hand. The basic premise that makes this method possible is that CNs are able to produce outputs even if all their inputs have not arrived or have arrived out of time-order. At all times there are available at each CN's input links a value, whether it be a previously used value or a new one.

Simulations are run using $n + 1$ closely coupled clocks, one *global* clock in the iPSC host processor, and one *local* clock at each HN. At the HN level, each complete sub-network cycle constitutes a single time step. At *synch points* all HN clocks are set to the value of the global clock. The distance between these global synchronization points is referred to as the *synch_count*. After executing *synch_count* cycles,

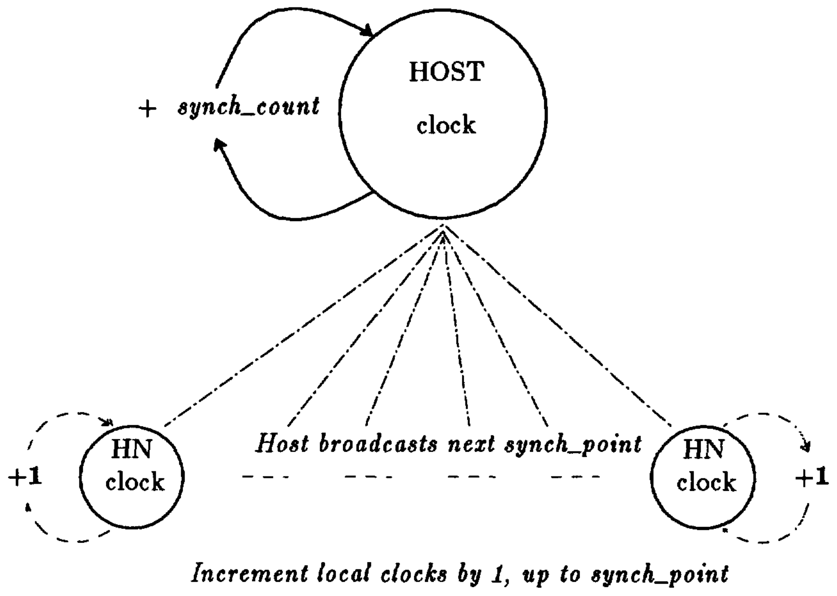
HOST
clock
+ synch_count
HN
clock
+1
Host broadcasts next synch_point
HN
clock
+1
Increment local clocks by 1, up to synch_point

Fig: 3.2 ANNE's simulation clocking model

each HN synchronizes with the host processor before continuing (see figure 3.2).

3.2. CN-to-CN Communication

Messages (output values) generated by CNs are stamped with the local HN time. These time-stamps are not used to strictly order messages between CNs. Rather, a *message_window* is defined that "slides" along with the local HN clock. When a message is received by a CN its time-stamp is compared with the window to determine its acceptability (see figure 3.3). The *message_window*, *synch_count*, and *synch_point* are set by the user at runtime and used to "tune" network performance.

The generation and reception of CN messages is interleaved in order to avoid an overflow of HN message buffer space. Interleaving message generation and reception also acts to balance the overall synchronization between HNs. Once message generation begins, message reception and processing is driven by the arrival of more messages. Continued reception of messages is controlled by the setting of a message probe *timeout*. After processing messages in reception mode, *timeout* probes are made for more incoming messages before resuming message generation. The *timeout* value is a user-controlled parameter.

In general, a larger *synch_count* elicits better performance from ANNE. This is especially true when running networks in "learning" mode where it takes many cycles to achieve convergence of the network. Experience has shown that it is best to set the *message_window* to about half the *synch_count*, or smaller. Network performance may also be sensitive to the setting of the probe *timeout*. The *timeout* acts

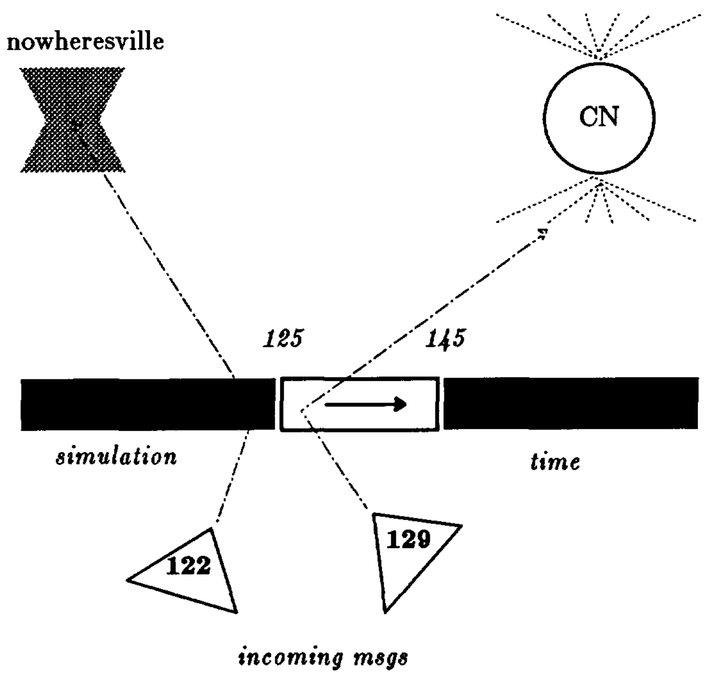

Fig: 3.3 Conceptual model of the time-stamp window used in ANNE

as a throttle on simulations, often resulting in more synchronized performance for larger configurations of the iPSC.

3.3. Message Packetization

Messages between CNs that reside on different HNs are placed in packets holding from 1 to 63 CN messages. Larger packets utilize the iPSC communication facilities more efficiently. The size of these packets is user-controlled and is called *maxmsgnum*.

Besides setting the maximum size of message packets, ANNE provides two major options controlling the behavior of message packaging. The difference between these options has to do with the relationship between the local simulation time and the time-stamps on the messages at the originating HN. In the first method, *synchronous packetization* or *SP*, all message packets are sent to their destination HN during a single clock cycle. With SP, any remaining message packets are flushed at the end of each network cycle, regardless of their current size. Thus, the messages contained in each packet carry the same time-stamp.

In contrast, *asynchronous packetization* or *AP*, is designed to obtain maximum utilization of message packet space. The AP method only ships message packets when they have reached the maximum capacity set by the user in *maxmmsgnum*. No packet flushing occurs at the end of each network cycle. Messages with different time-stamps may be found within the same packet. AP, in general, will improve the speed of simulations at the cost of increased asynchrony in CN-to-CN communications. For the feed-forward mode of most neural network models, the SP method is most appropriate.

4. User Code

4.1. User Procedures

Two procedures are needed to implement a network algorithm in ANNE. The *convergence procedure* checks the global output of the network against some user-defined measure to see if the network has converged to a state that requires a new input vector. ANNE, of course, delivers the first input vector of a simulation without checking the convergence state of the network. The *network procedure* resides in ANNE's HN image and acts as the script for an entire network cycle from the presentation of global input by the host to the sending of global output from the CNs designated as "output" CNs back to the host. The network procedure is identical for each iPSC node. Neither procedure needs to be changed to accomodate different dimensions of the iPSC. Simply re-mapping a network's BIF specification is all that is needed to run networks on different iPSC dimensions.

This version of ANNE allows the user to load two versions of each of the above procedures before runtime. Thus, one pair of convergence and network procedures might be used to simulate the learning phase of a network, while the second pair is used for the network's feed-forward phase. The user can toggle between these pairs at runtime. Special names for these procedures must be used. The convergence procedures must be named `Convergence1()` and `Convergence2()`. The network procedure names are `User_fx1()` and `User_fx2()`. The network procedures must also include initialization routines, named `Init_userfx1()` and `Init_userfx2()`. These allow the user to initialize her own parameters and data structures at the start of each network run. `Init_userfx*()` is called once by ANNE.

At runtime, ANNE will prompt for the set of procedures to be used for a particular run, designated by a "1" or "2". Auxiliary utilities, described later, can be used to facilitate the loading of user procedures with ANNE. These utilities can also load default "dummy" procedures should the user be interested in loading only a single set of user procedures.

When writing the convergence procedure, the user has access to the simulation parameters, the input vector, output vector, and target vector, if any. `Convergence*()` returns 1 if convergence is met, 0 otherwise. The network procedure has access to the simulation parameters, and all the network data, which are stored in structures modelled after the BIF format. A number of ANNE system calls are also provided to the user's network procedure, mainly to effect CN-to-CN communication. Although some of these system calls can be used to on individual CNs, other calls have been provided to encourage the user to write the network procedure as operations on groups of CNs, for instance the layers in a back-propagation model. The details of writing these procedures is covered in more detail in the following sub-sections.

4.2. ANNE System Calls

Here is a listing of the ANNE system calls provided for the user's `User_fx*()` procedure.

`Send_node_output(cn_index,site_index);`
Sends the output from a single CN along all the output links belonging to the named site.

`Send_group_output(group_name,site_index);`
Performs the same operation as `Send_node_output()` for a named group of CNs.

`Send_net_output(filename);`
Used specifically for global network output to the host process. The output vector is written by the host to the named file.

`Update_node_weights(cn_index,site_index);`
If the user has set the link weights on a particular node and the link is used bidirectionally the user makes this call to transmit the new weights to the other end of the links. The CN and site indices in the parameter list name the group of links that transmit their weights.

`Update_group_weights(group_name,site_index);`
Performs the same function as `Update_node_weights()` for the named group of CNs.

`cnlist * Get_cnlist(group_name);`
Returns a structure that holds both a list of the local CN indices belonging to the group name and the number of CNs in the list.

Following is an example of the network procedures for a straightforward back-propagation network model. These procedures will work for any three-layer back-propagation network up to the limitations of ANNE's data structures. Note that function calls beginning with an upper-case letter and in **this font** are system calls provided by ANNE, others are written by the user. The code for the user procedures has been included.

4.3. Network Procedure: Node Level

```
#include "userfx.h"
#include <math.h>
#define TOP 1
#define BOTTOM 0

/* cnlists are lists of the cn indices present on the local HN */
static cnlist *cns_in, *cns_hid, *cns_out;
/* user's network functions */
```

```
void Input_site_fx(), Other_site_fx(), Assign_to_outsite(), Squash();
void Calculate_output_error(), Calculate_other_error();
void Init_user_fx1();
short Activate();
/* simulation parameters accessed by the user code */
extern cycle_params cp;
/* buffer for iPSC log messages */
char sb[80];

/*********************************************************/
/* Init_user_fx1: Called once by ANNE to init user data */
/*********************************************************/
void Init_user_fx1()
{
    int i;

    cns_in  = Get_cnlist("input");
    cns_hid = Get_cnlist("hidden");
    cns_out = Get_cnlist("output");
    Update_group_weights("hidden",TOP);
    Update_group_weights("hidden",BOTTOM);
    /* no fault simulation */
    faulting = OFF;
}
/*********************************************************/
/* User_fx1: for back-prop network.                      */
/* This procedure describes the "script" modelling the   */
/* network's behavior for a single network cycle.        */
/*********************************************************/
void User_fx1()
{
    int cnx, i, l;
    float ferr, upd, fout;

    /** FORWARD PASS **/
    /* get an input vector from the host */
    Input_site_fx();
    Send_group_output("input",TOP);

    /* sum weighted inputs at hidden layer, activate, and send output */
    Other_site_fx(cns_hid,BOTTOM,1);
    Squash(cns_hid,BOTTOM);
    Assign_to_outsite(cns_hid,TOP);
    Send_group_output("hidden",TOP);

    /* sum weighted inputs at output layer, activate, send to host */
    Other_site_fx(cns_out,BOTTOM,1);
    Squash(cns_out,BOTTOM);
    Assign_to_outsite(cns_out,TOP);
    Send_net_output();

    /** BACKWARD PASS **/
    Calculate_output_error(cns_out);
    for(i = 0; i < cns_out->numcns; i++) {
     cnx = cns_out->cns[i];
     SITEVALUE(cnx,BOTTOM) = ERROR(cnx);
    }
    Send_group_output("output",BOTTOM);

    /* received error from output layer */
    /* sum weighted error signals */
    Other_site_fx(cns_hid,TOP,0);
    /* calculate this layer's error and send down */
    Calculate_other_error(cns_hid);
    for(i = 0; i < cns_hid->numcns; i++) {
     cnx = cns_hid->cns[i];
     SITEVALUE(cnx,BOTTOM) = ERROR(cnx);
    }
    Send_group_output("hidden",BOTTOM);

    /* send new weights to other end of links */
    Update_group_weights("hidden",TOP);

    Other_site_fx(cns_in,TOP,0);
    Calculate_other_error(cns_in);
    /* send new weights to other end of links */
    Update_group_weights("input",TOP);
} /* end User_fx1() */
/**********************************************************/
/* Calculate_other_error                                  */
/**********************************************************/
void Calculate_other_error(cnl)
cnlist *cnl;
{
    int cnx;
    short i, local_error;
    float ferr,fout;

    for(i = 0; i < cnl->numcns; i++) {
     cnx = cnl->cns[i];
     fout = SHORT_TO_FLOAT(OUTPUT(cnx));
     ferr = SHORT_TO_FLOAT(SITEVALUE(cnx,TOP)) * DERIV(fout);
     ERROR(cnx) = FLOAT_TO_SHORT(ferr);
    }
}
/**********************************************************/
```

```
/* Calculate_output_error                                       */
/***************************************************************/
void Calculate_output_error(cnl)
cnlist *cnl;
{
    int cnx;
    short i, local_error;
    float ferr,fout;

    for(i = 0; i < cnl->numcns; i++) {
     cnx = cnl->cns[i];
     local_error = targetvals[cnx] - OUTPUT(cnx);
     fout = SHORT_TO_FLOAT(OUTPUT(cnx));
     ferr = SHORT_TO_FLOAT(local_error) * DERIV(fout);
     ERROR(cnx) = FLOAT_TO_SHORT(ferr);
    }
}
/***************************************************************/
/* Input_site_fx                                                */
/***************************************************************/
void Input_site_fx()
{
    int cnx;
    short i, siteval;

    /* global inputs are ready and waiting in site value */
    if(cns_in != (cnlist *)NULL) {
     for(i = 0; i < cns_in->numcns; i++) {
         cnx = cns_in->cns[i];
         /* output function is identity */
         OUTPUT(cnx) = SITEVALUE(cnx,0);
         if(faulting) (void)flt_cn(cnx,&OUTPUT(cnx));
     }
    }
}
/***************************************************************/
/* Other_site_fx                                                */
/***************************************************************/
void Other_site_fx(cnl,site_index,direc)
cnlist *cnl;
short site_index;
int direc;
{
    int cnx, siteval, i;

    if(cnl != (cnlist *)NULL) {
     for(i = 0; i < cnl->numcns; i++) {
         cnx = (int)cnl->cns[i];
```

```
              if(direc != 1) {  /* make weight change while error in inval */
             Weight_change(cnx,site_index);
            }
            /* weight and sum inputs be they error or output */
            siteval = Sum_inputs(cnx,site_index);
            if(faulting) {
              (void)flt_site(cnx,site_index,(short *)&siteval);
            }
            SITEVALUE(cnx,site_index) = siteval;
      }
    }
}
/*********************************************************/
/* Squash: applies the activation function to the      */
/* output and assigns the result to the output site.   */
/*********************************************************/
void Squash(cnl,site_index)
cnlist *cnl;
short site_index;
{
     int  cnx, i;

     if(cnl != (cnlist *)NULL) {
       for(i = 0; i < cnl->numcns; i++) {
            cnx = cnl->cns[i];
            OUTPUT(cnx) = Activate(cnx,SITEVALUE(cnx,site_index));
            if(faulting) (void)flt_cn(cnx,&OUTPUT(cnx));
       }
     }
}
/*********************************************************/
/* Assign_to_outsite: assign output to output site     */
/*********************************************************/
void Assign_to_outsite(cnl,site_index)
cnlist *cnl;
short site_index;
{
     int cnx;
     int i;

     if(cnl != (cnlist *)NULL) {
       for(i = 0; i < cnl->numcns; i++) {
            cnx = cnl->cns[i];
            SITEVALUE(cnx,site_index) = OUTPUT(cnx);
       }
     }
}
/*********************************************************/
```

```
/* Activate: CN activation function.                      */
/*********************************************************/
short Activate(cnx,siteval)
int cnx;
short siteval;
{
    double dblval;
    char s[80];

    dblval = SHORT_TO_DOUBLE(siteval);
    /* don't blow up exp() */
    if(dblval < -30) {
     return(0);
    }
    if(dblval > 30) {
     return(500);
    }
    dblval = 1.0/(1.0+exp(-1.0*dblval));
    return(DOUBLE_TO_SHORT(dblval));
} /* end Activate() */
```

4.4. Convergence Procedure: Host Level

Here is the corresponding convergence procedure that runs in the host:

```
#include "convergence.h"
FILE *fpcyc; /* print out cycle data to this file */
/* used to convert standard BIF vectors to floating point */
double dbl_err[NCNS], dbl_out[NCNS], dbl_targ[NCNS];
static int total_cycles = 0;
char alf[16] = {'A','B','C','D','E','F','G','H',
                'I','J','K','L','M','N','O','P'};
/* alf is for use in character recognition network */
/*********************************************************/
/* Convergence1: for back-prop nets.                      */
/* return 1 if converged, 0 if didn't                     */
/*********************************************************/
int Convergence1()
{
    int i,mark,ok = 1, maxind;
    double ferr, maxout;

    /* detect if each node is within defined limits */
    fprintf(stderr,"\nTARGET: ");
    for(i = 0; i < numoutputs; i++) {
     dbl_targ[i]  = INT_TO_DOUBLE(targetvec[i]*SCALE);
     dbl_out[i]   = INT_TO_DOUBLE(outputvec[i]);
     dbl_err[i]   = dbl_targ[i] - dbl_out[i];
     errorvec[i] = DOUBLE_TO_INT(dbl_err[i]);
```

```
      /* fabs() didn't fx properly */
      ferr = dbl_err[i];
      if(ferr < 0.0) ferr = -1 * dbl_err[i];
      if (ferr > cp.err_factor) { /* then too much error */
          ok = 0;
      }
      fprintf(stderr,"%2.2f ",dbl_targ[i]);
     }
     /* print output vector to screen at each synch point */
     fprintf(stderr,"\nOUTPUT: ");
     for(i = 0; i < numoutputs; i++) {
      fprintf(stderr,"%2.2f ",dbl_out[i]);
     }
     if(ok) {
      maxout = dbl_out[0];
      maxind = 0;
      for(i = 1; i < numoutputs; i++) {
          if(dbl_out[i] > maxout) {
           maxout = dbl_out[i];
           maxind = i;
          }
      }
      total_cycles += cp.numcycles;
      fprintf(stderr,"CONVERGED on %c in %d cycles,total cycles=%d\n",
                             alf[maxind],cp.numcycles,total_cycles);
      fpcyc = fopen("cycles","a");
      fprintf(fpcyc,"CONVERGED on %c in %d cycles,total cycles=%d\n",
                             alf[maxind],cp.numcycles,total_cycles);
      fclose(fpcyc);
     }
     return(ok);
}
```

4.5. Header File for User Procedures

```
#if VAX
#include <stdio.h>
#endif
#include "nglob.h"

/* ANNE system calls */
extern void Send_node_output(), Send_group_output(), Send_net_output();
extern void Update_node_weights(), Update_group_weights();
extern cnlist *Get_cnlist();
/* local CN table */
extern CNentry CN[MAX_CNS];
/* used for target vector, if any */
extern short targetvals[MAX_CNS];
```

```
/* structure holding user-controlled simulation parameters */
extern cycle_params cp;
/* faulting flag */
extern int faulting;

/* USER MACROS */
/* for simplified access to CN table fields */
#define DELAY(cn) CN[cn].C->delay
#define HISTORY(cn) CN[cn].C->history
#define RESTPOT(cn) CN[cn].C->restpot
#define POT(cn) CN[cn].C->pot
#define STATE(cn) CN[cn].C->state
#define OUTPUT(cn) CN[cn].C->output
#define ERROR(cn) CN[cn].C->error
#define SD(cn) CN[cn].C->sd
#define SITEVALUE(cn,s) CN[cn].sites[s].value
#define SITENLINKS(cn,s) CN[cn].sites[s].nlinks
#define LINKPTR(cn,s,l) &CN[cn].sites[s].links[l]
#define LINKVEC(cn,s,l) CN[cn].sites[s].links[l].lnkvec
#define LINKHISTORY(cn,s,l) CN[cn].sites[s].links[l].history
#define LINKWEIGHT(cn,s,l) CN[cn].sites[s].links[l].weight
#define LINKINVAL(cn,s,l) CN[cn].sites[s].links[l].inval
#define SETWTUP(cn,s,l) CN[cn].sites[s].links[l].lnkvec|=LV_WTUP

/* used to convert int fields to float and vice versa */
#define SHORT_TO_FLOAT(s) (((float)s)/(float)SCALE))
#define FLOAT_TO_SHORT(f) ((short)(f * (float)SCALE))
#define SHORT_TO_DOUBLE(s) (((double)s)/(double)SCALE))
#define DOUBLE_TO_SHORT(d) ((short)(d * (double)SCALE))
#define DERIV(f) (f * (1 - f))
```

4.6. User Accessible Data Structures

In writing the network procedure the user has access to local data structures, including those holding the CNs. These structures are modelled closely after the BIF format. The following data structures can be accessed by the user:

```
/* entry in local cube node CN table */
typedef struct {
  unsigned char hn;      /* hypercube node index where CN lives */
  SFWL *sites;           /* array of sites belonging to this CN */
  CFWN *C;               /* pointer to a CN structure           */
}CNentry;

CNentry CN[MAX_CNS];     /* table of local CNs */

/* a link */
typedef struct {
  unsigned char lnkvec;  /* bit vector for this link */
  char history;            /* recent history of link   */
  BYT4 cn;                 /* CN this link goes to     */
  short site;              /* site this link goes to   */
  short link;              /* link this link goes to   */
  float weight;            /* weight value for link    */
  short inval;             /* input value to this link */
} LFWI;

/* a site */
typedef struct {
  short value;             /* result of site function  */
  unsigned char sitevec;   /* bit vector for this site */
  short nlinks;            /* number of links attached */
  LFWI *links;             /* pointer to links array   */
}SFWL;

/* a CN */
typedef struct {
  char  group;             /* group this CN belongs to */
  BYT4 index;              /* unique CN index          */
  short procid;            /* cube processor for CN    */
  short delay;             /* delay of output message  */
  unsigned char bitvec;    /* bit vector for this CN   */
  char  history;           /* recent history of CN     */
  short restpot;           /* resting potential        */
  short pot;               /* potential                */
  char  state;             /* current state of CN      */
  short output;            /* current output value     */
  short error;             /* error value              */
  short sd;                /* statistical deviation    */
  short nsites;            /* number of sites on CN    */
}CFWN;

/* struct passed to user from Get_cnlist(groupname)   */
/* recommended that a cnlist be allocated statically */
typedef struct {
  int numcns;             /* number of CNs in list     */
  BYT4 *cns;              /* list of CN indices        */
}cnlist;

/* simulator's synchronization parameters */
/* shared by both the host and nodes      */
/* recommended that these not be assigned */
typedef struct {
  short global_clock,      /* global simulation clock (host) */
     local_clock,      /* local simulation clock (nodes) */
     msg_window,       /* window for valid cn<->cn msgs  */
     synch_count,      /* # of local clock cycles to run */
     synch_point,     /* global_clock + synch_count - 1 */
     checkpoint;      /* synch_point to break at (host)  */
} cycle_params;

cycle_params cp;

/* node vectors only available in the host */
int *targetvec,            /* target vector length=numoutputs */
    *outputvec,            /* output vector length=numoutputs */
    *errorvec,             /* error vector  length=numoutputs */
    *inputvec;             /* input vector  length=numinputs  */

int numoutputs,            /* number of output CNs */
    numinputs;             /* number of input CNs  */
```

5. Runtime Commands

buildnet {-tvlfd{1 | 2 | 3 | x}}

Constructs and initializes the network and auxiliary data structures and leaves the network in a suspended state. The flags have the following meanings:

t - Turn on timing information, such as measuring the number of connections per second.

v - Turn on "verbose" mode. Allows a dump of the BIF file with all fields labelled.

l - Turn on logging. Although this option is not extensively implemented in ANNE's code, it can allow logging to a default file of the actions taken by ANNE during a simulation.

f - Turn on fault calls. Used in conjunction with FltSim for simulating faulted networks.

d - Turn on various debug options for debugging ANNE itself. Additional qualifiers may be used individually or combined to effect the following results:

> 1 - turns on debug statements in ANNE's host level code.
>
> 2 - turns on debug statements in ANNE's node level code.
>
> 3 - turns on "verbose" mode in each HN, similar to **v** above. The individual BIF sub-files sent to each HN are labelled and recorded in the iPSC log file.
>
> **x** - turns on debug statements associated with running ANNE in faulted mode, i.e. with FltSim.
>
> Without additional qualifiers **d** turns on all debug options, except **x**.

newrun

Begins a simulation run. The user is prompted for the names of the input vector, the target vector (if any), and output vector file names. The user sets the size of message packets and the length of timeout for receiving messages.

startnet

Activates the simulation. The user is prompted to set *synch_count*, *message_window*, and the next *checkpoint*. The *checkpoint* is checked to see that it corresponds to a global synchronization point. A checkpoint may be set such that the simulation works in step mode, that is, it only progresses one clock cycle before suspending.

stopnet

Causes the network to suspend at the next global synchronization point whether or not it has reached a preset *checkpoint*. It may be restarted with the **startnet** command.

savenet

Saves the current network structure (including modified weights, etc.) in a new BIF file, which can be used later for a new simulation. The user is prompted for the name of the save file.

show

Checks the state of the "local" simulation parameters and lists the currently active *traces* (see below).

quit

Causes ANNE to exit. Once a network is activated (by **startnet**) summary statistics are sent from the HNs and written to a summary file, then ANNE exits.

help

Prints a list of ANNE's commands.

The user gains access to network data via *CN maps*. These are named groups of CNs. The default CN maps are the `CNgroup` blocks found in a BIF file. Attached to each CN group is a list of the CNs that belong to it. By giving the name of a CN group to certain user operators, either via the terminal or at the node level through the user network code, entire groups of CNs are addressed. Currently, only the retrieval and assignment of "cn" fields is fully implemented.

The following are user commands that can be invoked when the network is in a suspended state. They operate on CN maps.

print <*struct_name*> <*field_name*> <*CNmap_name*>

Prints the current value of *struct_name.field_name* for each CN in the named CN map.

assign <*struct_name*> <*field_name*> <*CNmap_name*>

Assigns a value to *struct_name.field_name* in each CN in the named CN map. The user is prompted for a CN index and enters a value for each CN.

trace <*struct_name*> <*field_name*> <*CNmap_name*>

This command acts similarly to **print** except that the current values for each CN in the CN map are printed at each global synchronization point, according to *synch_count.*

untrace <*trace_number*>

The **show** command lists all active *traces* with a corresponding *trace_number*. These numbers can be passed to **untrace** to delete a particular trace.

reset {<*CNmap_name*> ¦ *all*}
Resets the link weights in a particular CN map or the entire network (implementation incomplete).

eucldist <*CNmap_name*> <*CNmap_name*>
Gives the Euclidean vector distance between two CN maps, for example, the hamming distance (implementation incomplete).

intersect <*CNmap_name*> <*CNmap_name*>
Gives the conjunction of two CN maps (implementation incomplete).

union <*CNmap_name*> <*CNmap_name*>
Gives the disjunction of two CN maps (implementation incomplete).

makemap <*CN_range*> <*CNmap_name*>
Allows the user to define CNmaps other than the default CN maps built when the simulator is initialized (implementation incomplete).

6. Auxiliary Utilities

loadconv convfx1.c convfx2.c

loaduf userfx1.c userfx2.c

These utilities ease the job of compiling and linking the convergence procedure and the network procedure with ANNE's host and node images, respectively. ANNE can be loaded with two versions of each type of network procedure. Thus, each utility expects two file names as arguments. If only one file name is given, a default (empty) procedure will be loaded. The named 'C' files will automatically be compiled and linked in with ANNE, according to the makefile supplied with ANNE's code. At runtime the user can switch between the sets of user functions with the newrun command.

ndl may be used to create back-propagation networks. This version of ndl only creates one this type of network. Type "ndl" to see its usage.

mapdim is an alternative mapping utility to Mapper. Mapdim takes a raw BIF file, created by NDL (or ndl) and adds the appropriate processor ids to the CN specifications according to a simple algorithm. In effect, mapdim slices the network vertically into equal partitions. The user may supply her own CN-to-HN map and feed this to mapdim using the -m flag. Type "mapdim" to see its usage.

bifsplit splits a mapped BIF file into as many sub-files as their are HNs, plus one groups file containing the CN groups and the CN-to-HN map. These files are labelled with the same prefix as the original mapped BIF file and suffixed with the number of the HN they belong to, or with "groups" in the case of the CN groups file. When specifying the BIF file to load at runtime, using buildnet, only the file prefix is specified.

loadnl takes two arguments corresponding to the cube dimension to be used and the number of HN message buffers. It automatically specifies a stack size of 8000, necessary for ANNE's HN code.

7. Files

Several files are important to ANNE's operation. The executable modules for the host and node portions of ANNE are called "anne" and "nl" respectively. Input vectors must be supplied in a file of any name with the number of input vectors at the top of the file. If supervised learning is to take place, a target vector file of any name must be supplied. It does not specify how many vectors it contains. The input, output, and target file names are prompted for at runtime.

Summary statistics are written to "NET.SUMM" at the end of each session with ANNE. This file is overwritten at the start of each session.

If logging was turned on using the "-l" flag for buildnet, then log messages will be written to "ANNE.LOG".

8. Runtime Example

Following is a script containing the steps necessary to simulate a three layer back-propagation network containing 32 CNs in each layer.

```
Script started on Mon Jul 18 16:55:40 1988

Oregon Graduate Center Intel iPSC System 310-40
XENIX 286 R3.4      iSC Release 3.1      (05/87)
 Hypercube Manager
 LAN Name:   hyper

cube[1] ndl
 Usage: NDL <#inputs> <#hidden> <#outputs> <name of output file> [r|f]
cube[2] ndl 32 32 32 n32 f

32 inputs
32 hidden
32 outputs
filename: n32
done

cube[3] mapdim
usage:
mapdim <biffile> <mapfile> <cubedim> [-m <cnhnmap>]|[-l <bl> <ml> <tl>]
cube[4] mapdim n32 n32.d2 2 -l 32 32 32
Calculating pids for net with:
  32 cns in the bottom layer
  32 cns in the middle layer
  32 cns in the top layer
End of bif file found
```

```
cube[5] bifsplit
Not enough arguments
usage: bifsplit [-r] <biffile> <cubedim>
cube[6] bifsplit n32.d2 2
End of bif file found

cube[7] loadconv bp.conv.c ff.conv.c
*make anne "CFC1=bp.conv.c" "CFO1=bp.conv.o" "CFC2=ff.conv.c" "CFO2=ff.con
     cc -Alfu -K -W 2 -c ff.conv.c
ff.conv.c
/usr/include/stdio.h(89) : warning 67: unexpected characters following `#er
     cc -Alfu -K -W 2 -o anne hscan.o annlib.o cli.o hash.o hnet.o printbl
anne completed
cube[8] loaduf bp.uf.c ff.uf.c
/bin/rm nl
* make nl "UFC1=bp.uf.c" "UFO1=bp.uf.o" "UFC2=ff.uf.c" "UFO2=ff.uf.o"  *
     ld -Ml -o nl -m nlmap nl.o nscan.o nnet.o bp.uf.o ff.uf.o noflt.o /li
nl completed

cube[9] loadnl 2 150
load -c 2 -S 8000 -b 150
Number of system buffers: 150, available memory: 250854 bytes
Load succeeded

cube[10] anne
                ** ANNE **  V0.7
             Happy neural networking...

ANNE: buildnet -t
Set timing flag ON
Name of bif file ?   -> nets/n32.d2

Building network  *
Loading BIF sub-files * * * *
Read groups file
Sent group tables  * *
BIF loading took 8 secs

** Network suspended ** (type 'help' for commands)

ANNE: newrun
Which set of user functions to use [1 or 2] ? -> 2
Supervised Learning? [y/n] (y) -> n
Name of input file ?       -> in
Name of output file ?      -> out
Size of cn message packets (default  63)    ->
Timeout for cn message probes (default  0) ->
```

```
ANNE: startnet

           ** AT SYNCH POINT **
** Run: 0       Clock: 1   Checkpoint: -1
** Synch_count: 1    Msg_window: 0   Msgpak size: 63
** Msg timeout: 0    Error: 0.00     Supervised: OFF
Are current synchronization parameters ok? [y/n/q] (y) n

Enter synch_count (now 1) -> 10

Enter msg_window (now 0)  -> 5

synch_count = 10 msg_window = 5 ok? [y/n/q] (y)

Is the current checkpoint ok? (now -1) [y/n/q] (y)
Checkpoint (-1) expired, set new checkpoint -> 11

** New network cycle ** Vector #1 **

OUTPUT: 0.64 0.65 0.80 0.28 0.67 0.64 0.61 0.50 0.61 0.54 0.49 0.70 0.72 0.
           ** AT SYNCH POINT **
** Run: 1       Clock: 1   Checkpoint: 11
** Synch_count: 10   Msg_window: 5   Msgpak size: 63
** Msg timeout: 0    Error: 0.00     Supervised: OFF
** Reached checkpoint **
** Network CONVERGED **
Printing output vector to out

** Network suspended ** (type 'help' for commands)

ANNE: help
         /*** current command syntax Jan. 18, 1988 ***/
/*** '#' in front of command signifies it is unimplemented ***/

/** global simulation commands **/
buildnet  [-tv]
newrun
show
startnet
stopnet
savenet
quit

/** CN map commands **/
print    <struct_name> <field_name> <CNmap_name>
assign        <struct_name> <field_name> <CNmap_name>
trace      <struct_name> <field_name> <CNmap_name>
#reset     [<CNmap_name>|all]
```

```
#hammdist   <CNmap_name> <CNmap_name>
#intersect  <CNmap_name> <CNmap_name>
#union      <CNmap_name> <CNmap_name>
#makemap    <CN_range> <CNmap_name>

ANNE: quit
MPS = 5175.72   WPS = 469.19    XPS = 4564.95  total msgs =   22592
HXPS = 1107.00  Percentage of msgs delivered = 97.26%

Leaving ANNE
cube[11]
script done on Mon Jul 18 17:07:28 1988
```

9. BIF Grammar

BIF's (pseudo) grammar and structure is presented in three parts in order to clarify its structure. First, there are some notes on BIF syntax. This section is followed by BIF's BNF description, which reduces to the level of *field-terminals.* These terminals correspond to the field names in BIF's current specification. Last, there is a field-by-field BIF specification, showing how CNs, sites, and links are layed out, along with definitions of the named fields. This description also includes the CN types fields.

9.1. BIF Syntax

Six reserved words plus four bit flags are used to facilitate BIF parsing. There is one bit vector per CN, site, and link. Three reserved words are used to delimit the beginning and end of the CN groups block and each group within this block. **sgbk**, **egbk**, and **egrp** mean "start CN group block", "end CN group block", and "end CN group", respectively. In a similar manner, the other three reserved words mark the start and end of the CN record section (**scbk** for "start CN block", and **ecbk** for "end CN block") with a word to end each CN record (**endc**) In each site and link specification there is a bit that tells the parser when it is reading the last site or link in the current sub-block. Two additional bits indicate which of certain optional fields are present in CN and link records.

Special conditions and limitations for this BIF that should be noted:

CN group names are a maximum of 8 characters in length.

"Optional" fields are, for the time being, non-optional. Current BIF parsers ignore the bits flagging optional fields. Thus, all optional fields exist in every BIF file.

"User model" fields are those fields necessary for the user when manipulating the network data in the simulators, but not necessary to the input BIF file used by our simulators. Thus, they are left out of the BIF files to reduce the file length.

Reserved words which end a sub-block (**egrp**,**endc**) must precede the reserved word for ending the block.

9.2. BIF's BNF

The different fonts used in the grammar have these meanings:

non-terminal, field-terminal, **reserved word**, real terminal, **constant**.

Items on the right side of a production rule are disjoint if they are on separate lines, except where they must extend over more than one line. In that case an '@' symbol is used to denote continuation.

bif-file:
 gblock cblock

gblock:
 sgbk *grouplist* **egbk**

grouplist:
 grouplist
 groupfields **egrp**

groupfields:
 index name initpot initstate

cblock:
 scbk *cnlist* **ecbk**

cnlist:
 cnlist
 cn **endc**

cn:
 cnfields sites

cnfields:
 group index procid delay bitvec *cn-options cn-data*

cn-options: (assume all are present for now, see cover note)
 history restpot pot state output error sd

cn-data: (not included in bif file, but included in user's data model)
 nsites

sites: (up to last *bit set in* iotype*)*
 sites
 sitefields links

sitefields:
 value sitevec *site-data*

site-data: (not in bif file, but in user's data model)
 nlinks

links: (up to last *bit set in* lnkvec*)*
 links
 linkfields

linkfields:
 lnkvec *link-options* cn site link weight *link-data*

link-options:
 history

link-data: (not in bif file, but in user's data model)
 inval

9.3. BIF Field Specifications

Field-Terminals Definitions			
field name	def	field name	def
bitvec	nn-byte	name(type)	id8
cn	nn-int	nlinks	nn-short
delay	nn-short	nsites	nn-short
error	short	output	short
history(CN)	nn-byte	pot	short
history(link)	nn-byte	procid	nn-short
group	nn-byte	restpot	short
index(CN)	nn-int	sd	short
index(type)	nn-byte	site	nn-short
initpot	short	sitevec	nn-byte
initstate	nn-byte	state	nn-byte
inval	short	value	short
link	nn-short	weight	short
lnkvec	nn-byte		

nn-byte: 8 bits unsigned (non-negative value)
short: 16 bits signed
nn-short: 16 bits unsigned
nn-int: 32 bits unsigned
id8: 8 byte character string (9 bytes including '\0')

bitvec

7	6	5	4	3	2	1	0
4 bits				1 bit	1 bit	1 bit	1 bit
no. of udf's				*not used*	*history* *restpot* *pot*	*state* *output*	*error* *sd*

sitevec

7	6	5	4	3	2	1	0
1 bit	1 bit	1 bit	6 bits				
i/o	*last*	*pbh/ptp*	*no. udf's*				

lnkvec

7	6	5	4	3	2	1	0
1 bit	1 bit	1 bit	1 bit	4 bits			
history	*last*	*active*	*wtup*	*not used*			

bitvec *(4 low-order bits)*		
decimal	*octal*	*optional fields present*
0	00	no optional fields present
1	01	error, sd
2	02	state, output
3	03	state, output, error, sd
4	04	history, restpot, pot
5	05	history, restpot, pot, error, sd
6	06	history, restpot, pot, state, output
7	07	all optional fields present

sitevec *(4 high-order bits significant)*		
decimal	*octal*	*meaning*
0	0000	output site, not last site, ptp
32	0040	output site, not last site, pbh
64	0100	output site, last site, ptp
96	0140	output site, last site, pbh
128	0200	input site, not last site, ptp
160	0240	input site, not last site, pbh
192	0300	input site, last site, ptp
224	0340	input site, last site, pbh

lnkvec *(3 high-order bits)*		
decimal	*octal*	*meaning*
0	0000	history off, not last link, inactive link, no weight update
16	0020	history off, not last link, inactive link, weight update
32	0040	history off, not last link, active link, no weight update
48	0060	history off, not last link, active link, weight update
64	0100	history off, last link, inactive link, no weight update
80	0120	history off, last link, inactive link, weight update
96	0140	history off, last link, active link, no weight update
112	0160	history off, last link, active link, weight update
128	0200	history on, not last link, inactive link, no weight update
144	0220	history on, not last link, inactive link, weight update
160	0240	history on, not last link, active link, no weight update
176	0260	history on, not last link, active link, weight update
192	0300	history on, last link, inactive link, no weight update
208	0320	history on, last link, inactive link, weight update
224	0340	history on, last link, active link, no weight update
240	0360	history on, last link, active link, weight update

Field definitions (w/o user-defined fields)

CN (* marks optional fields, † marks user's fields)		# of bytes
group	pointer to a CN group	1
index	unique # in network	4
procid	physical processor id	2
delay	delay to add to output	2
bitvec	optional field flags	1
history*	avg. of recent change	1
restpot*	resting potential	2
pot*	potential	2
state*	resting, firing, ...	1
output*	output of CN	2
error*	error in output	2
sd*	statistical deviation	2
nsites†	number of sites	2
	total	12-24
sitelist	*sites repeat within a CN*	
value	value passed to CN	2
sitevec	input/output, last flag	1
nlinks†	number of links	2
	total	5
linklist	*links repeat within a site*	
lnkvec	history flag, last flag, active flag, weight update flag	1
history*	avg. of recent change	1
cn	CN this connects to	4
site	site this connects to	2
link	link this connects to	2
weight	current weight	2
inval†	input on this link	2
	total	13-14

Field definitions (cont.)

CNgroup		# of bytes
index	unique index for this group	1
name	name shared by CN's	8
initpot	initial potential	2
initstate	initial state	2

References

[Bah88] Bahr, C., "ANNE: Another Neural Network Simulator," Tech. Report CS/E-88-028, Dept. of Computer Science/Engineering, Oregon Graduate Center, Beaverton, OR, August 1988.

[BHJ88] Bahr, C., Hammerstrom, D. and Jagla, K., *Concurrent Neural Network Simulation: Two Examples Within A Single, Integrated Neural Network Hardware Development Environment,* IASTED Applied Simulation and Modelling Conference, Galveston, Texas, May 1988.

[Fan86] Fanty, M., *A Connectionist Simulator for the BBN Butterfly Multiprocessor,* Department of Computer Science, Univ. of Rochester, NY, January 1986.

[Jag88] Jagla, K., "A Broadcast Hierarchy Simulator for the Intel iPSC," CSE Technical Report, Oregon Graduate Center, Department of Computer Science/Engineering, Beaverton, OR, March 1988. In preparation.

[JeS82] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation using the Time Warp Mechanism, Part I: Local Control," Tech. Rep.-83-204, Univ. of Southern California, Computer Science Dept., December 1982.

[Joh88a] Johnson, M. A., "NDL User's Manual," CSE Technical Report, Oregon Graduate Center, Department of Computer Science/Engineering, Beaverton, OR, July 1988. In preparation.

[Joh88b] Johnson, M. A., "NDL Reference Manual," CSE Technical Report, Oregon Graduate Center, Department of Computer Science/Engineering, Beaverton, OR, July 1988. In preparation.

[May88] May, N., "Fault Simulation of a Wafer-Scale Neural Network," Tech. Report CS/E-88-020, Dept. of Computer Science/Engineering, Oregon Graduate Center, Beaverton, Oregon, May 1988.

[MRH86] McClelland, J. L., Rumelhart, D. E. and Hinton, G. E., "Distributed Representations," in *Parallel Distributed Processing,* vol. 1, P. R. Group (ed.), 1986.

[Mis86] Misra, J., "Distributed Discrete-Event Simulation," *Computing Surveys,* vol. 18, 1 (March 1986), .

[SSB83] Small, S. L., Shastri, L., Brucks, M. L., Kaufman, S. G., Cottrell, G. W. and Addanki, S., "ISCON: A Network Construction Aid and Simulator for Connectionist Models," Tech. Rep. 109, Univ. of Rochester, Dept. of Computer Science, April 1983.