HOSTING THE NADEX ENVIRONMENT ON THE UNIX OPERATING SYSTEM

by

DENIS EVERETT EATON

B.S., Kansas Wesleyan, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

Approved by:

Vic Wallentine
Major Professor

TABLE OF CONTENTS

ILLUSTRATIONS

ACKNOWLEDGEMENTS

This research was supported in part by the Army Institute for Research in Management, Information, and Computer Systems under grant number DAAG 29-78-G-0200 from the Army Research Office.

Special thanks go to Dr. Virgil Wallentine, Quincy, and the OS Group for their contributions to this work.

CHAPTER ONE

Introduction

Command language facilities for the construction and execution of software configurations (networks of communicating processes) are very limited today because current operating systems do not support this level of complexity. The Network ADaptable EXecutive (NADEX) [10,11,12] was designed to support dynamic configurations (those configurations which are constructed at command interpretation time) of cooperating processes. These dynamic configurations include arbitrary graphs which may contain cycles. NADEX runs on top of the NADEX core operating system. The objective of this work is to adapt the NADEX environment so it will run with the UNIX* [5,6] operating system as its' core operating system.

In chapter two software configurations are described along with the capability of the UNIX and NADEX core operating systems to handle them. Chapter three describes the implementation of NADEX using the UNIX operating system as its' core operating system. Chapter four contains a

---

* UNIX is a trademark of Bell Laboratories.

description of the procedure that brings up software configurations in the UNIX implemetation. Future work to be done on the UNIX implementation of NADEX is given in chapter five.

CHAPTER TWO

Software Configurations

2.1 Concept of Software Configurations

The idea of software configurations follows the idea of modular programming. Each node (module) in the configuration can be developed independently and all they must know about the other nodes is the internode communication protocol. They do this interprocess communication over some standard communication mechanism.

2.2 Software Configuration With the UNIX Operating System

The UNIX shell [1,5,6] supports limited software configurations called pipelines (Figure 1). A pipeline is where processes are strung together with a process' standard output connected to the next process' standard input to form a chain of processes. These processes do not know from where their input comes or to where their output goes. It could be a file, a physical device, or another process.

Processes running on top of UNIX can create software configurations. This is done by the process getting pipes from UNIX and by making copies of itself with the FORK system call. These different copies can communicate over

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

FIGURE 1

EXAMPLE OF A PIPELINE SOFTWARE CONFIGURATION

the pipes. Also any of these copies can bring up another process in its place and pass the pipes to it.

2.3.1 NADEX

NADEX is a software environment whose objective is to support modular programming. The concept of "programming in the small" which has been so successful in UNIX is extended under NADEX to support general graphs of communicating nodes. These general graphs are called software configurations and consist of nodes which communicate via Data Transfer Streams (DTS's). These DTS's are full-duplex in nature, and therefore, support bi-directional communication between any two nodes which they connect. Nodes access DTSs via ports. These ports are distribution-independent and, therefore, permit nodes of a configuration to be distributed across a computer network without reprogramming.

Figure 2 shows the structure of NADEX. Level one is the level that provides the NADEX environment to the user, while levels two through four are the NADEX core operating system.

NADEX is written in Concurrent PASCAL [2] with the idea of it being easily moved to different machines. It is designed to run on top of the host machine's operation system or on the bare machine.

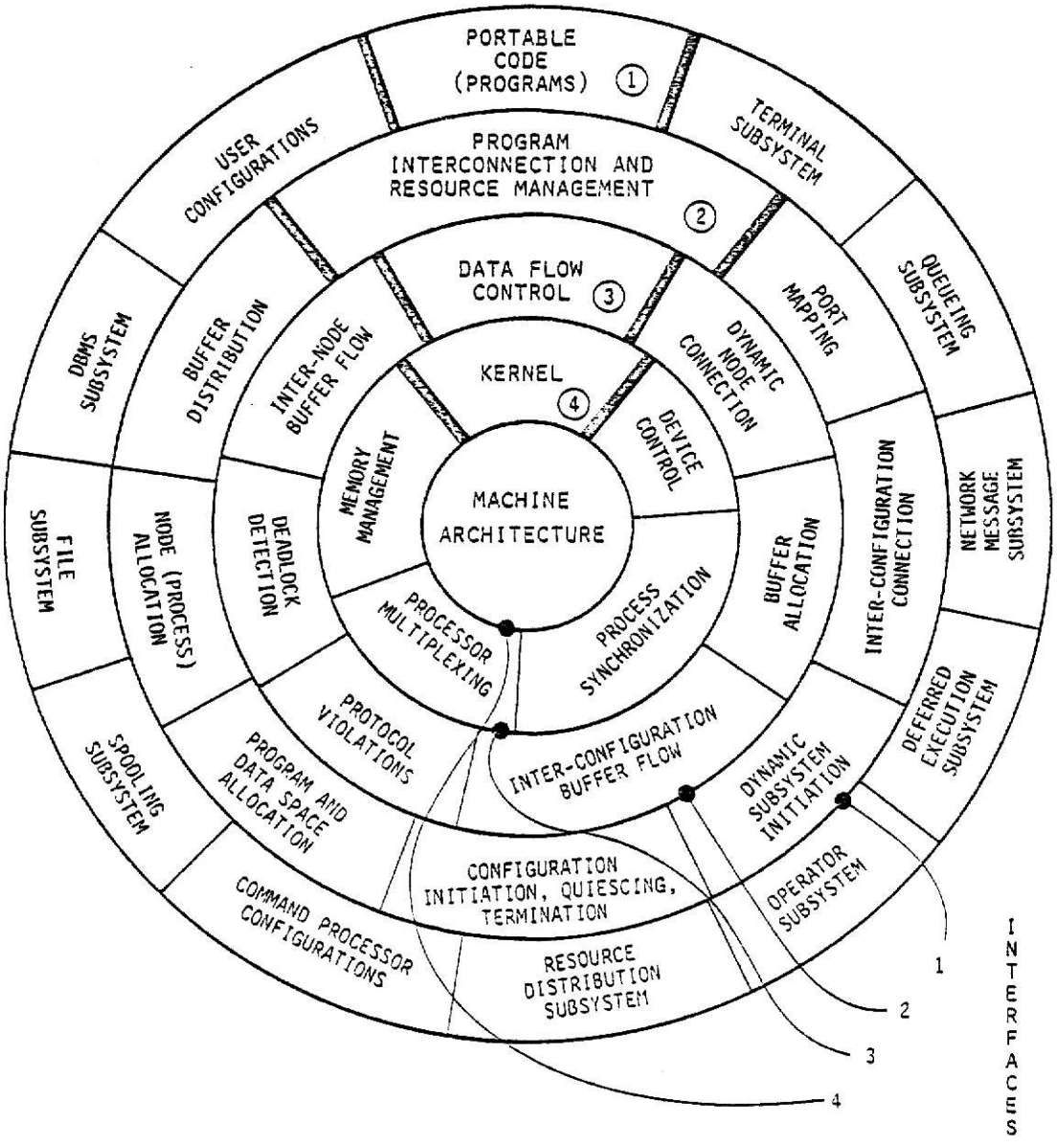

FIGURE 2
NADEX STRUCTURE

2.3.2 Properties of NADEX Software Configurations

A configuration consists of a collection of nodes connected by DTS's. Nodes can be user programs (both sequential and concurrent languages such as SPASCAL and CPASCAL), file access nodes (for accessing files within the NADEX file system), I/O device access nodes (for accessing I/O devices not supported by the NADEX file system), or external configurations such as subsystems.

Nodes within the configuration are connected by DTS's which are also called connections. Each connection consists of two bi-directional components--data and parameter. The data component transfers data in page-sized blocks (a page is 512 bytes) and interfaces to the user program at the page, logical record, or character level. The parameter component transfers small parameter blocks typically used for control information. The data and parameter components are totally independent. The two directions of each component are independent in the sense that each direction has its own queue, but the user protocol restrictions are defined in terms of the bi-directional components.

For the purposes of these discussions, we will speak of a node issuing reads and writes to a port which is local to the node. The connection of these ports forms the DTS's. Figure 3 contains the PREFIX which implements these operations. These should be assumed to be read-page and write-page requests for the data component, and read-parm

Figure 3

NADEX Native Prefix

```
"************************************************************
*                                                          *
*                    NADEX NATIVE PREFIX                   *
*                                                          *
************************************************************"

  ; CONST PAGE_SIZE = 512          "SIZE OF DATA PAGE"
  ; PARM_SIZE = 32                 "SIZE OF PARAMETER BLOCKS"
  ; MAX_DTS = 40                   "MAX GLOBAL DTS ID"
  ; MAX_PORT = 20                  "MAX PORT ID"
  ; MAX_PARM = 10                  "MAX PARM ID"
  ; SVC1_BLOCK_SIZE = 24           "SIZE OF SVC 1 PARM BLOCK"
  ; SVC7_BLOCK_SIZE = 28           "SIZE OF SVC 7 PARM BLOCK"
  ; SD = 700                       "PREFIX STACK DEPTH"

; CONST NL = '(:10:)'
  ; CR = '(:13:)'
  ; ETB = '(:23:)'
  ; EM = '(:25:)'
  ; BEL = '(:07:)'

; TYPE PAGE = ARRAY [ 1 .. PAGE_SIZE ] OF BYTE
  ; PARAMETER = ARRAY [ 1 .. PARM_SIZE ] OF BYTE
  ; UNIV_SVC1_BLOCK = ARRAY [ 1 .. SVC1_BLOCK_SIZE ] OF BYTE
  ; UNIV_SVC7_BLOCK = ARRAY [ 1 .. SVC7_BLOCK_SIZE ] OF BYTE

; TYPE DTS_INDX = 1 .. MAX_DTS
  ; DTS_INDX0 = 0 .. MAX_DTS
  ; PORT_INDX = 1 .. MAX_PORT
  ; PORT_INDX0 = 0 .. MAX_PORT
  ; PARM_INDX = 1 .. MAX_PARM
  ; PARM_INDX0 = 0 .. MAX_PARM

; TYPE DTS_SET = SET OF DTS_INDX

; TYPE BUF_TYPES = ( PARM_BUF , DATA_BUF , NIL_BUF )
                                              "BUFFER TYPES"

; TYPE PREFIX_TYPES = ( NATIVE_PREFIX "0"
                      , PASDRIVR_PREFIX "1" )
```

```
; TYPE REQ_CODES
    = ( REQ_OK "0" , REQ_NODE_ABORT "1" , REQ_DTS_ABORT "2"
      , REQ_DEFER "3" , REQ_UNRES_DTS "4"
      , REQ_PROT_ERROR "5" , REQ_BAD_PORT "6" )
      "PREFIX DTS OPERATION RETURN CODES"

; PROCEDURE READ_CHAR ( PORT : PORT_INDX ; VAR C : CHAR )

; PROCEDURE WRITE_CHAR ( PORT : PORT_INDX ; C : CHAR )

; PROCEDURE READ_DATA
    ( PORT : PORT_INDX
    ; VAR DATA : UNIV PAGE
    ; VAR LENGTH : INTEGER
    ; VAR RESULT : REQ_CODES
    )

; PROCEDURE WRITE_DATA
    ( PORT : PORT_INDX
    ; DATA : UNIV PAGE
    ; LENGTH : INTEGER
    ; CONDITIONAL : BOOLEAN
    ; VAR RESULT : REQ_CODES
    )

; PROCEDURE READ_PARM
    ( PORT : PORT_INDX
    ; VAR PARM : UNIV PARAMETER
    ; VAR RESULT : REQ_CODES
    )

; PROCEDURE WRITE_PARM
    ( PORT : PORT_INDX
    ; PARM : UNIV PARAMETER
    ; CONDITIONAL : BOOLEAN
    ; VAR RESULT : REQ_CODES
    )

; PROCEDURE MAP_PORT
    ( PORT : PORT_INDX
    ; BUF_TYPE : BUF_TYPES
    ; VAR RDTS : DTS_INDX0
    ; VAR WDTS : DTS_INDX0
    )

; PROCEDURE AWAIT_EVENTS
    ( VAR READ_WAITS , WRITE_WAITS : DTS_SET
    ; VAR READ_READY , WRITE_READY : DTS_SET
    ; VAR RESULT : REQ_CODES
    )
```

```
; PROCEDURE DISCONNECT
    ( PORT : PORT_INDX
    ; VAR RESULT : REQ_CODES
    )

; PROCEDURE FETCH_USER_ATTRIBUTES

; PROCEDURE SUBMIT_CONFIG

; PROCEDURE SVC1 ( VAR PARM : UNIV UNIV_SVC1_BLOCK )

; PROCEDURE SVC7 ( VAR PARM : UNIV UNIV_SVC7_BLOCK )

; PROCEDURE FETCH_PARM
    ( PARM_ID : PARM_INDX
    ; VAR PARM : UNIV PARAMETER
    ; VAR OK : BOOLEAN
    )

; PROCEDURE LOAD_OVERLAY
    ( PORT_ID : PORT_INDX
    ; VAR OK : BOOLEAN
    )

; PROCEDURE INVOKE_OVERLAY
    ( VAR ARG : INTEGER
    ; PREFIX_TYPE : PREFIX_TYPES
    ; VAR RESULT , LINENO : INTEGER
    )

; PROCEDURE CANCEL_NODE

; PROCEDURE CANCEL_CONFIG

; PROCEDURE BREAKPNT ( LN : INTEGER )
```

and write-parm requests for the parameter component. The blocking of character and logical record data into pages is handled by the prefix of the nodes and will not be discussed here. Unless otherwise specified all discussions apply equally to data and parameters, and no distinction will be made.

There are no structural restrictions on the graph formed by the nodes and connections (DTS's). In particular, it need not be linear (like *SOLO* [2] and UNIX) or hierarchical. It need not even be acyclic as in AMPS [4] or connected. Nodes are not precluded from having connections to themselves. Thus, the configuration is described by a (labeled) undirected graph. The labeling occurs where each connection enters the two (not necessarily distinct) nodes it connects.

The user programs (as well as the various system routines which implement the other nodes) address the connections emanating from each node by DTS ids local to the node. These local DTS ids are also called port numbers. The meaning of the data stream associated with each port is defined by the program. Port numbers are generally assigned by the programmer starting with one (since the system will place a configuration-dependent upper limit on the port numbers for economy in table storage). These port numbers are the labels on the configuration graph.

The structure of a configuration is defined by a

language which builds a file called a Partial Configuration Descriptor (PCD) [7,8]. Figure 4 shows the programs that handle PCD and their relationships. The PCD defines the structure of the configuration and the type (user program, file access, etc.) of each node. PCD can be built hierarchically in that a PCD can be included as part of another PCD in a hierarchical way. When the user requests that a configuration be run (either through a terminal command or a command in a batch job), the PCD's are used along with information from the command to construct a Configuration Descriptor (CD). The CD includes all of the information about the configuration including, for example, the names of the files to be accessed by the file access nodes. The CD contains enough information for the system to allocate resources and run the configuration.

Examples of configuration description languages are given in reference [3,8]. There are statements which define each node and the function it is to perform as well as those which define each connection. The node definitions may completely specify the function, or some information (such as filenames) may be left to be filled in from the command. The connection definition may include buffer allocation parameters (to be discussed later). The program which converts the PCD into a CD is part of the command processor configuration and runs as a separate configuration.

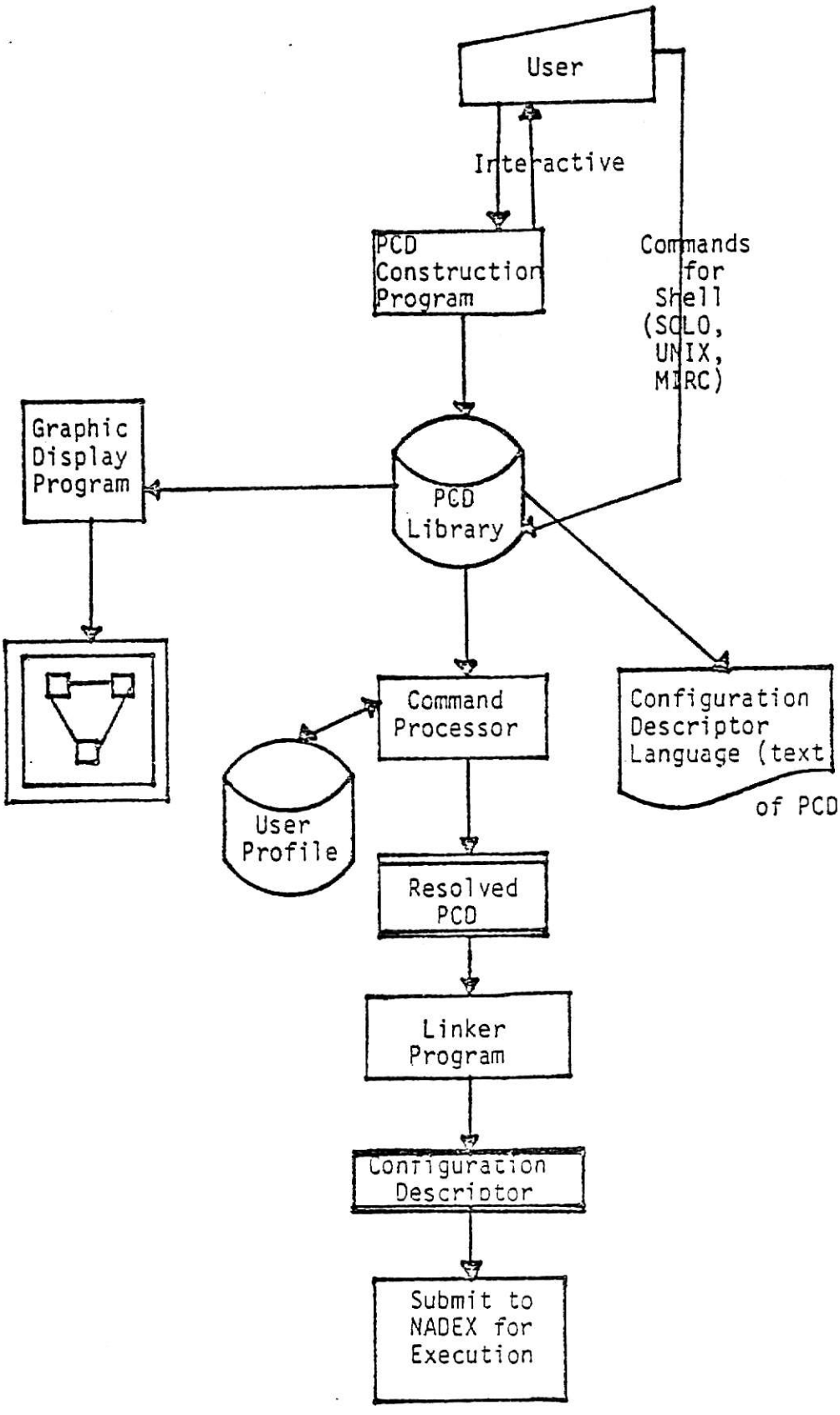

Figure 4

THE PCD WORKBENCH

2.3.3 NADEX Subsystems

In addition to running isolated user configurations, NADEX allows user configurations to communicate with special configurations known as subsystems. As mentioned, subsystems are themselves configurations, but also have an interface to allow connection to nodes of user configurations. Typical uses of subsystems would be a data base management system, a file system, and an interconfiguration communications system (for example a network Interprocess Communication System).

A subsystem is unique in that it is not activated directly by a user command or a user batch job. Instead, it is activated whenever a configuration is started which requires the services of that subsystem. The subsystem then continues to run until there are no more active users (configurations). Then, depending on the subsystem, it may be automatically terminated or it may remain active waiting for additional users.

A subsystem serves multiple configurations concurrently and has much of the responsibility for multiplexing itself among its users. Furthermore, as user configurations are initiated and terminated, they can dynamically connect to and disconnect from the subsystem.

When a user describes a configuration, access to a subsystem is shown as a single node with connections from other nodes in the configuration. However, when the

configuration is implemented, the system will not create such a node. Instead, the connections to the subsystem node in the user description will be connections between user nodes and the user interface of the subsystem.

The user nodes communicate with the subsystem just as if it were another node. The normal DTS operations (read and write parameter and data) are used to implement a protocol defined by the particular subsystem. The user nodes themselves are not aware that they are communicating with a subsystem rather than with another node (which uses the same protocol, of course).

From the subsystem's point of view, there is a set of connections defined in the subsystem's configuration description called the user interface connections. These have one end which terminates within the subsystem (perhaps on a one-to-many basis) and the other end is initially left free. When user configuration is started, its connections to the subsystem will be implemented over some of these user interface connections.

The subsystem again uses the normal DTS operations to communicate with the various users which it serves. Since subsystems serve multiple users, they will typically be users of the multiple-condition wait, which waits for requests coming from the various users.

Note that neither the subsystem nor the user configuration is aware (at this point) that the subsystem is

actually a subsystem. In fact, without changing any of the programming discussed so far, the collection of nodes which constitute a subsystem could be taken and placed in the user configuration. The connections from user nodes to the subsystem would be the user interface connections, and the two parts of the configuration would otherwise be independent. This allows a user to use a private copy of the subsystem within a user configuration if necessary. This is the recommended procedure for debugging subsystems. Note that no changes in any of the programming is required to move between these two modes.

2.3.4 MIRACLE Command Processor Configuration

The MIRACLE command processor configuration (Figure 5) is a software configuration that runs on top of NADEX. Using the MIRACLE command processor [3] a user can interactively describe a software configuration. MIRACLE builds a completely resolved PCD which is a complete description of the configuration to be run. This PCD is given to LINK [8] which makes a configuration descriptor from it. This configuration descriptor is given by LINK to the NADEX core operating system which brings up the configuration and runs it (Figure 4).

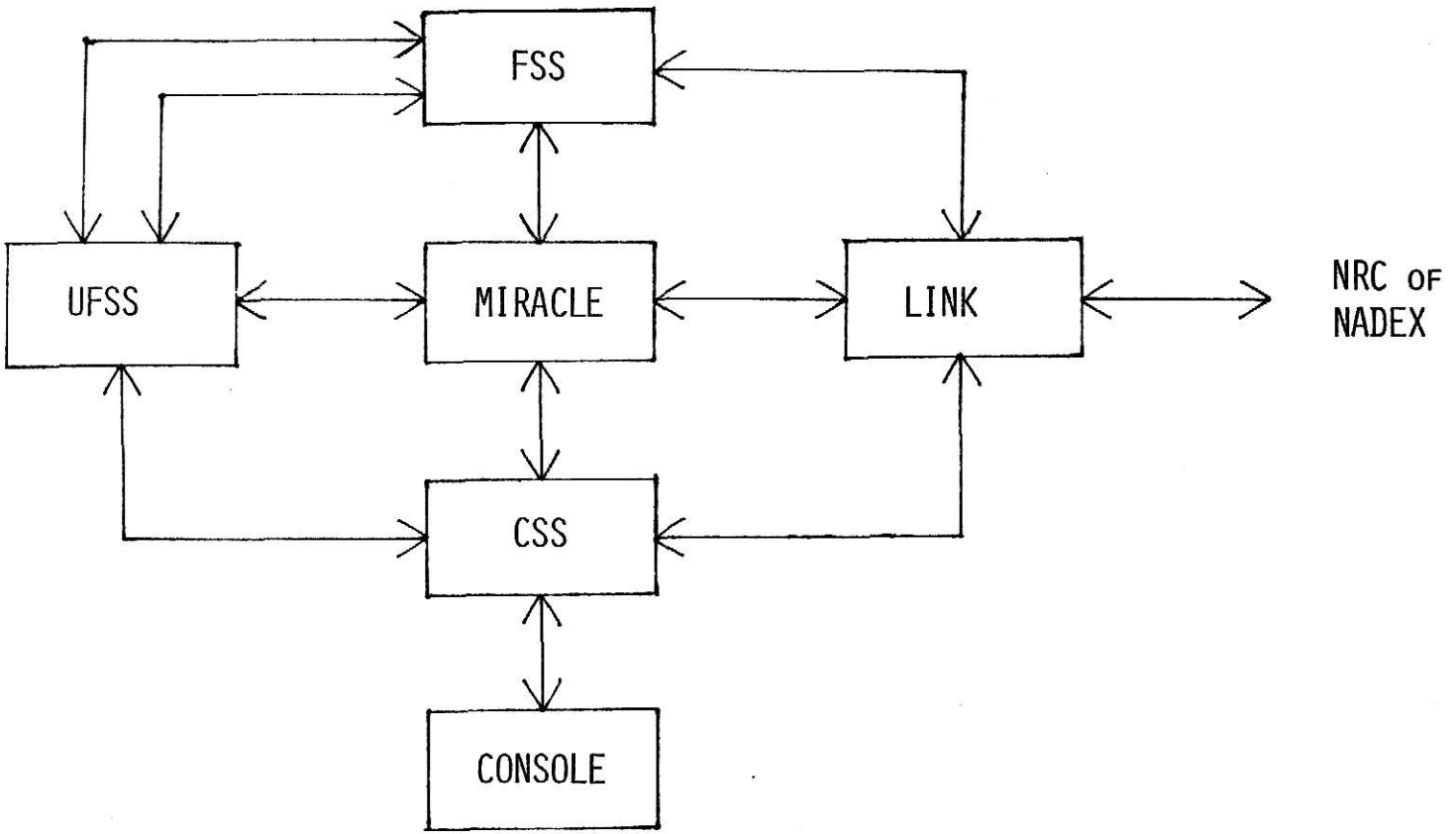

FIGURE 5

MIRACLE COMMAND PROCESSOR CONFIGURATION

CHAPTER THREE

Hosting the NADEX Environment On Top of the UNIX Operating System

3.1 Introduction

In this chapter the interface number one between levels one and two in figure 2 is described. This new interface allows levels two through four, which is the NADEX Core Operating System, to be replaced by the UNIX Operating System (Figure 6).

3.2 MIRACLE Command Processor Configuration

In implementing the MIRACLE command processor configuration under UNIX it was desirable that as little modifications as possible be made to the software and that the implementation be consistent with the rest of the NADEX network. The procedures of MIRACLE, LINK, and UFSS [9] did not have to be changed. Although new versions of the subsystems had to be made available to run with UNIX. Since the network resource controller (NRC) subsystem of NADEX is not available with the UNIX implementation to have configurations brought up, a procedure UNIXSC (UNIX Software

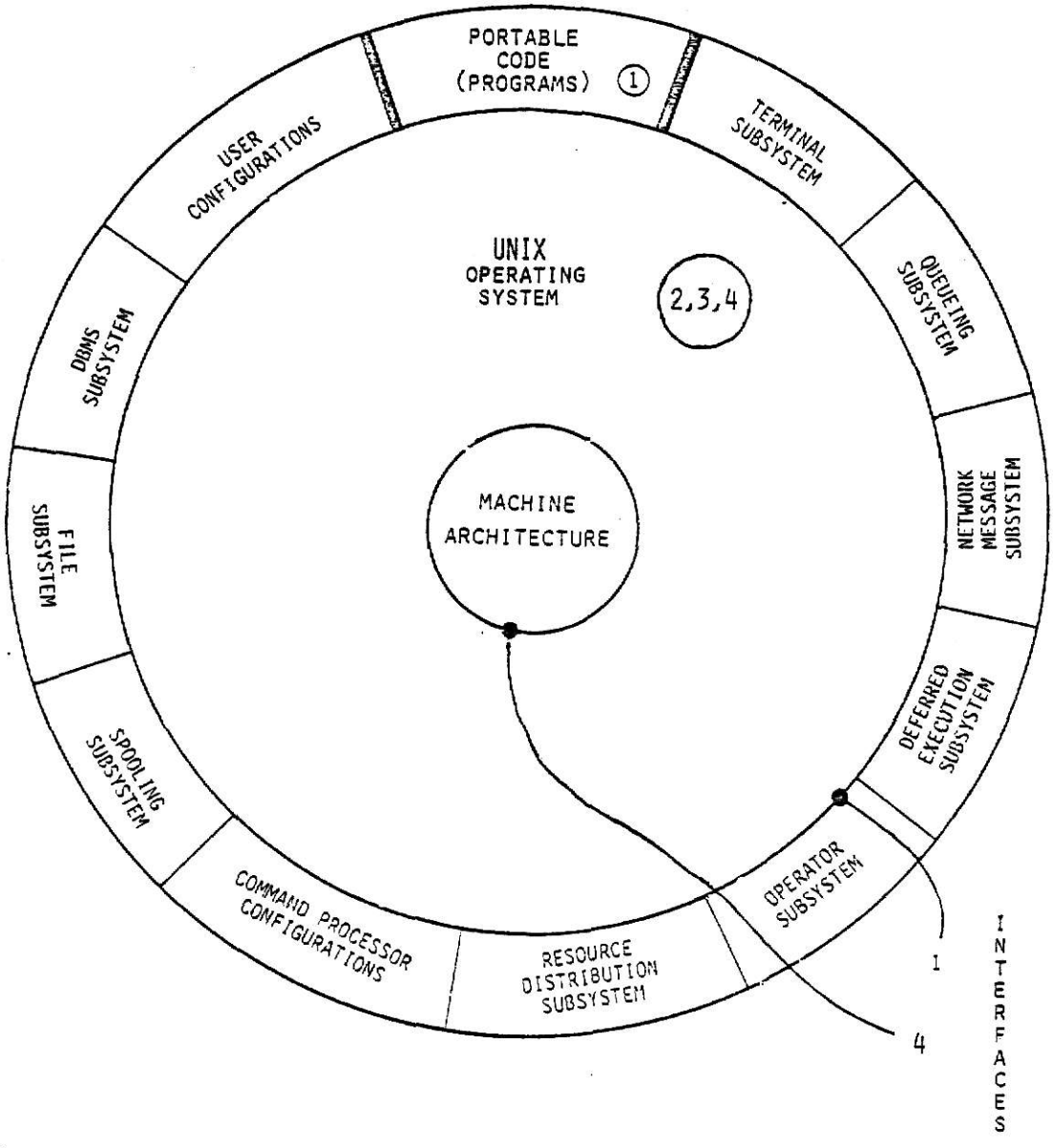

FIGURE 6
NADEX STRUCTURE
WITH UNIX CORE

Configurations) was writen to bring up the Software Configurations. UNIXSC is connected to LINK in the NRC's place (Figure 7).

3.3 Mapping NADEX Ports Onto UNIX Pipes

Under NADEX nodes communicate over ports. The ports of two nodes are connected by means of NADEX Connections (CONN) (figure 8-a). These connections consist of two Data Transfer Streams (DTS), one for each direction (figure 8-b). Each of these data transfer stream can handle two types of separate transmissions, one called data buffers (up to 512 bytes) and the other called parameter buffers (32 bytes). The DTS keeps these two types of buffers separate (figure 8-c). Thus each connection can be considered as four separate transmission streams, two in each direction. The mapping of these connections onto pipes is simply having one pipe for each of these four separate transission streams (figure 8-c). If either data buffers or parameter buffers are not used then the pair of pipes for the one not used are not needed.

On data buffers the length of the buffer is put on the pipe before the buffer is. This is so that when the buffer is taken off of the pipe the buffer length is known which allows the right number of bytes can be taken off the pipe.

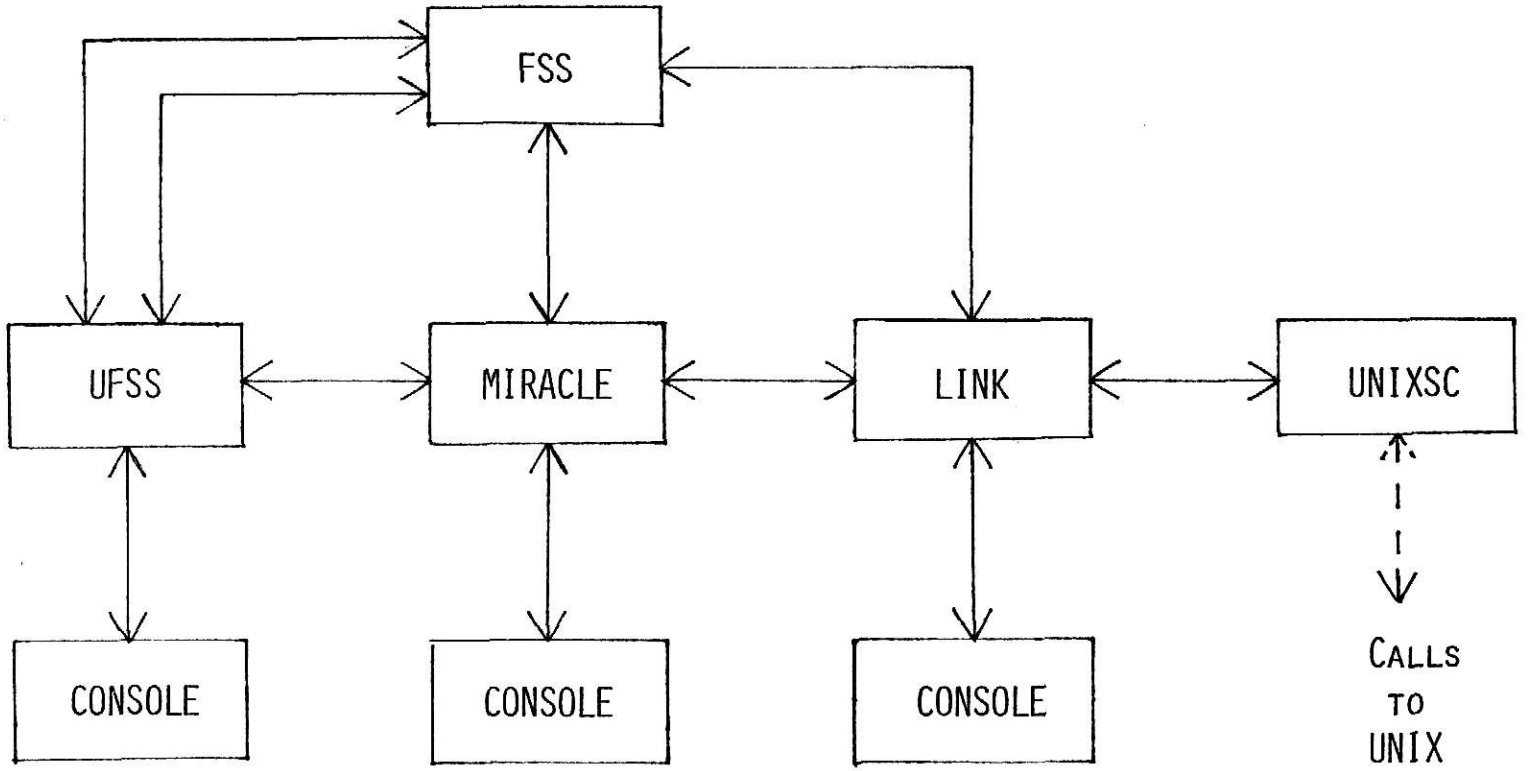

FIGURE 7

UNIX VERSION OF THE MIRACLE
COMMAND PROCESSOR CONFIGURATION

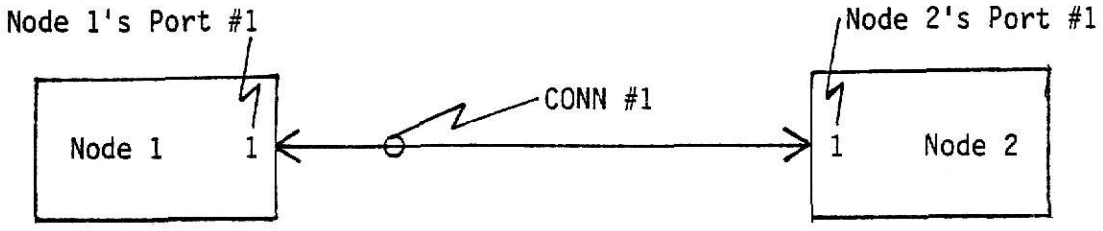

Figure 8-A

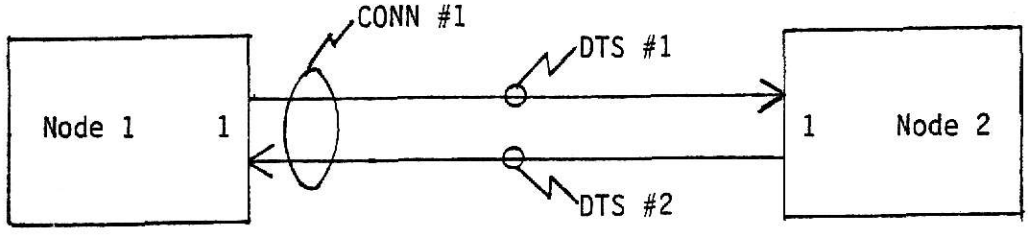

Figure 8-B

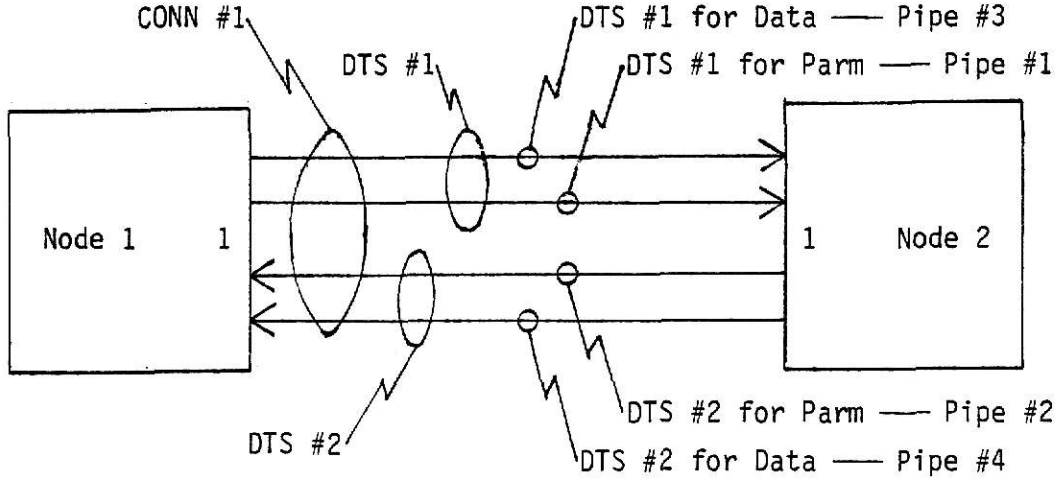

Figure 8-C

FIGURE 8

MAPPING OF PORT
CONNECTIONS ONTO PIPES

3.4 NADEX Prefix for the UNIX implemetation

The UNIX implementation of NADEX must have its own prefix which is similar to the NADEX Native Prefix (Figure 2). When a node is brought up it is passed a parameter which contains the node's parameters, the node's attributes, and the table for mapping ports onto pipes (Figure 9).

The read and write calls appear the same to the user except not all of the REQ_CODES can be implemented. The read and write calls use the port to pipe mapping described in section 3.2. If the CONN_NUM in the node's port to pipe mapping table is not zero then this port is connected to the file subsystem and the port to pipe mapping is done in the way described in section 3.4.3. DISCONNECT just removes all the pipe ids in the NODE_PIPE_TABLE for this port so that this node can not use it again. FETCH_USER_ATTRIBUTES returns the nodes attributes which were pasted to it in its parameter. The CANCEL_NODE and the BREAKPTN calls causes the node to terminate. The rest of the prefix calls are not implemented.

3.5.1 Subsystems

The subsystems in the NADEX implementation are dynamically connected to the configurations as they are activated. When the configuration is finished it goes down

```
; TYPE UNIX_PIPE_TABLE_ENTRY
    = RECORD
        PIPE_PARM_READ
        , PIPE_PARM_WRITE
        , PIPE_DATA_READ
        , PIPE_DATA_WRITE
        : INTEGER
      ; CONN_NUM : CD_CONN_INDX0 (* IS ZERO UNLESS ONE END
                                    OF THE CONN IS TO THE
                                    FSS  *)
      ; PORT_PARM : PARAMETER
      END

; TYPE NODE_ATTRIBUTES
    = RECORD
        USER_ID : CHAR8
      ; NODE_ID : PL_NODE_INDX
      ; TERM_ID : INTEGER
      END

; TYPE NODE_PARM_TYPE
    = RECORD
        PIPES : ARRAY [ CD_PORT_INDX ]
          OF UNIX_PIPE_TABLE_ENTRY
      ; ATTRIBUTES : NODE_ATTRIBUTES
      ; PARAMETERS : ARRAY [ NODE_PARM_INDX ] OF CHAR
      END
```

Figure 9

Data Type of the Parameter to a Node.

but the subsystem stays up and is dynamically connected to whatever comes up next. In the UNIX implementation using pipes there is no dynamic connect, because the pipes must be known ahead of time by the users of it. Thus subsystems must be implemented differently. This is done by having the subsystems brought up with each of the configuration that uses them.

3.5.2 Console Subsystem

The Console SubSystem (CSS) [9] is not needed in the UNIX implementation because UNIX does the multiplexing of the console itself. So each connection that was connected to the console subsystem is directly connected to the console node.

3.5.3 File Subsystems

The command processor configuration as well as most other configurations need the File SubSystem (FSS). The file subsystem is system dependent because it makes calls to the host operating system for file operations. To make software easily transportable to the UNIX implemetation the protocol to the file subsystem was not changed.

There are three ways to implement the file subsystem.

(1) They can be accessed directly by the program with

the normal UNIX system calls.

(2) They can be accessed as they are in NADEX by having a file subsystem that does the actual access operations. All requests and responses to the subsystems are done over ports in the usual way.

(3) They can be done as in (2) as far as the node knows. But instead of having a separate file subsystem to do the accesses, the operations sent over the ports are interpreted by the prefix and the prefix does the accesses by doing system calls to UNIX. Thus the subsystem is included in the prefix.

In this adaptation the file subsystem is done in the way described in (2) above.

The file subsystem only has one incoming pipe and all nodes that are connected to the file subsystem do their sending over this one pipe. For each node connected to the file subsystem over a port there are two pipes for the file subsystem to write to and the other node to read from. The mapping of the port onto these two pipes is the same as normal port to pipe mappings. The mapping of the one incoming pipe to the file subsystem looks the same as any other port to the user node, but the prefix treats it differently. The prefix uses the one pipe to the file subsystem for both parameter buffers and data buffers. In front of the buffer on the pipe the prefix puts two things: the connection number for this port connection (provided by UNIXSC) and the type of the buffer being sent. The file subsystem knows who sent the buffer by the connection number. This allows it to also tell which set of pipes to

use to respond over if a responce is required. The buffer type tells the file subsystem how many bytes to take off the pipe for this buffer.

Since the protocol between the file subsystem and the rest of the configuration has not been changed the rest of the configuration will still think it is dealing with the NADEX file system. The file subsystem must emulate the NADEX file system on the UNIX file system. This design choice was made for the portability of software and because the NADEX file system is the NADEX network file system. Since the NADEX file system is the network file system, UNIX will also need to emulate it so that when it is added to the NADEX network it is consistent with the NADEX network.

CHAPTER FOUR

Bringing Up NADEX Software Configurations On Top of the UNIX Operating System

A configuration is activated by UNIXSC when it gets a configuration descriptor from LINK. If it is a SPIN_OFF then UNIXSC makes a copy of itself with the UNIX system call FORK and this copy brings up the configuration. If it is not a SPIN_OFF then UNIXSC brings up the configuration itself.

First UNIXSC builds a PIPE_TABLE with one entry for each entry in the configuration descriptor's CD_PORT_TABLE. It does this one connection at a time. It checks to see if PARM's are used by looking at the minimum number of PARM_BUF for this connection. If PARM_BUF is more than zero then two pipes are gotten, one for each direction, by using the UNIX system call PIPE. The pipe ids are put in the two PIPE_TABLE entries for this connection. Each pipe is put in one table as read and the other as write . If no PARM_BUF's are used for this connection then no pipes are gotten for PARM's. Then the same thing is done for DATA_BUF's for this connection. This is repeated for all the connections in the configuration.

Next UNIXSC forks once for each node in the

configuration with NODE_NUM set for the entry in CD_NODE_TABLE that it wants that forked copy to bring up. Then the parent copy of UNIXSC waits for all of its children to terminate. Then if the configuration was a SPIN_OFF the parent copy of UNIXSC also terminates. If it was not a SPIN_OFF then it brings back up the command processor configuration.

Each copy of UNIXSC that is to bring up a node in a configuration sets up a parameter list for that node and then does an EXEC system call to run that node. The parameter list consists of the parameters for the node that are in the configuration descriptor, the nodes attributes, and a NODE_PIPE_TABLE which is a table of the pipes for this node. This table is used by the prefix to map port operations onto UNIX pipe operations.

To bring up the command processor configuration the user starts UNIXSC and UNIXSC brings up the rest of the configuration. It gets the pipes and sets up the PIPE_TABLE for the configuration and does a FORK for each of the other nodes in the configuration. Each of these forked copies brings up their node by using EXEC.

When a configuration, other than a SPIN_OFF, is brought up all the nodes in the command processor configuration except UNIXSC terminate. So when all the nodes, in the configuration that was brought up, terminate the command processor configuration is brought back up by UNIXSC.

CHAPTER FIVE

Future Work

Future work on the UNIX implementation is to make it a host in the NADEX network. Before this can be done the port mechanism developed at RAND [13] needs to be installed. This will allow nodes to be dynamically connected to subsystems. The dynamic connect is needed to implement NADEX's message subsystem, which is needed for interhost communication. This is because there can only be one message subsystem (MSS) on a host and it must be available to all configurations all the time.

REFERENCES

1. Bourne, S.R. The UNIX shell. The Bell System Technical Journal 57, 6, Part 2 (July-August 1978), 1971-1990.

2. Brinch Hansen, Per. The Architecture of Concurrent Programs. Prentice_Hall, Englewood Cliffs, N. J., 1977.

3. Fundis, R.M. and Wallentine, V.E. Command processors for dynamic control of software configurations. Technical Report TR-80-03. Department of Computer Science, Kansas State University, Manhattan, Kansas (1980).

4. Morrison, J.R. Data Stream Linkage Mechanism. IBM Systems Journal 17,4, 1978.

5. Ritchie, D.M. UNIX time-sharing system: A retrospective The Bell System Technical Journal 57, 6, Part 2 (Jule-August 1978), 1947-1969.

6. Ritchie, D.M. and Thompson, K. The UNIX time-sharing system. The Bell System Technical Journal 57, 6, Part 2 (July-August 1978), 1905-1930.

7. Rochat, K.L. and Wallentine V.E. A software structuring tool for message-based systems. Technical Report TR-80-04. Department of Computer Science, Kansas State University, Manhattan, Kansas (1980).

8. Rochat, K.L. and Wallentine, V.E. NADEX job control system implementation. Technical Report TR-80-05. Department of Computer Science, Kansas State University, Manhattan, Kansas (May 1980).

9. Rochat, K.L. and Wallentine, V.E. Utility programs for NADEC command processors. Technical Report TR-80-06. Department of Computer Science, Kansas State University, Manhattan, Kansas (June 1980).

10. Wallentine, V.E., Young, R.A., and Rochat, K.L. NADEX implementation notes. Department of Computer Science, Kansas State University, Manhattan, Kansas.

11. Young, R. and Wallentine, V. The NADEX core operating system services. Technical Report TR-79-11. Department of Computer Science, Kansas State University, Manhattan, Kansas (November 1979).

12. Young, R and Wallentine, V. The structure of the NADEX operating system. Technical Report TR-79-12. Department of Computer Science, Kansas State University, Manhattan, Kansas (December 1979).

13. Zucker, S. Interprocess communication extensions for the UNIX operating system: II. implementation. Technical Report R-2064/2-AF. Rand Corp, Santa Monica (June 1977).

HOSTING THE NADEX ENVIRONMENT ON THE UNIX OPERATING SYSTEM

by

DENIS EVERETT EATON

B.S., Kansas Wesleyan, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

ABSTRACT

Command language facilities for the construction and execution of software configurations -networks of communicating processes- are very limited today because current operating systems do not support this level of complexity. The Network Adaptable Executive (NADEX) was designed to support dynamic configurations -those configurations which are constructed at command interpretation time- of cooperating processes. These dynamic configurations include arbitrary graphs which may contain cycles. The NADEX environment runs on top of the NADEX core operating system. The object of this work is to make the NADEX environment so it will run with the UNIX (a trademark of Bell Laboratories) operating system as its core operating system.