AN ABSTRACT OF THE THESIS OF

Peter S. Norton for the degree of Master of Science in Electrical and Computer Engineering presented on April 13, 1989.

Title: The Implementation of a Control and Development Program for a Local Area Network (LAN).

Redacted for privacy

Abstract approved: _____
James H. Herzog

Rapid advances in technology have seen computing move from a large central computer to desktop microcomputers. This advance was also seen in the design of dedicated computers, computers designed to perform a few specialized tasks or duties. These computers, or microcontrollers, are often controlled by a central computer and themselves perform a large variety of tasks ranging from the monitoring of equipment, to the control of automated machines. Since these microcontrollers could be scattered around in the number of sites, many of them are interconnected by Local Area Networks (LANs).

One such experimental network of microcontrollers, the Task*Master, was developed at Oregon State University. The Task*Master system consists of a central personal computer (PC), and a number of smaller single board computers interconnected on some form of network. The PC issues commands (called tasks) to the microcontrollers to have them perform specific jobs. A number of network configurations have been developed for the Task*Master system, but none have included a application program for organizing and scheduling the Task*Master units. The Network Control Program (NCP) provides these needed capabilities.

The NCP program provides high level support for the Task*Master system. It is designed to be a development enviroment for the creation and execution of programs to manipulate the Task*Master microcontrollers in a useful manner. To achieve that goal, it contains access to a text editor, file manipulation facilities, lists of the Task*Master units connected to the PC and the tasks that they can perform, and a program interpreter that

executes programs written in a simple programming language. The NCP program language contains the basic programming structures, such as IF/THEN statements or WHILE loops, as well as arithmetic and logical operators and functions. A number of other functions provide access to the Task*Master network for the transmission of commands, and reception of data.

The program provides the Task*Master user the ability to create a program that controls the Task*Master units in whatever fashion he desires. The type of control system does not matter, as the PC can be reprogrammed to fit the situation.

The Implementation of a Control and Development Program for a Local Area Network (LAN)

by

Peter S. Norton

A THESIS

submitted to

Oregon State University

In partial fulfillment of
the requirements for the
degree of

Master of Science

Completed April 13, 1989

Commencement June 1989

APPROVED:

Redacted for privacy

Associate Professor, Electrical and Computer Engineering in charge of major

Redacted for privacy

Head of Department of Electrical and Computer Engineering

Redacted for privacy

Dean of Graduate School

Date Thesis Is Presented April 13, 1989

Typed by Peter Norton for Peter Norton

Acknowledgement

I wish to thank Professor Herzog for all of the help he has given me in getting this thesis completed. I also wish to thank my sister Margaret, for her poofreading, which saved me more frustration than I can imagine.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

THE IMPLEMENTATION OF A CONTROL AND DEVELOPMENT PROGRAM FOR A LOCAL AREA NETWORK (LAN)

1. INTRODUCTION

1.0 Background

In the years since computers have been invented, they have become progressively smaller and more powerful. This progression has benefited people and industry by taking the computer from a big "computing room" to the desk top of the user. Spread out between offices, or buildings, it became necessary to find a way to share the resources that the user needed. Resources such as printers, plotters, or even intangible objects like data bases were originally always available, since all computing took place in a single area. When the computer moved from the user's office, it became difficult and time consuming to access these things. This is when computer networks, specifically local area networks (LANs), became more and more useful, allowing the user to communicate with other computers and access their capabilities, whether on another floor or in another building.

People and businesses were not the only ones to benefit from the improvements in the computers and computer networks. Machines and the automated industry also benefited. Smaller, more powerful computers meant that processes could be run by dedicated controllers, i.e., computers designed for the specific purpose of running a particular machine. The connecting of a LAN to these machines permitted a main computer to synchronize all the machines to perform their tasks in the most efficient way: A particular machine's own computer would monitor its processes and instantly make adjustments based on that machine's actions and the instructions received by the main supervising computer.

However, the use of "intelligent" controllers need not be limited to the factory. Any place a process must be monitored and adjusted, such a controller could be used. In scientific sensors, burglar alarms, learning devices for schools, the list is endless.

1.1 Task*Master

The Task*Master system, was a distributed microcontroller system developed at Oregon State University. The system was created to test the idea

of a simple real-time control system utilizing multiple microcontrollers controlled by a single scheduler [HERZ 87].

The original Task*Master system was organized with a personal computer (PC) as the central scheduler, and the individual microcontrollers link to it and each other in a daisy-chain structure. The scheduler and the individual Task*Master units were organized in a master-slave fashion. The individual units acted on the scheduler's instructions and could only respond by its permission.

Commands to the Task*Master units were in a "task" format. The tasks were commands that utilized the resources or hardware of the microcontroller to perform a specific action. Each task is a pre-programmed subroutine written specifically for the microcontroller and its purpose. While simple in operation, with the proper sequencing of different tasks, the microcontroller is capable of complex operations [HERZ 87].

1.2 The COLANs

The Task*Master system was modified to change its interconnection structure from the daisy-chain to that of a LAN. These modified systems, called COLANs, for Control Oriented Local Area Network, were designed to bring a high performance control and mailing network to the older Task*Master system. The first COLAN was developed by Yue-Peng Zheng in 1986 [ZHEN 86], and since then three others have been proposed: COLAN II [KAO 87], COLAN III [EUM 87], and COLAN IV [THYE 88]. Each provided a different method for implementing the LAN, but maintained the basic structure of the Task*Master microcontroller system.

Each of the COLANS, as well as the original Task*Master system, provided a simple application program to control the units on the network. These programs generally consisted of menu driven routines that allowed the user to select the specific microcontroller (called NIUs or Network Interface Units [THYE 88]) and the task that it was to execute. Some of these programs also provided other functions as well, such as allowing monitoring of the network or transmission of files or messages.

However, none of the application programs provided what was originally intended for the Task*Master system: A system scheduler that

provided control and synchronization for all of the connected microcontrollers.

1.3 The NCP Program

The NCP, or Network Control Program, was written to provide the control and scheduling that was needed for the systems. Designed to run on the host PC, it is compatible with all of the Task*Master based systems, provided the interface between the host and the NIU remains the same. Transparency in the network structure permits this. The NCP program does not see what kind of network it is on, whether daisy-chain or token ring; it only sees the information passed to it from an NIU.

The heart of the NCP is a program interpreter. It takes an actual program written by a user and runs it. The program may consist of tasks for the microcontrollers, or it may have other sorts of computations or calculations. The NCP provides all of the basic program structures, such as If/Then or For/Next statements, as well as a number of useful functions. It also supports variables, with characters, strings and integer types available.

The NCP, however, is designed to be more than a control program, it is also a development environment. The user, when running the program, has access to an editor, a directory, and a files manipulation function. A list of the microcontrollers can be accessed, as well as a library of the tasks that they can perform. Even the operating parameters of the program itself may be accessed and changed. All of this allows a user to sit down and write a program that will permit not only the execution of complex operations on a microcontroller, but the synchronization of that controller with other controllers to perform even larger interactive tasks.

1.4 Chapter Overview

Chapter 1 has dealt with the background of and introduction to the NCP. Chapter 2 will cover the design issues surrounding the creation of the NCP and the actual implementation of the program. Chapter 3 contains my conclusions and recommendations for future programs.

2. The Design and Implementation of the NCP

2.0 Overview

The NCP program consists of two main parts, the interpreter and the support functions for the environment. Of these parts, the interpreter required a number of major design decisions. This chapter deals with these decisions as well as the implementation of the interpreter and associated functions. Also discussed is the PC to NIU interface and the protocol used in it.

2.1 Main Functions

The NCP program was designed to be a development system, and so has a number of functions that supported the creation of programs for the interpreter. Some of these functions, such as the Directory Information, which allows access to a disk's files, and the file viewing function, are straightforward, but others require some discussion.

2.1.1 The Editor

For a development environment, it is vital to have some sort of text editor. Since a good text editor is difficult to program, and most people already have an editor they are comfortable with, it was decided not to include an editor with the NCP program. Instead, access to an editor from the NCP environment was programmed so the editor could be used without leaving the NCP program.

2.1.2 Program Setup

In the program setup section, a user may make adjustments to some of the NCP program's parameters. The Program Setup section has four subsections: File Directories and Paths, which allows a user to change the directory of the NCP's support files; the Communications Parameters section, which allows the adjustment of the operating parameters of the computer's serial port; the Program View Screen function; and the Transmit Toggle. The first two are straight forward, but the last two require further explanation.

To debug a program properly, especially one that uses the serial communications port, it is often necessary to see the data being transmitted or received. The Program View Screen function permits such data to be displayed on the computer screen. Data received on the serial port is displayed on the screen, as is any data that is transmitted. The received data is enclosed in "< >" symbols and the transmitted data is enclosed in and "* *" symbols. Program output may also be turned on and off from this function.

The transmit enable/disable function toggles the communication serial port on and off for transmission only. This allows a user to test and debug a program without actually transmitting commands out to the NIUs. The data that would have been transmitted can be still be seen on the screen by setting the view to display the transmitted data.

2.1.3 The Task Library and the List of NIUs

The Task Library and List of NIUs, are the most important of the environment support functions. They provide information not only to the user of the NCP program, but also for the interpreter.

The Task Library lists all of the tasks the NIU can perform. The tasks are listed in order by their number. Each listed task also has a brief, helpful description with it, permitting easy reference when writing programs. It is important to know that the list is only as accurate as the user makes it. If changes are made in the firmware of the NIU, these changes will not be reflected on the list unless the user puts them there.

The NIU list is very similar to the Task Library. It lists all of the NIUs that are attached to the host, or network. The NIUs are listed in order by address, and each, as in the task library, has a short description following it. Again, if new microcontrollers are added or removed, the list will not reflect this, unless the user make the appropriate changes. However, this function contains a polling feature that orders all of the NIUs to report back their addresses. Those that respond are indicated on the list by an asterisk.

Both the Task Library and the NIU list have another important feature: the interpreter accesses their information when sending commands to a microcontroller. The commands can be sent using either numbers for the address and task, or the NIU or task names. If the names of

either the task or NIU are used, the interpreter checks the list to find which numbers to use.

2.2 The Interpreter

2.2.1 Program Statements and Format

When the interpreter was written, a decision had to be made on the type and format for the statements the programs would use. One of the design criteria was that the commands be simple and reasonably easy to use. As a result, the formats from a number of languages were used. Examples of this are the use of BEGIN and END statements to define program blocks, which comes from PASCAL. The format for declaring variables is similar to that in C. Others, such as the semi-colons at the end of a statement and the use of a NEXT with a FOR statement and a WEND with a WHILE statement, were used because the parser required ends of statements to be explicitly declared.

Once the statements themselves were decided, all of the basic flow control and decision making statements, such as IF/THEN statements, FOR/NEXT, and WHILE/WEND, were included. Also included were all of the basic logic and arithmetic functions and assignment statements. Since the CHAR type was included, as well as arrays, a number of string functions were added. Strings were implemented simply as character arrays, and the functions provided may concatenate, compare, copy and convert them. Access to the computer's time, date were provided for, as well as access to the parameters of the serial communications port.

A few statements were added that are peculiar to the NCP program and its purpose. The WAIT and WAIT UNTIL statements are provided for scheduling the microcontrollers. These statements provide for a time delay. The WAIT statement delays the execution of the program for a number of seconds. WAIT UNTIL delays execution until a specified time or date.

The SEND and RECV statements provide access to the microcontrollers through the communication port. The statements allow the sending of commands to a specified NIU and the reception of data from an NIU.

2.2.2 Lex and YACC

The interpreter depends on two important programs, the lexical analyzer, which processes the user written program files, and the parser, which finds the order of the programs.

The lexical analyzer and the parser were created using two application programs called, respectively, Lex and YACC. Both take a series of "rules" that specify their actions, and when Lex and YACC are used, they produce source code for programs based on these rules. Lex and YACC were used because they provided a simple and easy way to generate two rather complicated programs. They also provided an easy and quick way to make changes, even complicated ones, to the lexical analyzer and parser.

Once the parser has matched some of the input with its grammar rules, support functions are executed that do the actual work of the interpreter.

2.2.3 The Lex Preprocessor

The lexical analyzer takes a text file as input, and outputs numbers or tokens that correspond to the input it has recognized. For example, if the lexical analyzer recognizes the word BEGIN, it returns the value 270. It is this token, and others, that the parser attempts to match to its own rules. However, since the lexical analyzer runs through the input once, loops and other similar structures in a network program cannot be implemented. The problem was overcome by creating a preprocessing function.

The preprocessing function is called "yylex", which is the name of the lexical analyzer function called by the parser. The actual analyzer is renamed "yylex2". The preprocessor first calls the lexical analyzer for input. It then checks the input to see if the token that has been returned matches the beginning of a loop (i.e. FOR or WHILE). If it does, the token and all other vital information associated with it is stored in a linked list. The location of the beginning of the loop in the list is then temporarily stored in memory. After that, all input from the lexical analyzer is stored in the linked list, essentially making a copy of the loop. When the end of the loop is recognized, it too is stored in the list, immediately followed by the location of the beginning of the loop. In this way, if the loop is to be executed again, the preprocessor reads from the linked list to implement the

loop instead of reading from the input file. (See fig 1) Nested loops may be implemented in the same way, using the same linked list, by simply recording the beginning of each loop, and putting this information on the list.

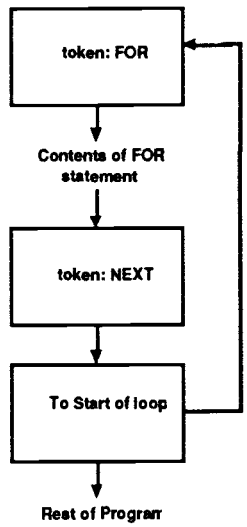


Fig. 1 Preprocessor Input List

Each time a loop is executed, the conditional at the beginning of the loop is checked. If it is false, the parser runs through the loop as it normally would, but it sets a flag to prevent the execution of any of the functions contained within the loop.

This flag is also used by the IF/THEN statement. The lexical analyzer and parser cannot skip over portions of the input. The input must be read in and parsed as it normally would be. To prevent the execution of any functions contained in a "false" section of code, a flag must be set. For example,

```
if (x=1)
        then  print  ("one");
        else  print  ("not one");
```

If "x" equals 1, the THEN statement must execute, but not the ELSE statement. However, the parser must parse all of the input, including the ELSE

statement. So a flag is set to indicate which statement can be executed. If "x=1" is true, the flag is set to true, and the THEN "print" statement executes, after which the flag is set to its opposite value. Then the ELSE statement will be read in and parsed, but the PRINT statement will not be executed.

Since IF/THEN statements may be nested (i.e., one within another), the execute flag must have several levels. This was done by making it an array of flags. The level of the array is advanced when another level of nesting is reached. When the nesting level is dropped by one, the array level also decreases. To find out if a statement can be executed or not, all of the flags in the array are ANDed together. If the result is a "1," then the statement may be executed; if it is a "0," it may not. For example:

```
1          if (x=1)
2                then if (y=1)
3                          then print ("one and one");
4                          else print ("one and zero");
5                else print ("x equals zero");
```

The above demonstrates nested IF/THEN statements. Figure 2 shows the array of flags and its value at each line. If the value of "x" and "y" are 1 and 0 respectively, then figure 2 shows the result. The first position of the array is always 1, as statements outside a loop or IF/THEN statement are always executed.

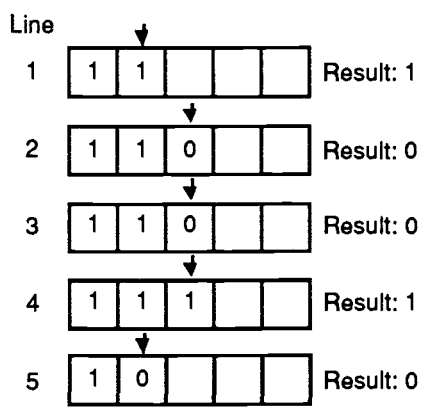


Fig. 2 Execute Flag Array

2.3 PC to NIU Interface

One of the more important issues in designing this program was the communications protocol between the PC and the NIUs. For the NCP program to be compatible with the microcontroller network, their protocols must be identical. There are two levels to this protocol. There is the low level scheme of transmission and reception of data and acknowledgements, and there is the higher level with the command packet format. The NCP program uses the old Task*Master system for both levels.

The PC is connected to the NIU network via its serial communications port. Assuming that the various parameters such as baud rate and parity are correct, then the two pieces of equipment can exchange data. There is not any system of acknowledgements for data received. Data that is transmitted onto the serial communications link is either received or lost. Garbled packets are ignored and also lost. This is not considered a problem, due to the reliability of the RS232 link.

The NCP program uses the command format from the old Task*Master system, as shown in figure 3a. Data received by the PC is enclosed by square brackets, as shown in figure 3b. Beyond the delimiters, there is no internal format to the data.

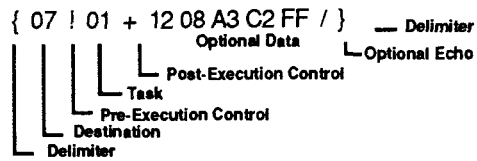


Fig. 3a Task*Master Command Packet

[07 00]

Fig. 3b Task*Master Data Packet

The COLANs use a different format for the command packet, depending on the destination of the packet. Packets going to the local NIU (i.e. the NIU attached to the PC) have a format that is different from packets going to remote nodes. Incoming data is sometimes enclosed by square brackets, while at other times it is not. These differences make the NCP program incompatible with the COLANs. This incompatibility can be fixed by writing an interface function for the NCP program. The interpreter has such a function include in it.

3. User's Manual for NCP

3.0 In the Manual

Section 3.2 deals with the main functions of the NCP program. Each sub-section covers in detail how to use the Editor command, the libraries, the directory function and other functions. The Run Program command is only briefly covered in this chapter, as it is covered in depth in Section 3.3.

Section 3.3 covers the Run Program command and all of the functions and program structures that the interpreter understands. Examples of the use of the program statements, as well as programming examples, are shown in this part of the text.

Section 3.5 covers the protocol between the PC and the NIU (network interface unit) to which the PC is attached. The translator function for the NCP program is also explained here, with examples of how to change it to fit the protocol requirements of the PC to NIU interface. Section 3.5 also deals with, respectively, YACC and Lex theory, covering how they work, how to program with them, and how to adapt them to work with the PC C compiler.

3.1 A Brief List of NCP Commands

Keystrokes	Description
F1 or Alt-E	Evokes the text editor specified in Setup menu.
F2 or Alt-P	Calls the interpreter and runs the specified program.
F3 or Alt-T	Displays a list of tasks understood by the NIUs.
F4 or Alt-N	Displays a list of the NIUs attached to the network.
F5 or Alt-D	Evokes program to allow manipulation of directory information.
F6 or Alt-V	Displays specified file on the screen.
F8 or Alt-G	Invokes the DOS command processor and permits the user to run other programs or DOS commands while still in the NCP program.
F9 or Alt-S	Displays the Setup menu, which allows the user to change the communication port parameters, set the view port, disable the communications port, and set the path to systems files.
F10 or Alt-X or ESC	This exits the NCP program.
Alt-H	This provides on-line help for the user.
Alt-Z	In the main menu, this toggles the menu screen on and off.

3.2 Main Functions of the NCP

3.2.1 The Main Menu Screen

The NCP main screen has three sections of interest. The first is the output window. This is the top window of the main screen, where all program output, as well as data received and transmitted on the serial port, is displayed. The window may be expanded to provide more room for program output by using the Zoom function (see **Zoom**). Program output appears as normal text in this screen, while received data is surrounded by bold angle brackets and transmitted data is surrounded by bold asterisks. Any or all of the program output may be turned off by the Program View function (see **Setup Menu: Program View Screen**).

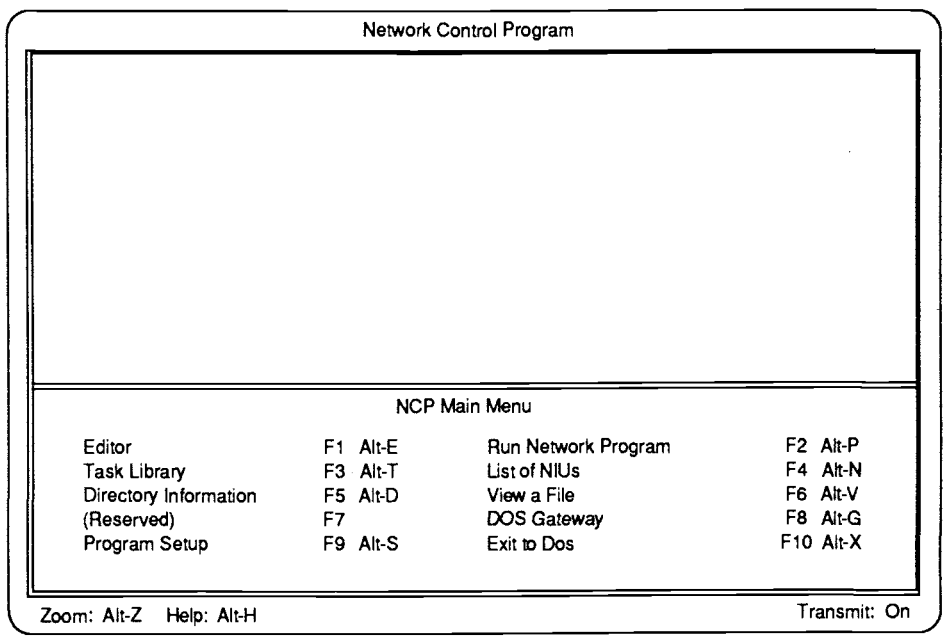


Figure 4: Main Menu Screen

The second window is the NCP Main Menu Screen. It contains a list of the commands that are usable from the main menu. This screen can be made to disappear by using the Zoom function.

The third point of interest is the bottom line of the screen. It lists the keystrokes for the Zoom function and the Help function. It also displays the status of the transmitting portion of the serial port. If Transmit is ON, data that is sent to the serial communications port will be transmitted out to the NIU and network. If it is OFF, data sent to the serial port will not be transmitted, although it will still be printed out on the output screen. This status may be toggled on or off by using the Transmit Enable/Disable function (see **Setup Menu: Transmit Enable/Disable**).

3.2.2 The Editor

Keystroke: Function Key 1 (F1) or Alt-E

Description:

The key command invokes an editor for the user. The editor is an external program specified by the user in the Setup Menu (see **Setup Menu: File Directories and Paths**). The NCP program is not exited, but remains suspended until the user quits the editor program, whereupon the NCP program is restored.

As the NCP program remains in the computer's memory, occasionally there may not be room for the editor program to be loaded. In that case, a message box will appear with the message "**Error: Not enough memory.**" If this happens, the NCP program will have to be exited before the editor can be used, or another, smaller editor used instead.

3.2.3 Run Program Interpreter

Keystroke: Function Key 2 (F2) or Alt-P

Description:

This key command invokes the program interpreter to execute a network program. When the command is given, a message box appears asking for the program to be executed.

Figure 5: Run Program message box

When the program begins to run, the message "**Running Program xxxxx**" will appear in the view screen. Any output from the program will also appear on the view screen, along with data received and transmitted by the communications port, if the view port is set to do so (see **Setup Menu: Program View Screen**). When the program ends, the message "**End of Program xxxxx**" is displayed.

If the user desires to run a program without entering the NCP development environment, he can easily do so by typing the name of the program to be executed after the NCP program name when he starts NCP in DOS. For example,

A:> NCP first.tst

will run the program **first.tst** without actually entering the NCP program environment. The start and end messages and the program output appears as they normally would, but that is all. The libraries and other menu functions will not be accessible when NCP is used in this manner.

3.2.4 Task Library

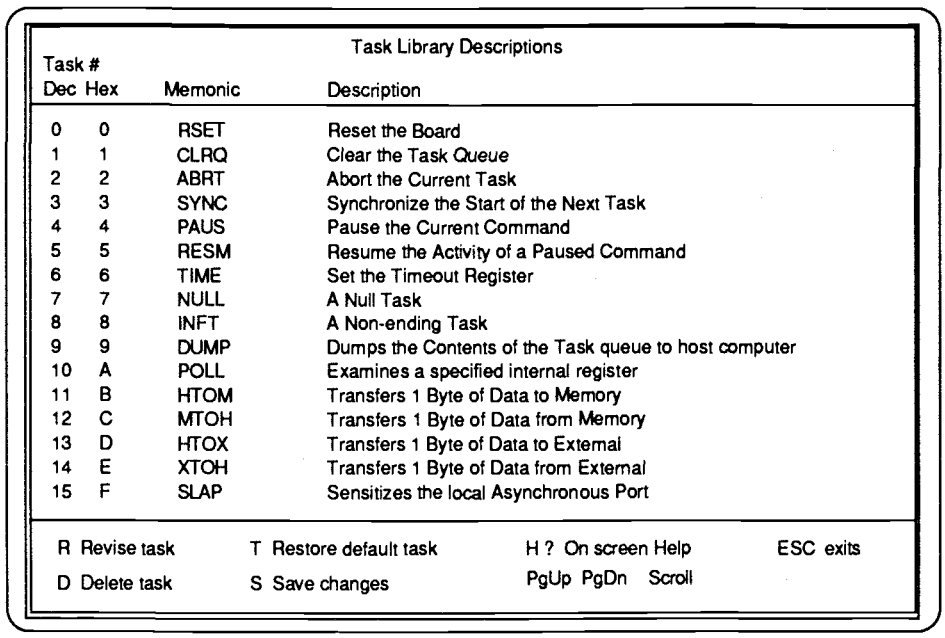
Keystroke: Function Key 3 (F3) or Alt-T

Description:

The Task Library displays a list of the commands used by the NIUs. The list is provided only as a reminder to the user. If a task is changed in the firmware of the NIU, the user must change the listing in the library

accordingly. The listing also provides a lookup table for the interpreter when a task is addressed by its mnemonic (see **Section 3.3 : Program Functions**).

The task library is displayed in order by the task number. On each line is the task number, the mnemonic of the task, and a description of the task.



Task Library Descriptions

Task # Dec	Hex	Memonic	Description
0	0	RSET	Reset the Board
1	1	CLRQ	Clear the Task Queue
2	2	ABRT	Abort the Current Task
3	3	SYNC	Synchronize the Start of the Next Task
4	4	PAUS	Pause the Current Command
5	5	RESM	Resume the Activity of a Paused Command
6	6	TIME	Set the Timeout Register
7	7	NULL	A Null Task
8	8	INFT	A Non-ending Task
9	9	DUMP	Dumps the Contents of the Task queue to host computer
10	A	POLL	Examines a specified internal register
11	B	HTOM	Transfers 1 Byte of Data to Memory
12	C	MTOH	Transfers 1 Byte of Data from Memory
13	D	HTOX	Transfers 1 Byte of Data to External
14	E	XTOH	Transfers 1 Byte of Data from External
15	F	SLAP	Sensitizes the local Asynchronous Port

Figure 6: Task Library Descriptions

Selecting a Description

When the task library is entered, the selection bar will be at the top of the screen. The bar appears as a line of reverse type (white background, black letters). The cursor or arrow keys can be used to move the selection bar up or down. The **PgUp** and **PgDn** can be used to move up or down one screen (16 lines) at a time. The **Home** and **End** keys move the selection bar to the very top and bottom of the list. The selection bar is used for selecting a task description to delete, revise, or restore.

Revise Task

Revise Task allows the changing of a task description. The selection bar must first be moved to the description to be changed and the 'R' key pressed. When that has been done, a window will appear with the old contents of the description and will prompt the user to enter a new description. If a mistake is made, pressing **ESC** will exit the window without changing the description. Old entries may be kept by pressing **Return**.

Delete Task

Delete Task allows the removal of a task description from the list. Again, the selection bar must be moved to the description to be deleted and the 'D' key pressed. A message box will appear asking the user to verify the operation. Typing an 'N' for "No" will leave the section unchanged, while typing 'Y' will replace the mnemonic with a line of dots and the description with the statement "Not currently in use." Once a description is deleted, it can be recovered if the NCP program is exited and then reentered. If the changes are saved before this is done, then the removal of the description will be permanent.

Restore Default Task

Pressing the 'T' key will produce a window asking the user for verification on the Restore Default Task operation. If **No** is selected, then the window is closed and nothing is changed. If **Yes** is selected, then the currently selected task description is returned to its default setting. This is the original set of task descriptions provided with the NCP program. If **All** is selected, then all of the task descriptions in the library will be restored.

At times the user may wish to change the task descriptions permanently, or he may wish to have several default task descriptions. The default list and the changes made to it, are stored in a directory specified in the **File Directories and Paths** section of the Setup Menu (see **Setup Menu**). By changing the Support Files path to a new directory, the user can switch to a new default list.

To create a new default list, first make all desired changes to the list. Then, one by one, select all of the unchanged task descriptions wanted in the new default list and revise the descriptions without making any

changes to them. This is done by simply typing a carriage return twice. This marks the description as being changed, and when the changes are saved, the old descriptions will also be saved.

Once the changes have been saved using the Save function, exit the Task Library and the NCP program. In DOS, copy the file "TLIB-SUP.NCP" to the new directory and then rename the file to "TLIB-DFT.NCP". Once the Support Files path in the Setup Menu has been changed to the new path to the default library, the NCP program will use the new library in place of the old. In this way, the user may have several libraries, switching between them, by changing the Support Files path. *Note: The changes will only take place when the NCP program is exited and then reentered.*

Save Changes

The changes made to the task descriptions may be saved to the disk by pressing the 'S' key. If the changes are not saved, they will be lost when the NCP program is ended, although they will remain in effect until then.

Help

Pressing the 'H' or '?' key will invoke the on-line help function. A window will appear with information on how to use the Task Library function. **ESC** will let the user exit the Help function.

Exit NIU List

Pressing the ESC key will leave the NIU List. Any changes made to any descriptions will remain until the NCP program ends, at which time they will be lost. If the **Save** function is used, the changes will be permanent.

3.2.5 List of NIUs

Keystroke: Function Key 4 (F4) or Alt-N

Description:

List of NIUs displays a list of the NIUs (Network Interface Units) that are connected to the network. The list is provided only as a reminder to the user, as any changes to the list, such as adding or removing an entry, must be done by the user. For example, if the network contains 5 NIUs with the addresses of 1, 2, 3, 4, and 5, the list will not contain these addresses unless they are added by the user. Once the information is added, it is available to the user to look at when programming.

The List of NIUs function has one other use: when commands are sent to the NIU, the name of the NIU may be used instead of the address. In that event, the interpreter will look up the name in the list to find the corresponding address of the NIU.

NIU List and Descriptions

Node # Dec	Hex	Name	Active	Description
0	0	ALL		Allows broadcasting to all nodes
1	1	.....		Not currently in use
2	2	.....		Not currently in use
3	3	.....		Not currently in use
4	4	.....		Not currently in use
5	5	.....		Not currently in use
6	6	.....		Not currently in use
7	7	.....		Not currently in use
8	8	.....		Not currently in use
9	9	.....		Not currently in use
10	A	.....		Not currently in use
11	B	.....		Not currently in use
12	C	.....		Not currently in use
13	D	.....		Not currently in use
14	E	.....		Not currently in use
15	F	.....		Not currently in use

R Revise description P Poll active nodes S Save changes ESC exits
D Delete description C Clear active nodes PgUp PgDn Scroll

Figure 7: NIU List

Selecting a Description

When List of NIUs function is entered, the selection bar will be at the top of the screen on line 0. The bar appears as a line of reverse type (white background, black letters). The cursor or arrow keys can be used to move the selection bar up or down. The **PgUp** and **PgDn** can be used to move up or down one screen (16 lines) at a time. The **Home** and **End** keys move the selection bar to the very top and bottom of the list. The selection bar is used for selecting a description to delete or revise.

Revise Description

Revise Description permits the changing of an NIU description. The selection bar must first be moved to the description to be changed and the 'R' key pressed. When that has been done, a window is produced with the old contents of the description, and the user may enter a new description. Old entries may be kept simply by pressing return. **ESC** will exit the function without making any changes.

Delete Description

Delete Description allows the removal of a description from the list. Again, the selection bar must be moved to the description to be deleted and the 'D' key pressed. A message box will appear asking the user to verify the removal of the description. Typing an 'N' for "No" will leave the section unchanged, while typing 'Y' will replace the name with a line of dots and the description with the statement "Not currently in use." Once a description is deleted, it can be recovered if the NCP program is exited and then reentered. If the changes are saved before this is done, then the removal of the description will be permanent.

Poll / Clear Active Nodes

The Poll function lets the user poll the network for attached and functioning NIUs. Pressing the 'P' key will send a command to all NIUs on the network. If they are active, they will respond to the command and send back an acknowledgment. If an acknowledgment is received, an asterisk '*' will be displayed under the Active column on the line corresponding to the

NIU's address. While this function is running, a message box will be present informing the user to press ESC to end the polling.

Pressing the 'C' key will clear the Active column to allow the user to poll the network again.

Save Changes

Any changes made to the descriptions may be saved to the disk by pressing the 'S' key. If the changes are not saved, they will be lost when the NCP program is ended, although they will remain in effect until then.

Help

Pressing the 'H' or '?' key will invoke the on-line help function. A window will appear with information on how to use the List of NIUs function. **ESC** will let the user exit the Help function.

Exit NIU List

Pressing the **ESC** key will leave the NIU List. Any changes made to any descriptions will remain until the NCP program ends, at which time they will be lost. If the **Save** function is used, the changes will be permanent.

3.2.6 Directory Information

Keystroke: Function Key 5 (F5) or Alt-D

Description:

The directory information function provides the user a way to view the contents of the disk drives on the computer. It also allows the copying, deleting, renaming and viewing of files on the disks. Also provided are commands to change, create, delete, and view directories. The information is displayed in a easy-to-read format, enabling the user to perform many useful functions without exiting the NCP program.

Top Line

The top of the Directory screen are two items of information that the user will find useful; directory information, and disk information.

The Directory Information shows both the current directory and the default directory. The current directory is the directory whose information is currently being displayed on the screen. The current directory will change when the directory is changed by either the change directory command or the view command.

The default directory line shows the path on which any file operations will take place. Usually the current and the default directories will be the same, but when the View function is used, they can be different.

The Volume Name displays the name of the disk whose information is currently being displayed. If the disk has no name, then "**(none)**" will be shown.

The Disk Space displays the amount of free space on the current disk drive.

Directory Format

The Disk Information is displayed in the center window. It shows the filenames, extensions, size, and time and date of the file's creation. If an entry is a sub-directory, then the symbol "<DIR>" replaces the file size.

Selecting a Description

To select a file for copying, renaming, deleting, viewing, etc., the user can use the selection bar. The bar appears as a line of reverse type (white background, black letters). The cursor or arrow keys can be used to move the selection bar up or down, left or right. The **PgUp** and **PgDn** can be used to move up or down one screen (16 lines) at a time. The **Home** and **End** keys move the selection bar to the very top and bottom the of the list. The selection bar is used to select entries for viewing, copying, renaming, or other operations.

Directory Mask/New Directory Mask

When the Directory program is first called, a window is created, which asks the user for a directory mask with the characters "*.*" already

entered. The directory mask is used to determine which files are to be displayed. If the name "testfile.txt" is entered, then the directory function will only display the file "testfile.txt" if it exists. This, however, is not the easiest or the best way to use the directory program. More than one file may be displayed at one time if wildcards are used.

Wildcards are used to match single or multiple letters in a filename. The asterisk '*' matches any number of characters, while the question mark '?' matches only a single character. For example, the "*.*" matches all files with any number of characters before the period, and any number of characters after it. In this way it matches all files on the disk. Other examples are as follows:

"*."	matches all files with no extension, i.e. "test".
"*.NCP"	matches all files with the extension "NCP".
"TLIB*.NCP"	matches all files starting with "TLIB", which have the extension of "NCP". The same could have been done with "TLIB????.NCP".
"TEST?.TXT"	matches all files starting with "TEST" followed by a single character and end with the extension "TXT". This would match "TEST1.TXT", "TEST2.TXT", "TESTS.TXT" and so on.

Copy File

To copy a file, press the 'C' key. A window will appear asking for the source and destination file names. If the selection bar is positioned on a filename, that name will appear as the source filename. If the source filename is correct, then a carriage return will accept the name; otherwise the user may change it. Pressing **ESC** will exit the delete function without making any changes.

Delete File

To delete a file, press the 'D' key. This will bring up a window asking the user for the name of the file to delete. If the selection bar is currently positioned on a filename, that name will appear as the name of the file to

delete. The user may delete the named file or enter a different name. Pressing **ESC** will exit the delete function without making any changes.

Rename File

The Rename File function lets the user rename a file. Pressing the 'R' key will produce a window similar to the Copy window. The user will be prompted for an old name and a new name. If the selection bar is currently positioned on a filename, then the old name will be that of the file currently selected. The user may accept the name or change it as desired. Pressing **ESC** will exit the rename function without performing any changes.

View File or Directory

By pressing the 'V' key, the file or directory currently selected by the selection bar will be displayed on the screen. If it is a directory, then the directory selected will be displayed. This will change the current directory line at the top of the screen, but will not change the default directory. In other words, the user is only looking at the directory; he has not actually changed it.

If a file was selected, then the View File function (see **View File**) is called, and the contents of the file are printed on the screen.

Change Directory

The default (and thus the current) directory can be changed by pressing the 'H' key. A window will appear asking the user for the name of the directory which to change. If the selection bar is currently on a directory entry, then that name will appear in the window. Otherwise the window will be empty, and the user must enter the required information.

Once the directory has been changed, then the displayed information will change to that of the new directory, and the default and current directory information will be altered accordingly.

Pressing **ESC** while in the Change Directory window will exit the function without making any changes.

Make/Remove Directory

The Make and Remove Directory functions allow the user to make and remove sub-directories. Pressing the 'M' key will bring up a window asking for the name of the directory to be created. When the new name has been entered and a carriage return pressed, the new directory will be created as a sub-directory of the current one.

By pressing the 'E' key, the user may remove or delete a sub-directory. If the selection bar is currently positioned on a directory entry, then that name will appear in the window. The name may be accepted by pressing <CR> or changed as desired. If the sub-directory the user is attempting to remove is not empty, then an error will result and the deletion will fail.

Pressing **ESC** while in the window of either function will exit the function without making any changes.

Help

Pressing the '?' key will invoke the on-line help function. A window will appear with information on how to use the Directory function. **ESC** will let the user exit the Help function.

Exit Directory

Pressing the **ESC** key will leave the Directory program, and return to the NCP main menu.

3.2.7 View File

Keystroke: Function Key 7 (F7) or Alt-V

Description:

The View File function lets the user examine the contents of a file. After the user presses the appropriate key, a window will appear, asking the user for the name of the file to view. Sometimes a name will already be provided for the user. This name is the name of the network program the

user last ran. This is provided as a shortcut to the user who is developing and testing a program.

When the user has entered a filename, the View program is called and the contents of the file are displayed on the screen.

The View program is simple and only allows the user to examine of the file's data. It will print out one screen of data at a time. Pressing any key will print out another screen full of data.

Pressing **ESC** will exit the view program and return the user to the NCP main menu.

3.2.8 DOS Gateway

Keystroke: Function Key 8 (F8) or Alt-G

Description:

DOS Gateway invokes the command processor, "Command.COM", and suspends the NCP program to allow the user to perform DOS functions. Any sort of DOS functions may be used, and other programs may be run as well if memory permits. When the user is done, typing 'EXIT' will return him to the NCP program's main menu.

3.2.9 Program Setup

Keystroke: Function Key 9 (F9) or Alt-S

Description:

The Program Setup function the setup menu for the NCP program. It allows the user to change many of the default settings of the program, such as the parameters of the Communications Port and the File Directories and Paths, the Program View screen. It also allows the user to enable/disable the communications port for transmitting data.

The sub-function the user desires to change may be selected by using the cursor or arrow keys to move the selection bar. When the bar is on the sub-function desired, pressing <CR> will take the user to that function.

File Directories and Paths

File Directories and Paths holds the directory paths to certain files or programs. The first path is the one to the Support files. These are the Task Library default and supplemental lists and the NIU list. The Support Files path may be changed, if the user has more than one set of Task and NIU descriptions he.

The View path is the path and filename to the program used by the View File function. The default filename is "NCPVIEW.EXE". The entry is provided to allow the user to substitute a different program if such is desired.

The Editor path is provided to allow the user to use the text editor that he is familiar with. No editor is provided with the NCP program; to use the Editor function, this entry must be filled with the path and filename of a text editor chosen by the user.

Pressing **ESC** will exit this function.

Communication Parameters

The Communication Parameters sub-section permits the changing of the parameters of the communications port. When this function is selected, a window will appear which shows all of the parameters currently in place. Other windows will appear which allow the user to select new parameter values. The user may change the serial port used and the baud rate, as well as the number of the data bits and stopbits and parity. When all of the parameters have been selected, the user will be returned to the Setup menu.

Pressing **ESC** at any time will return the user to the Setup menu without making any changes.

Transmit Enable/Disable

Transmit Enable/Disable allows the user to turn on or turn off the communications port for Transmitting only. This is to allow the testing of a program that sends data out onto the network without actually sending the

data. The data will still appear on the screen if the View port is enabled (see **Program View Screen**). The reception of data is not changed.

When the Transmit Enable/ Disable function is selected, a window will appear briefly, informing the user of the new status of the serial port. This status is echoed at the bottom of the NCP main menu.

Program View Screen

The Program View Screen function allows the user to select what will appear on the screen when the network program is executed by the interpreter. There are three kinds of output that can appear on the screen: program output, data received by the serial port and data transmitted by the serial port. The user may turn on or off any or all of these outputs. If a particular output is turned on, it will have an asterisk '*' next to it.

Program output is data printed out by the network program in Write statements. Usually this is turned on, but it may be turned off if necessary.

Received data is data the serial port has received from the NIU. When this selection is turned on, this data is printed out as it arrives. To distinguish it from other data, it is surrounded by angle brackets printed in bold type. For example, if the characters "1F" were received, it would appear on the screen as the following: <1F>.

Transmitted data is data that the serial port is transmitting out to the NIU. When this selection is turned on, the data is printed on the screen surrounded by asterisks "*" printed in bold type. For example, if the characters "1F" were to be transmitted, they would appear on the screen as the following: *1F*. This is useful when the user needs to see what is being transmitted out onto the network. If the user wants to test a program that transmits commands, the transmitting may be disabled by the Transmit Enable/Disable function in the Setup menu. The data will still appear on the screen, but it will not be transmitted out on the serial port to the NIU.

Exit Setup Menu

Pressing **ESC** will exit the Setup Menu. If any changes were made while the user was in the Setup Menu, a message box will appear asking the user if these changes should be saved. Answering "Yes" will save the

changes to the disk, making them permanent. This means when the NCP program is exited, the changes will not be lost.

If "No" is answered, then the changes will not be saved, but neither will the user lose them. The changes will remain until the NCP program is ended, at which time they will be lost. Thus, for example, if the user changed the baud rate on the serial port and did not save the change, then the change will remain until changed again or until the NCP program ends.

3.2.10 Exit to DOS

Keystroke: Function Key 10 (F10) or Alt-X or ESC

Description:

Exit to DOS allows the user to exit the NCP program. When the appropriate key is pressed, a message box will appear asking the user to verify his intent to the exit the program. Typing 'N' or "No" will return the user to the NCP program, while typing 'Y' or "Yes" will take the user to DOS.

Any changes made to the Task Library, NIU list, or Setup Menu that were not saved will be lost when the NCP program is ended.

3.2.11 Zoom

Keystroke: Alt-Z

Description:

When the NCP program first starts, the NCP Main Menu Screen will be at the bottom of the screen, while the top is reserved for program output. If more room is needed for program output, then the Zoom function will make the Menu screen disappear and the output window expand to fill the entire screen.

Pressing Alt-Z again will restore the window to the way it originally was.

3.2.12 Help

Keystroke: Alt-H

Description:

The Help function will bring up a window with information on how to use the main menu of the NCP program. The help file may contain several pages of information, and these may be seen by pressing the PgDn key to advance through the help, or PgUp to return to the previous page.

Pressing **ESC** will exit the Help file and return the user to the main menu.

3.3 Run Program: The Interpreter

3.3.1 About This Section

This section covers the interpreter and the commands it understands. Included in this section are ways to control the interpreter and examples of programs.

3.3.2 The Interpreter

The interpreter is the core of the NCP program. It is the device by which programs written by the user to control the micro-controllers on the network are deciphered and executed.

The interpreter analyzes the code written by the user line by line and then acts upon the instructions it sees. Basic flow control statements and arithmetic statements are supported, as well as a number of useful functions.

The interpreter consists of two main sections of code. The first is the lexical analyzer, which searches the program code for patterns that it recognizes, substituting numbers (called tokens) for these patterns and then passing the tokens on to the second section of the interpreter, the parser.

The parser takes the tokens and compares them to a set of rules. These rules dictate the order in which the tokens may appear, and thus the structure of the user's program is determined. When a series of tokens matches a certain rule, then a support function is executed. A good number of support functions are provided to run all aspects of the network programs.

More on the lex analyzer and parser are provided in section 3.5 (see **Section 3.5: Lex and YACC Theory**).

3.3.2.1 Running a Program

A program written for the interpreter may be executed in two different ways. First, is inside the NCP environment, press the F2 function key, or ALT-P. This will produce a window that asks for the name of the

program to be executed. Entering the program name and pressing return is all that is needed.

The second method is to run the program from DOS by including it as a parameter to the NCP program. An example is as follows:

```
C:\> NCP prog1
```

This will execute the program "prog1". When a program is executed in this manner, the development enviroment is not entered. The program runs, and when it is finished, the user is returned to DOS.

3.3.2.2 Stopping a Program

A program may be interrupted by pressing the **Ctrl-Break** key combination. The currently executing program will stop with the message "**User Interrupt at line xxx**". Once stopped, the program will have to be restarted from the beginning to be used.

3.3.2.3 Reserved Words

The following words are reserved for the use of the NCP interpreter, and should not be used as variable names, mnemonics for tasks, or NIU names. The use of a reserved word might result in a syntax error or an unexpected result.

all	char	file	open	then
and	close	for	or	time
append	cls	get	read	to
band	comm	gotoxy	recv	until
beep	comp	if	recvb	val
begin	copy	int	send	wait
bnot	data_ready	keypresssed	sendb	wend
bor	date	mod	set	while
bxor	else	next	step	write
cat	end	not	str	xor

Table 1 Reserved Words

3.3.2.4 Precedence

In all arithmetic operations, there is a precedence that is used to determine the order in which the operations are to proceed. This is necessary, in that without the hierarchial structure, all equations would be executed in the order they were written. While somewhat workable, lack of precedence makes reading an interpretation of the arithmetic equations difficult.

The following table shows the precedence of the operators used in the NCP interpreter. While it covers a lesser number of operators, it is identical with most standard precedence tables.

Operator	Associativity
( )	left to right
- not bnot	right to left
* / mod	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
band	left to right
bxor	left to right
bor	left to right
and	left to right
xor	left to right
or	left to right
=	right to left

Table 2 Precedence

The operators at the top of the list have the highest precedence. All operators on the same line have the same precedence. For example, in the equations

x = 3+1*5 x = (3+1)*5

the answer to the first is 8, because multiplication (*) has higher precedence than the addition (+). Therefore multiplication is executed first, even though it comes after the addition. The answer to the second equation is 20, because the parentheses have the highest precedence, and so are executed first. That means the addition, which is inside the parentheses, is performed before the multiplication.

Associativity indicates the order of evaluation of the operators. In most cases, the associativity is from left to right, but in the case of the equals sign and the negation operators, it is the opposite. For them, the right operator is evaluated first. This makes sense for the equals sign, as the equation on the right must be solved before assigning it to the variable on the left.

3.3.2.5 Arithmetic Operators

The arithmetic operators simply perform standard mathematical operations. The operators available are as follows:

Operation	Name
-	Negation
* /	Multiplication, Integer division
mod	Modulo Division
+ -	Addition, Subtraction

3.3.2.6 Relational Operators

The relational operators are those that do comparisons of numeric expressions. The operators available are as follows:

Operation	Name
< >	Less Than, Greater Than
<= >=	Less Than or Equal To, Greater Than or Equal To
== !=	Equal To, Not equal To

3.3.2.7 Logical Operators

The logical operators are those which do comparisons between true and false expressions. In this program, "true" is any numerical expression that has a non-zero value, while "false" is any expression that has a value of zero. The operators available and their associated truth tables are given below:

AND

```
X  Y  |  X and Y
------+---------
0  0  |   0
0  1  |   0
1  0  |   0
1  1  |   1
```

Both values must be true for the result to be true in an "and" expression.

OR

```
X  Y  |  X or Y
------+---------
0  0  |   0
0  1  |   1
1  0  |   1
1  1  |   1
```

If either of the values is true, the result is true in an "or" expression.

XOR

```
X  Y  |  X xor Y
------+---------
0  0  |   0
0  1  |   1
1  0  |   1
1  1  |   0
```

XOR performs the "or" function, but with the exception that the result is true if either of the values is true, but not both.

NOT

X	not X
0	1
1	0

The NOT operator negates the value. If it is true, it becomes false, and vice versa.

3.3.2.8 Bit-wise Logical Operators

The bit-wise logical functions do the same thing as their counterparts, but on a bit-by-bit level. This means a bit-wise function takes the two values it is performing the operation on and does the logical operation on the first bit of each value, and then the next bit, and so on. An example of this is as follows:

x = 1 **band** 0 y = 1 **and** 0 = 0

Which does the following:

```
1 =    0000000000000001
0 =    0000000000000000
band   --------------------
x =    0000000000000000 = 0
```

This is, incidentally, the same result as if the logical "and" was used. However, in the next example, the difference is more obvious:

x = 1 **band** 6 = 0 y = 1 **and** 6 = 1

```
1 =    0000000000000001
6 =    0000000000000110
band   ----------------
x =    0000000000000000 = 0
```

Since none of the bits of either number has a matching "1" value in its counterpart, the result is a zero.

3.3.2.9 Strings and How to Use Them

The NCP interpreter allows the use of strings in variables and in some functions. In this program, a string is simply an array of characters, with one small difference, and is declared as such. The difference between an array of characters and a string is that a string is ended, or terminated, with an ASCII zero ('\0'). The ASCII zero tells the program where the string ends. This prevents functions that use strings, such as **copy** and **comp**, from running astray.

A string is declared in the same manner as a character array (See: **Variables**). The following declaration,

```
char teststring [10];
```

sets aside 10 bytes of memory for the character array under the name of "teststring". If this variable is used as a string, it can contain only nine characters, as the 10th and final character must be an ASCII zero.

The contents of a string variable may be accessed through its index in the same way as any other array. This allows the manipulation of individual elements. The string variable may also be accessed by its name alone, without the array index. This allows the entire variable to be used as a string. This is how strings are used in the string manipulating functions, or write statements. An example of this is as follows:

```
char teststring [10];
```

```
teststring [0]='H';                (String accessed by index)
teststring [1]='i';
teststring [2]='\0';               (Terminating ASCII zero)

write(teststring[0],"    ",teststring);      (Output: H  Hi)
      (Reference by name ^^^^^)
```

Strings may also be literal strings. These are strings that are entered as characters between two quotes. In the above example, a literal string is used as a separator in the write statement. Another example is in the following copy statement:

```
copy(teststring,"Hello  you");
```

In the above example, the string variable "teststring" is referenced by name, while the source string is a literal string.

Assignment of a string to a character array is important. There are two ways of doing this. The first is to assign each individual array element a character, and terminate the array with an ASCII zero. This was done in a previous example, with "teststring" being assigned the string "Hi".

The other way is to use the copy command. The copy command takes as input, a string variable as the destination string and a source string, which may be either a variable, or a literal string. Examples and more information may be found in the section on the copy command.

Finally, it is important to remember to provide enough space in the string variable to hold the string that will be placed there. If not enough space is provided, then data may be lost when copying strings.

3.3.2.10 Variables

There are three types of variable used in this program, **int**, char, and **file**. The **int** and **char** variables can be used in arrays, but the **file** type may not. There is no support for variables of any other type, including

floating point numbers. Such may be included in future versions of this program.

Int: The **int** type is used for storing integers. The integer is two bytes long, and has a range from -32768 to 32767.

```
int x;
```

Char: The **char** type is used for storing characters. The character has a range from 0 to 255.

```
char x;
```

File: The **file** type is used for storing file variables. The file variable holds information about a file that has been opened for access. The variable is initialized when a file is opened, and is used in all file accesses to indicate the file to be accessed.

```
file x;
```

3.3.2.11 Declaring variables

Variables are declared in the following format:

type variable name [[size]] [, variable name [[size]]] ... ;

The type may be any of the above types, (i.e. int, char, file). The name of the variable may be any name up to 64 characters in length. Capitals as well as lower case letters may be used, but it is important to remember that this program is case sensitive. If a capital letter is used in a variable name, it must be used whenever referencing that variable, or the program will not recognize it.

Any number of variables may be declared on the same line as long as they are of the same type. To do so, each name must be separated by a comma. The entire declaration must be ended by a semi-colon.

There is no order to the declaration of variables as long as all declarations take place at the front of the program. This means that integer variables may be declared, followed by some variables of character type, followed again by integer variables.

Examples of the declarations of variables are as follows:

```
int newvar;                (single declaration)
int x,y,z;                 (multiple declaration)

char a;                    (character declarations)
char cvar, newvar2;

file filevar;              (file variable declaration)
file file1, file2;
```

The optional size following the variable name is used in declaring arrays and represents the size of the array. If a variable is declared without the optional [size], then it is understood to be a single variable. The size of the array indicates the number of elements in that array.

The elements of the array may be accessed by specifying the index of the element desired. This is shown as follows:

```
int x [10];
int y [5], z[20];

x[0] = y[3] + z[19];

write(x[1]);
```

The elements of an array are numbered from 0 to the size of the array minus 1. This means that an array of size 10 will have elements numbered from 0 to 9.

If an array variable is used without the index, then an index of 0 is assumed. However, this does not apply with character arrays, as

referencing a character array without the index variable will result in treating the array as a string (See: **Strings**).

3.3.2.12 Program Structure

The NCP interpreter expects the program given to it to be in a specific format. The format is simple, but if the program does not follow it, an error will occur. The format for all programs to be executed by the interpreter is as follows:

```
begin

variable declarations

program statements .....

end
```

The program must start with a begin statement, and end with an end statement. The begin statement must be followed by the variable declarations. All variables to be used in the program must be declared here. If an attempt is made to declare them elsewhere, an error will occur. The program statements follow the variable declarations. There may be any number of program statements. The program statements must be followed by an end statement.

All variable declarations and all program statements (with a few exceptions) must be ended by a semi-colon. All of the program statements must also be in lower case, or the interpreter will not be able to recognise them. The specifics for each statement are found in the Program Statements section, and in the Variables section.

3.3.3 NCP Program Statements

3.3.3.1 Notes on Format

BOLD Words shown in **BOLD** letters are keywords, and are to be printed out just as they appear. If the word **SEND** appears in bold, then it will be written in a program just as it appears. All keywords must be typed out in lowercase letters. If any are typed using uppercase, the interpreter will not recognize them.

Italics Words shown in lowercase italic letters are to be replaced by whatever the words indicate. For example, if the command has the word *integer* in italics, then the user would substitute an integer or integer variable for the word *integer*. The actual word *integer* is not to be printed out.

[ ] Brackets indicate optional items. Items inside the brackets may be included if the user needs them. The brackets are not actually used in the statement. When including optional items do not type the brackets themselves but only the required items inside the brackets. The command **read** would appear as follows, both as its format, and how it is actually written:

Format: **read**(*var* [,*var*] ...) ;

Written: read(a,b);

The comma and second variable are both optional, as indicated by the brackets, but the brackets are not typed out when writing the statement.

| A vertical bar indicates either or. You may include one or the other of the choices, but not both. Do not type the vertical bar. An example is the **val** statement:

Format: **val**(*string* | *variable*);

Written: val("123");

The **val** statement will take either a string or a variable, but not both. When the statement is written out, it does not include the vertical bar.

... An ellipsis indicates that the item may be repeated as many times as necessary. Do not type the ellipsis. An example is again the **read** statement:

Format: **read**(*var* [,*var*] ...) ;

Written: read(a,b,c,d);

The ellipsis in the read statement indicates that the optional ",var" may be repeated as many times as necessary; in the example, it was repeated three times.

Punctuation All punctuation that appears in the statement format must appear in the written statement. The exceptions are those items of punctuation that were optional and were not used. The **read** statement provides an example:

Format: **read**(*var* [,*var* ...]);

Written: read(a);
read(a,b);

The comma does not appear in the first statement, as it was part of an optional item that was not used. It does appear in the second statement, in which the optional item was used. In both cases, the parentheses and semi-colon were required and were used.

3.3.3.2 BEEP

Purpose: Produces a short tone from the computer's speaker.

Format: **beep;**

Description:

This function produces a short tone from the computer's speaker. The tone has a frequency of 900 Hz. and a duration of 250 milliseconds.

3.3.3.3 BEGIN/END

Purpose: Indicates the beginning of a block of code. It also indicates the beginning of the program.

Format: **begin**

Description:

The **begin** statement indicates the beginning of a block of code. It also indicates the beginning of a program. The **begin** is terminated by a matching **end**.

The **begin** statement is useful when using **if/then/else** statements. The **then** or **else** of an **if** statement only executes one line of code. Using a **begin/end** pair allows the **if** statement to execute a block of code. An example of this is as follows:

```
if (x=1)
      then  begin
                  write(x);
                 x=x+1;
            end
      else x=x-1;
```

```
y=x;
```

The **begin/end** pair defines the block of code containing the two statements "write(x);" and "x=x+1," which is executed when the **if** statement is true. In comparison, the **else** section can only execute the one line "x=x-1;". The "y=x;" statement is separate from the **if** structure, and is always executed.

See also: end, and, if/then/else

3.3.3.4 CAT

Purpose: To concatenate two strings.

Format: **cat** (*var*, *string* | *var*);

Description:

The **cat** function concatenates two strings. The first string is the destination string and must be a variable consisting of a character array. The second string may be a string or a variable consisting of a character array. When the function performs the concatenation, the second string is appended to the first, and the result is stored in the first string.

An example of this is as follows:

```
char result[12];          (result contains the string "HI")

write(result);            (Output>     "HI")

cat(result,"  there");

write(result);            (Output>     "HI  there")
```

Care must be taken to ensure that the resultant string does not exceed the size of the destination array. If it does, then any excess might be lost. In the above example, the resultant string ("HI there") was 8 characters long, 9 characters with the null character terminator. Since this is less than the 12 characters that the array was defined to have, the **cat** function worked without any problems. Had the example concatenated the following string, instead of " there," data would have been lost:

```
cat(result," there again");
```

The resultant string should have been "HI there again," but since the character array "result" was only 12 characters long, the output of the **write** statement would actually be the following:

```
write(result);          (Output> "HI there ag")
```

See also: copy, comp, str, char

3.3.3.5 CLOSE

Purpose: To close a file that has been opened by the **open** command.

Format: **close** (*filevar*);

Description:

The **close** command's purpose is to close a file that has been opened by the **open** command. When the file is closed, the file that is being written to is updated and final disk "housekeeping" is taken care of. Failure to close a file that has been opened, before the end of the program, may result in the loss of data in the file.

The parameter to this function is a variable of the type file. No other type of variable can be used or will be accepted. An example of the use of the **close** statement is as follows:

```
file textvar;                    (declaration of the file variable "textvar")

open ("test.txt",textvar,w);     (opens "test.txt" for writing)

                (various writing to the file takes place...)

close (textvar);                 (file "test.txt" is closed)
```

Once a file has been closed, it can neither be written to nor read from. To perform any of those functions requires the reopening of the file.

See also: open, read, write, file

3.3.3.6 CLS

Purpose: To clear the screen.

Format: **cls**;

Description:

When executed, the **cls** function clears the computer screen and puts the cursor in the upper left-hand window. If the NCP environment is being used, the screen consists of the area within the double-lined border.

If the program was executed at the command line level, then the screen refers to the entire computer screen.

See also: gotoxy

3.3.3.7 COMP

Purpose: Compares two strings or string variables.

Format: **comp** (*string* | *var*, *string* | *var*);

Description:

The **comp** function compares two strings or string variables and returns a value indicating whether they are equal or greater than or less than each other.

The returned value is determined as follows:

Comparison	Return Value
string 1 < string 2	<0
string 1 > string 2	>0
string 1 = string 2	0

An example of the function's use is as follows:

```
comp  ("chair","chairs");  (Result= <0)
                    ^
comp  ("hello","hello");          (Result=  0)

comp  ("jello","hello");          (Result= >0)
       ^          ^
```

The comparison is done one letter at a time until a difference is detected or the end of the string is found. In a case such as the first example, where one string is shorter than the other, the long string is considered greater than the shorter.

See also: char, cat, copy

3.3.3.8 COPY

Purpose: Copies a string into a string variable.

Format: **copy** (*variable, string* | *variable*);

Description:

The **copy** function allows the copying of a string into a character array variable. The source must be a literal string or a variable containing a string. The destination must be a string variable. Any other configuration will result in an error.

An example of this function is as follows:

```
char a [10];  ( 'a' contains the empty string "")
char b [10];  ( 'b'  "    "  "   "   ")

copy (a,"Hello");    (Copy from a literal string)
copy (b,a);          (Copy from 'a' to 'b')

write(a);            (Output:   "Hello")
write(b);            (Output:       "Hello")
```

If the destination string contains any data, that data will be lost in the copying process. Care must be taken in observing the length of the strings, as if the size of the destination string is too small to hold the source string.

See also: char, comp, cat

3.3.3.9 DATA_READY

Purpose: Returns a non-zero value if there is data in the communications input buffer.

Format: **data_ready**

Description:

The **data_ready** function returns a non-zero value if the communications input or receive buffer is holding data. It provides the programmer the means to detect the reception of data by the serial communication port, other than polling. This function may be used in an assignment statement or in the decision process in a flow control statement such as **if/then/else**, or **while/wend**.

See also: if/then/else, while/wend

3.3.3.10 FOR/NEXT

Purpose: Provides the ability to perform a program loop.

Format: **for** (*variable* = *expr* to *expr* [**step** *expr*])
program statements ...
next

Description:

The **for/next** statement provides the ability to perform iterations in a program. The loop exists between the **for** and the **next** keywords and is executed a number of times determined by the (xxxx = xxx to xxx) statement.

The variable in the loop control statement holds the current number of the iteration. It is an integer and can be used like any other variable inside the **for/next** loop. When the loop exits, it will have a value equal to the second expression plus one iteration.

The expressions may be of any integer value and can cause the loop to go in either direction, i.e. count up or count down.

The optional **step** function allows the count variable to be incremented in steps greater than or less than one. The default step is 1.

Examples of the **for/next** statement are as follows:

```
int x;
```

```
for (x = 1 to 10)                    (Output: 1 2 3 4 5 6 7 8 9 10)
      write(x);
next


for (x = 1 to 10 step 2)       (Output: 1 3 5 7 9)
      write(x);
next


for (x = 10 to 1 step -1)      (Output: 10 9 8 7 6 5 4 3 2 1)
      write(x);
next
```

For/next loops may be nested inside each other or in other program statements, such as **while/wend** or **if/then/else**. Note: the maximum depth of nested statements of any type is 64.

See also: if/then/else, while/wend

3.3.3.11 GET COMM

Purpose: Allows access to the serial communication port parameters.

Format: **get comm** (*var*, *var*, *var*, *var*, *var*);

Description:

The serial communication port's operation is specified by five main parameters: Port number, baud rate, parity, number of data bits and number of stop bits. The **get comm** function retrieves of the values of these parameters. The values are stored in the variables that are specified in the calling of the function.

Port Number: The port number is the number of the serial communications port that is to be used for the transmission of data to the NIUs. This number may be either 0 or 1 for the number 1 serial port or the number 2 port. Many computers allow 4 or more serial ports to be installed within them. The NCP program only provides support for the first two ports.

Baud Rate: The baud rate is the speed of data transmission and reception through the serial port. The values that can be used here are as follows: 110, 150, 300, 600, 1200, 2400, 4800 or 9600.

Parity: The parity can be one of the following three choices: None, Odd or Even. This choices represented, respectively, by the values 0, 1 or 2.

Data Bits: The number of data bits can be either 7 or 8.

Stop Bits: The number of stop bits can be either 1 or 2.

Parameter	Values
Port Number	
Port #1	0
Port #2	1
Baud Rate	110, 150, 300, 600
	1200, 2400, 4800, 9600
Parity	
None	0
Odd	1
Even	2
Data Bits	7, 8
Stop Bits	1, 2

See also: set comm

3.3.3.12 GET DATE

Purpose: Reads and returns the date from the computer's system clock.

Format: **get date** (*var*, *var*, *var*);

Description:

The function **get date** retrieves the date stored in the computer's system clock. The format of the date that is returned is month, day and year, and the values are integers.

```
(This example assumes that the date is Jan 1, 1980)

int month, day, year;

get date (month,day,year);

write (month,day,year); (Output: 1 1 1980)
```

This function only retrieves the value; it does not allow the setting of the date. That requires the **set date** function.

See also: set date

3.3.3.13 GET TIME

Purpose: Reads and returns the time from the computer's system clock.

Format: **get time** (*var*, *var*, *var*);

Description:

Similar to the **get date** function, **get time** returns the time from the system clock and stores it in the variables specified in the calling of the

function. The values returned are, respectively, hours, minutes and seconds. They are in a 24 hour clock format.

(This example assumes that the time is 1:15am and 10 seconds)

```
int hours, minutes, seconds;

get time (hours,minutes,seconds);

write(hours,minutes,seconds);          (Output: 1 15 10)
```

This function only retrieves the time. To set the system clock's time, the **set time** function must be used.

See also: set time

3.3.3.14 GOTOXY

Purpose: Moves the cursor to a new location on the screen.

Format: **gotoxy** (*expr*, *expr*);

Description:

The **gotoxy** function moves the cursor to a new location on the screen specified by the given coordinates. The coordinates specified in the function are, respectively, the X-coordinate and the Y-coordinate. They are integers and should not exceed the width and height of the screen.

The screen size varies, depending on the circumstances of the execution of the program containing the **gotoxy** function. In the NCP development environment, the screen on which the program output appears has two sizes. Outside the NPC environment, there is the full screen dimensions available. The following table shows the sizes of the various screens.

	Width	Height
Small screen (with help menu)	78	12
Large screen (w/out help menu)	78	21
Full screen (outside NCP env.)	80	25

Table 3 View Screen Sizes

Examples of the **gotoxy** function show its use:

```
gotoxy (10,15);          (moves cursor to column 10, row 15)

gotoxy (4+5,ydist); (moves cursor to column 9, row
                         specified by the variable "ydist")
```

If a value for either coordinate exceeds the current size of the screen, then the function is not executed.

3.3.3.15 IF/THEN/ELSE

Purpose: The if/then/else statement provides flow control and decision making.

Format: **if** (*expr*)
then *statement*;
[**else** *statement*;]

Description:

The **if/then/else** statement allows decision making and flow control in the program. The true/false (TF) expression determines which of the two statements will be executed. If the TF expression is true, the **then** statement will be executed. If the expression is false, the **else** statement will be executed. In no case will both statements execute.

The TF expression is any numeric expression or value returned from a function. If the numeric expression is a non-zero value, then the

expression is true. It is false if the expression equates to a zero. The conditional expression is usually obtained by using logical operators, such as <, >, ==, and, or, and similar operators.

The **then/else** keywords are followed by a single statement. Multiple statements will produce an error. The exceptions to this rule are loops, such as **for/next** and **while/wend,** and blocks, such as **begin/end**. In these cases, the entire loop or block and all statements contained within are considered to be one statement. One other exception is another **if/then/else** statement. The following examples show the **if** statement and the use of loops and blocks in it:

(A simple **if** statement with the optional **else**)

```
if (x=1)
        then  y=y+1;
        else  y=y-1;
```

(A for loop is executed off the **then** statement)
(The optional **else** statement is not used here)

```
if (x=1)
        then for (y=1 to 10)
                        write(x,y);
                        z=z+1;
             next
```

(Nested **if/then** statements)

```
if (x=1)
        then if (y=2)
                        then  write(y);
                        else  write(x);
        else z=z+1;
```

An **if/then/else** or loop statement used inside another **if/then/else** statement is called nesting. In the case of the **else** statements, the **else** is associated with the last **if** that has no an **else**. The following section of code illustrates this:

```
if (x=1)
        then if (y=3)
                        then z=1;
                        else z=2;
```

The **else** statement is associated with the "if (y=3)" statement, not the "if (x=1)" statement, as the indentation shows. This is not as clear if the arrangement of the code is changed:

```
if (x=1)
        then if (y=3)
                        then z=1;
        else z=2;
```

This example is exactly the same as the previous one. The **else** statement is still associated with the "if (y=3)" statement. To associate the **else** with the "if (x=1)" statement, the code would have to be written as follows:

```
if (x=1)
        then if (y=3)
                        then z=1;
                        else;
        else z=2;
```

The maximum depth to which **if/then/else** statements, and the loops may be nested is 64 deep.

See also: while/wend, for/next, begin/end

3.3.3.16 KEYPRESSED

Purpose: Returns a non-zero value if a keystroke has been detected and a character is held in the keyboard input buffer.

Format: **keypressed**

Description:

The keypressed function returns a non-zero value if a key on the keyboard has been pressed and the keystroke is being held in the keyboard input buffer. The keystroke may then be retrieved by a **read** statement.

The value returned by the **keypressed** function is an integer and may be used in assignment statements or the control expression of an **if** or **while** statement.

See also: if/then/else, while/wend, data_ready

3.3.3.17 OPEN

Purpose: Opens a file for writing, reading or appending.

Format: **open** (*filename*, *filevar*, **read** | **write** | **append**);

Description:

The **open** function opens a file for reading, writing or appending. The filename can be any legal name, but the filevar must be a variable of the type file. The filevar is used when addressing the file for writing, reading or closing files. An example of its use shows the point:

```
file file1;

open("testfile.txt", file1, write);
```

```
write(file1,"test  string");
close(file1);
```

When a file is opened for reading, the file pointer points to the beginning of the file. Any reading will take place at the start of the file and proceed to the end. Opening a file for writing resets the file pointer to the beginning and also erases the file. Appending sets the file pointer to the end of the file, where writing takes place. In appending, the file is not erased.

To close a file, the **close** function is used.

See also: close, read, write, file

3.3.3.18 READ

Purpose: Provides input from the keyboard or from a file.

Format: **read** ([*filevar*,] *var*, ...);

Description:

The **read** function provides input from the keyboard or from a file. Data read in using this function is stored in the variables specified in the calling of the **read** function.

The **read** function will read either an integer or a character and can read any number of variables in one statement. The **read** statement by default reads from the keyboard. When entering data from the keyboard, the backspace key will delete a previous character. No other editing key will work, including the delete and the insert key. Input from the keyboard is terminated by a carriage return.

Reading from a file is done when the optional file variable is used at the beginning of the **read** statement. All data read, then, is read from the specified file. The following examples show how the **read** statement works:

```
int a,b,c;
```

```
char r,s,t;
file file1;

read(a,b,r,s);          (Typed: 123 55 A <CR>)
write(a);               (Output: 123)
write(b);               (Output: 55)
write(r);               (Output: " " (a space))
write(s);               (Output: "A" )

open("testfile.txt",file1,read);

read(file1,t,c);        (File Data: 123)
write(t);               (Output: "1")
write(c)                (Output: 23)
```

In the first code section, the first two **read** statements read the integers 123 and 55. The next **read** statement reads a character. Since a space is a character, that is what is read, followed by the "A".

In the second code section, the first **read** statement reads a character from the file "testfile.txt". It reads a "1," since that is the first character in the file. The second **read** statement reads the integer 23.

See also: close, open, write, file

3.3.3.19 RECV

Purpose: Reads a string from the serial communications input buffer.

Format: **recv** (*var*);

Description:

The **recv** function is very similar to the **recvb** function, except that it reads a string from the serial communications input buffer.

The reading of the string itself is also special. In most cases the NIUs that are sending information to the PC send the data surrounded by square brackets "[]." For example, the NIU with the address of 07 responds to the POLL command by returning its address:

[07 00]

Since the data from the NIU is delimited by the square brackets, any data outside the brackets may be ignored. The **recv** function reads data from the input buffer and ignores everything until it reads an opening bracket. It then copies the data into the string variable given in the parameters until it reads a closing bracket. This allows junk characters from noise or other sources to be ignored.

Care must be taken to ensure plenty of space for the string that is to be read, or data may be lost.

```
char a[20];

recv(a);          (assume that the characters "aad[1234]" )
                  (are in the receive buffer)

write(a);         (Output: "1234");
```

See also: recvb, send, sendb

3.3.3.20 RECVB

Purpose: Reads one character from the serial communications port receive buffer.

Format: **recvb** (*var*);

Description:

The serial communications port has an input buffer to store incoming data. This data may be retrieved from the buffer, one character at

a time, by the **recvb** function. The character that it reads is stored in the variable specified in the parameters of the **recvb** function.

In the following example, the operation of the **recb** function is shown:

```
(This assumes that the serial input buffer contains)
(the characters: "[12 34]" )

char  bufchar;

recvb(bufchar);
write(bufchar);          (Output: "[")

recvb(bufchar);
write(bufchar);          (Output: "1")
```

The variable used to store the character the **recvb** function returns may be either a character, or an integer.

See also: recv, send, sendb

3.3.3.21 SEND

Purpose: Transmits a command packet out on the serial communications port to an NIU and the network.

Format: **send** (*string* | *expr*, *string* | *expr*, *xcntl* [,*data*]);

Description:

The **send** function transmits a command packet out through the serial communications port to the NIU connected to the PC. The function takes as input the command to be sent, the address of the NIU for which the command is destined, execution control and up to 5 bytes of optional data.

Command: The command may be specified in several ways. The first is by the number of the command. The number may be written in decimal or hexadecimal form. The commands

```
send (10,10,I.);                    send(10,$A,I.);
```

are identical, and both send an order to the NIU to execute command number 10. The '$' indicates that the following number is in hexidecimal form instead of decimal.

This command may also be written using a variable instead of an explicit number. Thus the command

```
x=10;
send(10,x,I.);
```

is identical to the two previous commands.

A third way to specify the command is to reference it by its mnemonic. In the above examples, command number 10 is the poll command, which has the mnemonic POLL. The mnemonic may be referenced by a string, a variable containing a string or as a variable itself. The following commands are all identical:

```
send(10,"POLL",I.);
send(10,cmndstr,I.);                (where  cmndstr="POLL")
send(10,POLL,I.);
```

The mnemonics may be upper or lower case when using strings; they must be uppercase when using the mnemonic as variable.

Destination: The destination is the address of the NIU for which the command packet is destined. The destination address can be referenced (like the command above) by number (decimal or hexidecimal) or by variable. The following examples are all identical:

```
send(10,10,I.);
send($A,10,I.);
send(x,10,I.);              (Assuming that x=10)
```

The destination address may also be referenced by the name of the NIU specified in the NIU List. The name of an NIU may be entered in the NIU List. This assigns that name an address, and using that name in the **send** command will allow access to that address. The following example demonstrates this:

```
(This example assumes that the NIU with the address of 10)
(has been assigned the name "LAB".)

send("LAB",10,I.);
send(deststr,10,I.);        (Assumes that deststr = "LAB")
send(LAB,10,I.);
```

The previous examples are all functionally identical. In all respects, the use of the destination field of the **send** command is the same as the command field.

There is one special destination address, and that is the address of 00 (ALL). This indicates that all of the NIUs on the network are to receive the task specified in the **send** command.

Execution Control: The execution control refers to how the command will be executed by the NIU. There are three pre-execution commands, two post-execution commands and an optional echo command. The following execution controls must be used in order in the **send** command. They may be referenced as strings or as variables, although unlike command and destination fields above, they have no intrinsic values.

Pre-execution commands

Queued Task -Q

This command tells the NIU to place the task on the end of the task queue. All tasks before it on the queue will be executed in order before the execution of this task.

Immediate Task - I

The immediate task is to be executed immediately. This has priority over all tasks currently on the queue, as well as those currently being executed. Tasks that are currently running are suspended until after the Immediate task is through being executed. An immediate command is aborted if the NIU receives another immediate task.

Synchronized Task - S

The synchronized task is treated like a Queued task, in that it is placed at the end of the task queue, where it will wait until its turn arrives. When it is its turn to be executed, the task prevents its execution until the NIU receives a synchronizing signal. The synchronizing signal is an immediate task with the task number of 3 (SYNC). If task number 3 is used with the universal destination 00 (ALL), then multiple NIUs may be started at the same time.

Post-execution commands

Single Execution - .

The "." character indicates to the NIU that the command is to be executed only once. When the task is through running, it is discarded and lost.

Re-Queued Task - +

The "+" character tells the NIU to put the task back at the end of the task queue when the command is done running. In this way, a number of commands may be executed over and over again. Immediate tasks may not be requeued.

Optional Echo Command - /

The "/" character tells the NIU to echo the command it has just received. This allows verification that the command was received and received correctly. The command echoes without the '/' character, and it is surrounded by square brackets "[]."

The above commands must appear in order without spaces or the like separating them. Only one pre-execution command and one post-execution command may be used. Uppercase or lowercase may be used. Some examples of the **send** function are as follows:

```
send(10,10,Q+);     (Queued task that is requeued when done)
send(10,10,I.);     (Immediate task, with one execution)
send(10,10,S+);     (Synchronized task, requeued when done)
send(10,10,Q./)     (Queued task, one execution with echo)
```

The execution control may be referenced by a string, instead of by explicit characters.

```
send(10,10,"Q+");
send(10,10,xstr);          (Assumes xstr= "Q+")
```

Optional Data Field: Data may be sent to an NIU along with a task. Some tasks require data to properly perform operation. The data may be up to 5 bytes long, and may be written in decimal or hexidecimal form, or variables may be used. For example,

```
send(10,10,Q.,12,255,32,54,10);
```

sends the five pieces of data along with the command to the NIU. The data must be separated by a comma and nothing else. Not all five bytes need be used.

See also: recv, recvb, Section 3.2 : Task Library and Node List

3.3.3.22 SENDB

Purpose: Transmits one character out on the serial communications port to an NIU and the network.

Format: **sendb** (*var* | *character*);

Description:

The **sendb** function transmits a single character out from the serial communications port to the local NIU and the network. This simply allows the user some freedom in what he transmitts instead of being confined to using the **send** command.

In the following example, the operation of the **sendb** function is shown:

```
char  sendchar;

sendchar='A';
sendb(sendchar);          (Output: 'A');
```

The variable used to store the character that is to be sent using the **sendb** command may be a character or an integer.

See also: recv, recvb, send

3.3.3.23 SET COMM

Purpose: Sets the serial communications port parameters.

Format: **set comm** (*expr*, *expr*, *expr*, *expr*, *expr*);

Description:

The serial communications port operation is specified by five main parameters: Port number, baud rate, parity, number of data bits and number of stop bits. The **set comm** function allows the setting of these parameters.

Serial Port Parameters

The five serial port parameters that can be set from the **set comm** function are Port Number, Baud Rate, Parity, number of Data Bits and number of Stop Bits. For a description of these parameters, see the **get comm** function. The following table shows each of the parameters, and the values they may have:

Parameter	Values
Port Number	
Port #1	0
Port #2	1
Baud Rate	110, 150, 300, 600
	1200, 2400, 4800, 9600
Parity	
None	0
Odd	1
Even	2
Data Bits	7, 8
Stop Bits	1, 2

The function may take either variables or explicit numbers as parameters.

See also: get comm

3.3.3.24 SET DATE

Purpose: Sets the calendar date on the computer's system clock.

Format: **set date** (*expr*, *expr*, *expr*);

Description:

The function **set date** takes three parameters, month, day and year, and sets the date on the computer's system clock. The format for the date is

month, day, and year, and the values are integers. The following example demonstrates the use of **set date**:

(This example assumes that the date is currently Jan 1, 1980)
(and that the date will be set for June 1, 1988)

```
int month, day, year;

get date (month,day,year);
write (month,day,year); (Output: 1 1 1980)

set date(6,1,1988);

get date (month,day,year);
write (month,day,year); (Output: 6 1 1988)
```

In the above example, the **get date** function is used to retrieve the current date from the system clock, while the **set date** is used to set the new date.

See also: get date

3.3.3.25 SET TIME

Purpose: Sets the time on the computer's system clock.

Format: **set time** (*expr*, *expr*, *expr*);

Description:

Similar to the **set date** function, **set time** allows the setting of the time on the system clock. The parameters to the **set time** function are integers in the format hour, minute, sec in a 24 hour clock. This means that if the clock is to be set to 1:00 pm, the hour parameter would have to be 13 instead of 1.

(This example assumes that the current time is 1:15 AM, and is)
(going to be set to 2:15 PM)

```
int hours, minutes, seconds;

get time (hours,minutes,seconds);
write(hours,minutes,seconds);          (Output: 1 15 10)

set time (14,15,0);          (14 hours = 2pm)

get time(hours,minutes,seconds);
write(hours,minutes,seconds);          (Output: 14 15 0)
```

In the above example, the **get time** function is used to retrieve the time so that it may be printed, and the **set time** is used to actually set the time.

See also: get time

3.3.3.26 STR

Purpose: Converts an integer to a string.

Format: **str** (*var*, *expr*);

Description:

The **str** function takes an integer and a string variable as input, converts the integer into a string and copies it into the string variable. The integer will be written as a decimal number.

The following example show the function of the **str** command.

```
int x;
char a[10];

x=3;
str (a,x);
write(a);                    (Output: "3")

str(a,123);
write(a);                    (Output: "123")
```

Care must be taken to ensure that the variable is large enough to hold the string, or data may be lost.

See also: val

3.3.3.27 VAL

Purpose: Converts a string to an integer.

Format: **val** (*string* | *variable*);

Description:

The **val** function takes a string, or a variable containing a string, and converts the string to an integer. This function assumes that the input string contains an integer in a string form. The string must be in the following format to successfully be converted to an integer:

white space sign digits

The beginning of the string may have any amount of white space (i.e. tabs or spaces). It then may have a sign (+/-), followed directly by a series of digits representing the number to be converted. The function is terminated

when an unrecognized character is read. The following examples illustrate this:

```
int x;

x = val("   -102");          (x = -102)
x = val("124a");                  (x = 124, the function ends at 'a')
x= val("x12");                    (x = 0, the function ends at 'x')

char a[10];
int x;
copy (a,"+124");

x = val(a);                   (x = 124)
```

See also: str

3.3.3.28 WAIT

Purpose: Suspends the execution of the program for a specified number of seconds.

Format: **wait** (*expr*);

Description:

The **wait** function delays the execution of the currently running program for a number of seconds specified in the function parameters. Once the delay is over, the program resumes execution.

See also: wait until

3.3.3.29 WAIT UNTIL

Purpose: Suspends the execution of the program until a certain time or date is reached.

Format: **wait until** (*expr*, *expr*, *expr*);

Description:

The **wait until** function is very similar to the **wait** function, except that instead of waiting a set number of seconds, it waits until a time or date specified in the parameters has been reached. It then allows the execution of the program to continue.

The parameter will take either a time or a date. The **wait until** function determines whether it is supposed to wait for a specified time or a specifed date by examining the parameters it receives. If the function is until wait to a specific time, then the parameters will contain a time in the format hours, minutes, seconds. The time must be in a 24 hour clock format. An example of the **wait until** function in the time mode demonstrates this:

```
wait until (18,0,10);        (Wait until 6:00pm and 10 seconds)

wait until (18,70,10);
                  ^ (WRONG! Only 60 minutes in an hour)
```

If the function is in the date mode, then the format is month, day, year. The month and day parameters must be integers, but otherwise are written normally. The year parameter must be written completely out, i.e. it should not be abbreviated. It also must be greater than or equal to 1980 to work properly. The following examples show this:

```
wait until (1,1,1988);       (Wait until Jan 1, 1988)

wait until (1,32,1988);      (WRONG! Maximum of 31 days)

wait until (1,1,88); (WRONG! The year is less than 1980)
```

Little error checking is done by the computer on the **wait until** parameters. If an invalid date is entered, it may cause an infinite delay in the function. If this happens, the program may be aborted by pressing the **Ctrl-Break** key combination.

See also: wait

3.3.3.30 WHILE/WEND

Purpose: Provides a program loop based on a conditional statement.

Format: **while** (*expr*)
program statements ...
wend

Description:

The **while/wend** statement provides the ability to do repetitive loops based on a condition. The **while** statement repeats the loop, executing all of the statements between the **while** and **wend** keywords, as long as the conditional expression remains true. After each execution of the loop, the expression is reevaluated, and if the expression is false, the loop is exited. Any number of program statements may be in the loop.

The conditional expression may be any numeric expression that evaluates to an integer. If the number is non-zero, then the statement is true; otherwise, if it is a zero, it is false. The expression usually consists of logical expressions, such as <,>,==, and, or, etc.

The following examples illustrate the use of the **while** statement:

```
int x,y;

x=1;
y=0;
```

```
while (x<10)
      write(x);              (Output: 1 2 3 4 5 6 7 8 9)
      x=x+1;
wend

while (x<10 and y==0)
      write(x);
      y=x/5;                 (Output: 1 2 3 4 5)
      x=x+1;
wend

while (1)
      write("Loop  ");       (Output: "Loop Loop Loop Loop ...")
wend                                  (This is an infinite loop)
```

See Also: for/next

3.3.3.31 WRITE

Purpose: Outputs data to the screen or to a file.

Format: **write** ([*filevar*,] *expr* [, *expr*] ...);

Description:

The **write** function allows the output of data either to the computer's screen or to a file. If the data is to be written to the screen, then the optional filevar is excluded from the statement. If the filevar is included, then the file that the filevar is associated with will received the data.

The data to be written, is specified in the parameters of the function. Variables may be used or values or strings may be explicitly written. Any number of expressions may be included in the **write** statement. Each

expression or variable must be separated by a comma. Explicit strings must have beginning and ending quotes.

Writing to a file is done when the optional file variable is used at the beginning of the **write** statement. If that is done, then all data in the **write** statement is written to the file in the order in which it appears in the function. The filevar is obtained from the open statement and is of the type file.

There are several special characters that can be used in the **write** function. These characters are formatting characters and include the carriage return, line feed and backspace. These characters are written in a string using the '\' (backslash) as an escape character. The following table shows the escape sequences used in this program:

Sequence	Value	Char	Function
\a	07	BEL	Sounds bell
\b	08	BS	Writes backspace
\f	12	FF	Writes form feed
\n	10	LF	Writes line feed
\r	13	CR	Writes a carriage return
\t	09	TAB	Writes a tab
\\	92	\	Writes a backslash
\"	34	"	Writes a double quote
\any char	??	??	Writes character

Table 4 Escape Codes

The backslash escape sequence allows the writing of itself and of the double quote, which normally would not be allowed. The '\r' and the '\n' both translate automatically a a carriage return-linefeed operation. In all respects, they are identical.

The following examples show how the **write** statement works:

```
int  a,b,c;
char  r,s,t;
```

```
file file1;

a=5;
b=10;

write(a);                    (Output: 5)
write(a+b);                  (Output: 15)

r='A';
write(r);                    (Output: A);

write("There are ",a," ",r,"'s in Aaaaa");

      (Output: "There are 5 A's in Aaaaa")

open("testfile.txt",file1,read);

write(file1,a);                    (Output: 5)
write(file1,"Text")          (Output: Text)

write("This is a test of the carriage return. \nSee?");

      (Output: "This is a test of the carriage return.
                See?")
```

See also: close, open, read, file

3.4 Error Messages

The following is a short list of error messages produced by the NCP interpreter.

Program File "xxxxxxxx.xxx" not found

The Interpreter could not find the specified program file on the disk drive. Checking the name and location of the file is suggested.

Duplicate Definition on line: xxx

A variable of the same name was defined more than once in the variable declaration section.

Undefined Variable on line: xxx

An attempt was made to use a variable that was not declared.

Syntax Error on line: xxx

An error in the the format of the program statements was made. Check the location of the error and the manual to determine the specific error.

Illegal Value xxx in function Comm on line: xxx

A incorrect value was used in a Comm function (**Send** or **Get**).

Control Break - User Interrupt on line: xxx

The program was interrupted at the specified line.

"xxxxxx" not found in function SEND on line: xxx

The name of the task or destination NIU was not found in the lists. Check the name used with the Task Library list or the NIU list.

3.5 Lex and YACC Theory

The NCP interpreter is a combination of two major programs, the lexical analyzer and the parser. The lexical analyzer inputs an NCP program text file, matching keywords and other letter combinations with a set of rules, and then outputs representative tokens. The tokens are then matched up by the parser with rules that were written for it. When the incoming tokens have matched certain rules, the parser executes the functions that run the interpreter.

While the support functions do most of the work in running the interpreter, it is the lexical analyzer and the parser that give it structure. These programs are generated from a set of fairly simple rules provided by the user. These rules and how they work that will be covered briefly in the next section.

3.5.1 Lexical Analyzer

The lexical analyzer processes a character input stream by attempting to match the input with a set of string expressions. When a match is made, sections of program code are executed. Usually, as in the case of the NCP interpreter, a number (called a token) is returned. The token represents the matched string expression and is used later in the parser.

The lexical analyzer used in the NCP interpreter was written using the Lex program. The Lex program takes a set of rules, usually string expressions, and generates the lexical analyzer. The rules used by Lex are generally followed by a line of program code. The program code usually returns a token. Other times, when a value is needed, the token is returned as well as the string or value that was matched. Tokens are given values that are greater than 256 to avoid conflict with any members of the character set.

The structure of Lex is simple. The format for a Lex grammar consists of three sections:

```
definitions
%%
rules
%%
user's  functions
```

The definitions section holds any *include* statements, function predeclarations or variable definitions. The rules sections contains the regular expressions used by Lex to analyze the input. Functions that support the Lex rules are placed in the last section.

The rules used in matching the input to the expressions are simple. The rules are checked in order, from beginning to end, until a match is made or the last rule is checked. Input that matches an expression allows the execution of program code included with the expression. The most basic expression is the simple string, as the following example shows:

```
"int"       return(INT);
```

When the analyzer recognizes the string "int" in the input, it returns the value associated with the token INT. As the analyzer works through the input sequentially, it will not match the expression if it is contained inside a larger string. For example, there is the case of "to" and "gotoxy":

```
"to"            return(TO);
"gotoxy"        return(GOTOXY);
```

If the input was "gotoxy (1,1)," the analyzer would not match the "to" expression, even though "gotoxy" contains a "to." This is because it is trying to match the "g" of the "gotoxy" first, and thus would skip over the "to" in favor of the "gotoxy." If the rules were as follows,

```
"go"            return(GO);
"to"            return(TO);
"gotoxy"        return(GOTOXY);
```

then the analyzer would match the "go" with the first rule, the "to" with the second, and would then try to match the "xy." This shows that care must be taken in writing the Lex rules.

There are a number of other ways to write the string expressions. Using square brackets, matches may be made to a class of characters. The brackets may contain a number of characters, but the analyzer will only match one character in the brackets. For example,

```
[ \t]          ;
```

matches spaces (the first character) or tabs (the backslash is the C escape character), but not both. When the expression is matched, no action is taken. This rule simply strips out of the text file all of the spaces and tabs (i.e. white space).

Another example of this technique is the rule to recognize numbers:

```
[0-9]+         {yylval.num = atoi (yytext); return(CONST);}
```

The "0-9" is shorthand for "[0123456789]". It specifies a range of characters which in this case is all of the numbers. The "+" symbol is a command to the analyzer to match one or more of the characters in the brackets. This allows the matching of multiple characters in a single set of brackets. In the above example, the expression matches up with a number of any size. When the match is made, the number (stored in the variable yytext) is converted to an integer and assigned to a variable for passing to the parser. A token is also returned, indicating the nature of the output.

There are a number of other types of expressions, using a variety of operators, that can be used in a Lex grammer. More information on Lex and the Lex grammer may be found in [LESK 75].

3.5.2 The YACC parser

The parser gives the structure to the NCP interpreter. The heart of the parser is a set of rules that tells what order program statements may come in. It takes as input the tokens generated by the lexical analyzer. The parser then attempts to organize the tokens with its rules or grammar. When a rule is matched, added program code is executed, which performs the operations of the interpreter.

The parser was written using the YACC program. YACC (Yet Another Compiler-Compiler) takes a set of structure rules (or grammar rules) as input, and creates the source code for a parser. The parser takes the tokens, representing the information to be parsed as input. Parsers generated by YACC automatically call the lexical analyzer created by Lex. As the grammar rules are fairly simple, creation of a parser can be quite easy.

The format for the grammar rules for YACC consist of the following parts:

```
declarations
%%
rules
%%
user's  programs
```

The declarations section holds any variable declarations, function predefinitions, declaration of the tokens used and preceidence that the rules have. The rules section holds the parser rules, which in turn can call the functions in the third section.

A rule consists of an identifier, followed by tokens and identifiers for other rules. If an identifier has more than one rule, they are separated by vertical bars '|'. Rules are terminated by semicolons. An example of a rule is as follows:

```
starts : BEGIN vars stmts END ;
```

In the above example, tokens are in uppercase, and identifiers (or terminals) are in lowercase. The "starts" terminal is the first used in the NCP interpreter, and as such, has the first rule to be executed. The rule attached to the "starts" terminal states that it is looking for the token BEGIN. Since this is the first rule, the first token returned from the lexical analyzer must be the token BEGIN. If it is not the first token returned, then an error is generated, and the program stops.

Once the parser has matched the BEGIN token, it jumps to the "vars" terminal to execute other rules there. Once those rules are matched, the parser jumps the "stmts" terminal to match those rules, after which it looks for an END token. This ability to jump from rule to rule allows complex parsers to be written easily.

The following example shows the use of multiple rules and program statements. The example creates the part of the parser that handles simple arithmetic expressions.

```
stmts: NAME "=" expr          {assign_value($1,$3);}
       ;
expr  :      "(" expr ")"          {$$ = $2;}
      |      expr "-" expr         {$$ = $1 - $3;}
      |      expr "+" expr         {$$ = $1 + $3;}
      |      expr "*" expr         {$$ = $1 * $3;}
      |      expr "/" expr         {$$ = $1 / $3;}
      |      CONST                 {$$ = $1;}
      |      NAME                  {$$ = get_value($1);}
      ;
```

The rule "stmts" looks for the token NAME, followed by an equals sign. When it finds that, it jumps to the terminal "expr." The "expr" terminal has several rules, and the one that it picks is dependent on the input. Many times the parser must input several tokens before making a decision. If the input was "1 + 2" (the tokens would be "CONST + CONST"), it would pick the third rule. The third rule looks for an expression followed by a '+' sign, followed by another expression. Once the parser knows which rule it needs, it will execute that rule.

The third rule makes the parser jump to the "expr" terminal and execute a rule there. The 6th rule matches up with the first part of the input, and since that rule has been recognized, the "$$=$1;" statement is executed. The $$ is a variable that passes values back to previous rules, and the $1 gets values from parts of the rule. In this case $1 gets the value sent along from the lexical analyzer and passes it back to the previous rule.

After the first part has been matched, the third rule matches the '+' and then jumps again to the "expr" terminal to match the last half. After the third rule has been recognized, the "$$= $1 + $3" statement adds the values from the first and third parts of the rule and passes the result back to the previous "NAME = expr" rule. That rule executes the "assign_value" function to put the value in the variable specified by NAME.

This is only part of what YACC can do. More information on how to use this program can be found in [JOHN 75].

3.5.3 Changing and Adding to the Interpreter

If any changes are to be made to the interpreter or the NCP program, the following suggestions will be helpful.

3.5.3.1 The NCP Program

The NCP program is written in Turbo C Version 2.0. If any changes are made to the program, it will have to be recompiled using Version 2.0 or greater, although V1.5 or greater should work. The size of the program precludes the use of the integrated development environment for linking purposes, and the compiling of some of the functions. This means that stand alone compiler and linker will have to be used. A make file with all of the correct dependencies should be with the source code provided with this program.

The files "globals.c" and "globals.h" hold the major variables and arrays that are used in most of the functions. "maxs.h" holds the constants that dictate the size of the task and node libraries. Increasing them both to 256 elements will exceed the 64k allowed for the data section. If larger libraries are needed, then a switch to the COMPACT memory model will have to be made (see the **Turbo C Manual**).

3.5.3.2 Lex

The lexical analyzer is generated from the rules called "lex.rul". It was done on a UNIX based machine, using the Lex program. The resultant C source code is compatible with Turbo C, although a few changes must be made before it can be used in the NCP program:

1. The name of the source code file generated by Lex is "yylex.c". The name used in the make file for the NCP program is "lex.c".

2. The name of the analyzer function is "yylex". The function must be renamed "yylex2" to accommodate a preprocesssing function.

3. The 10th line in the source code must be changed. This is to prevent the occasional writing of line feeds. The line

```
# define output(c) putc(c,yyout)
```

must be changed to

```
# define output(c) /* putc(c,yyout) */
```

3.5.3.3 YACC

The parser was generated from a file of rules called "yacc.rul", using the YACC program on a UNIX based machine. The parser rules must be compiled with YACC with the "-d" option to get the "y.tab.h" header file. The command is as follows:

```
yacc -d yacc.rul
```

The resultant source code is compatible with Turbo C if a few changes are made. The changes are as follows:

1. The parser source code is called "y.tab.c", and the header file is called "y.tab.h". These must be renamed, respectively, to "ytab.c" and "ytab.h".

2. An extra '#' character must be deleted. The '#' appears in the code in the following place:

```
Delete>     #
            # define YYFLAG -1000
            # define YYERROR goto yyerrlab
            # define YYACCEPT return(0)
            # define YYABORT return(1)
```

3. The line

```
# line xxx  "yacc.rul"
```

appears throughout the source code. The "xxx" characters are line numbers. All of these lines must be deleted for the source code to compile correctly.

3.5.4 Protocol from PC to NIU

The communication between the PC and the Network Interface Unit (NIU) is simple. Data is transmitted without the requirement of an acknowledgment. If data arrives garbled, it will not be retransmitted. There is, however, a format for the data transmitted and received.

All data transmitted to the NIU is in the following format:

{ NN P TT Q DD DD DD DD DD / }

{ } The braces indicate the beginning and ending of the data packet.

NN This is the number of the destination NIU. The destiation of 00 is a universal address and sends the packet to all of the NIUs on the network.

P This is the pre-execution character. It can have the following values:

: The task is to be put on the end of the queue for execution when all tasks before it have been executed.

! The task is to be executed immediately upon reception by the NIU. It goes before all tasks on the queue and suspends the task that is currently running.

? The task is placed on the queue to wait its turn. When the task is to be executed, the NIU will stop and wait for a synchronizing signal from the host. This signal is usually the immediate task SYNC.

TT This is the number of the task that the NIU is to execute.

Q This is the post-execution character. It tells the NIU what to do with the task once it has been completed. There are two options, which are as follows:

+ The task is to be requeued. The task is placed at the end of the queue for execution again.

. The task is to be discarded.

DD This is the optional data field. In some instances, data needs to be passed to the NIU. Up to 5 bytes of data may be passed in this manner. The data field is entirely optional and may be removed.

/ This is also an optional character and may be eliminated. It is used to verify the reception of a command packet by the NIU for which it is destined. If the "/" is added, the receiving NIU will echo the command packet with square brackets "[]" instead of braces, and without the "/" character.

All data received by the PC from an NIU are contained in square brackets. The contents of the return packet are dependent on what commands were sent to the NIU. There is no a specific format. For example, the POLL task asks an NIU for its address and the contents of its activity register. The response is as follows:

[07 00]

The first number is the address; the second, the contents of the activity register. If the command sent was MTOH (Memory TO Host data transfer), the response might be

[56]

which is data stored in the NIU.

In both cases, the data returned were different and in a different format.

4. Conclusions and Recommendations

4.0 Conclusions

The NCP program is a good tool for creating and running programs to control the Task*Master microcontrollers. It provides a number of useful development tools and has a reliable interpreter for running developed programs.

The use of the Lex and YACC programs for creating the lexical analyzer and the parser is one of the strong points of the system. It provides a standard, easy-to-understand format for changes to either function and takes away the difficulty of deciphering unknown code for future programmers.

The interpreter provides a quick reliable execution of network programs for controlling the microcontrollers. A number of easy-to-understand and useful functions complement the interpreter.

4.1 Recommendations

4.1.1 The NCP Program

The NCP program is far from complete. It does suffer from a sparseness in certain functions. The READ and WRITE statements lack formatting capability, and the WRITE statement would perform better if it were changed completely. A format more like that found in the C programming language would be far better.

The FILE functions could provide random access to files instead of simple sequential access.

More arithmetic functions would be useful, as well as improved logic functions.

More types of variables and a better way of storing and accessing them could be added. The program currently has only character and integer variables, with arrays for both. Floating point variables could be added with all of the necessary arithmetic functions. Unsigned and higher precision variables would also be useful. Support for multi-dimensional arrays is almost mandatory. This would provide the ability to have arrays of strings, for storage of names. The single dimension array that this current

version of the NCP program uses is far too limiting. Only an oversight allowed this to occur.

The access to the Task Library and the NIU list should be improved. Currently, both of the lists can be accessed to obtain the integer values for the tasks or NIU addresses. Access to the names of the tasks or NIUs should be added. This would permit the users to find out the name of the task or NIU when he has only its number. In this way, far more useful information could be provided to the user.

Subroutines and functions are two structures that should be added. The current version of the NCP program allows only the sequential execution of the user's program. Repetitive statements must be executed in a loop, or rewritten for each execution. Subroutines would provide a much easier way of doing repetitive commands, as well as making the code more readable. Implementing these structures would require completely reworking the way the interpreter reads its input, but the reward would be worth the work.

4.1.2 Task*Master and COLANs Future

The NCP program was designed to work with the Task*Master microcontroller. This includes the original Task*Master daisy-chain system, as well as the COLANs. However, the NCP program will not work with the COLANs without the creation of a interface function to handle the various command packet formats. This is due to the inconsistency in the PC to NIU protocol that was developed for COLAN I [ZHEN 86]. The inconsistancy is that the packets sent to the local node have a different format from those sent to remote nodes, even though to the PC there is not a difference in the two type of NIUs.

This problem needs to be corrected by coming up with a consistant and well thought out format for command packets that is the same no matter for which node it is bound. A consistent format for receiving data should also be decided on. Preferably, it should be one that includes the identification of the source of the data. This would greatly improve the ability of the controlling program to schedule events, based on the current statuses of the operating microcontrollers.

Bibliography

[BORL 88] Borland International, Turbo C Reference and User's Guide, Scotts Valley, CA, Borland Int., 1988.

[EUM 87] Eum, D. COLAN III, A Control-Oriented LAN Using CSMA/CD Protocol, unpublished master's thesis, Oregon State University, Corvallis, OR, 1987.

[GOFT 86] Gofton, P., Mastering Serial Communications, Alameda, CA, Sybex Inc., 1986.

[HERZ 87] Herzog, J.H., A Design Methodology for Distributed Microprocessors in Real-Time Control Applications, a paper submitted to The Second International Conference on Computers and Applications, Beijing, China, June 1987.

[HERZ 85] Herzog, J.H., TASK*MASTER Operating Manual, Oregon State University, Corvallis, OR, 1985.

[JOHN 75] Johnson, S.C., YACC: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ, 1975.

[KAO 87] Kao, S. Design of COLAN II, A Control Oriented Local Area Network, unpublished master's thesis, Oregon State University, Coravallis, OR, 1987.

[KERN 78] Kernighan, B.W., The C Programming Language, Englewood Cliffs, NJ, Prentice-Hall, 1978.

[LESK 75] Lesk, M.E., Lex - A Lexical Analyzer Generator, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, NJ, October 1975.

[NORT 85] Norton, P., The Peter Norton Programmer's Guide to the IBM PC, Washington, Microsoft Press, 1985.

[TANE 81] Tanenbaum, A. S., Computer Networks, Englewood Cliffs, NJ, Prentice-Hall Inc., 1981.

[THYE 88] Thye, Y., COLAN IV, A Network for Communications and Control, unpublished master's thesis, Oregon State University, Corvallis, OR, 1988.

[ZHEN 86] Zheng, Y., A Simple Local Area Network, COLAN (Control Oriented Local Area Network), unpublished master's thesis, Oregon State University, Corvallis, OR, 1986.

APPENDICES

A.0 Example Programs for the NCP Program

A Program to poll the NIUs on the network hooked to the PC.

{This program polls all of the NIUs on the network attached to the PC. It sends out the command of POLL to ALL of the nodes on the network. It then waits for replies, and prints out which nodes responded. Since the nodes won't respond in any particular order, the output will not be in any order.}

```
begin

char  data[10] ;
char   temp10];
int    address;

send(ALL, POLL,I.);          {Sends to all NIUs, the immediate task POLL}
write("A list of active units on the network\n");
while(not keypressed)
      recv(data);            {get data from serial port}
      temp[0]=data[0];       {put address into temp variable}
      temp[1]=data[1];
      temp[2]=0;

      address=val(temp);          {convert string to integer}
      write("Unit #",address," responded\n");
wend
write("Polling Ended");

end
```

A Program to Control Heating in a building

{This program is designed to monitor and control the heating in a building. Each Task*Master unit is assigned a room, and is assumed to be attached to a thermostat that provides it with the temperature. It is again assumed that this data is contained at Data Port #1. The use of the Task DPT1 or 8 bit Transfer to/from Data Port #1 is used here to transfer the data. This task is currently not implemented, but it will suffice for an example. The control of the heating is done by using the relay ports. These are controled by the task RELP or 4 bit transfer to the relay port.}

```
begin

int     temp,  number,delay;
int     x,y;
int     hour,minute,sec;
int     stations[30];
char    data [10];
file    filevar;

cls;
write("Temperature  Monitoring  Program\n");
write("Please enter number of Stations: ");
read  (number);
write("\nPlease  enter  Station  addresses");
for (x=0 to number-1)
       write("\nStation #",x," : ");
       read  (stations[x]);
next
write("\nPlease enter time between checks (minutes): ");
read(delay);

send(ALL,RSET,I.);                {reset the boards);

open  ("Tempdata.txt",filevar,write);     {open a file for writing data}
```

```
while (not keypressed);
      for (x=0 to number-1)
            send(stations[x],DPT1,Q.,00);      {get temperature data}
            recv(data);
            temp=val(data);            {convert temperature to integer}
            get time(hour,minute,sec);

            {write the data to the screen}
            write("Station #",x," has a temperature of ",temp);
            write(" at ",hour,":",minute,"\n");

            {write the data to a file}
            write(filvar,"Station #",x," has a temperature of ",temp);
            write(filevar," at ",hour,":",minute,"\n");


            {turn on heat if temperature is too low}
            if (temp <70)
                  then send(stations[x],RELP,Q.,11,04);

            {turn off heat if temperature is too high}
            if (temp>80)
                  then send(stations[x],RELP,Q.,11,00);

            wait(delay*60);            {wait until time for next check}
      next
wend

close(filevar);
write("End of Temperature monitoring\n");

end
```