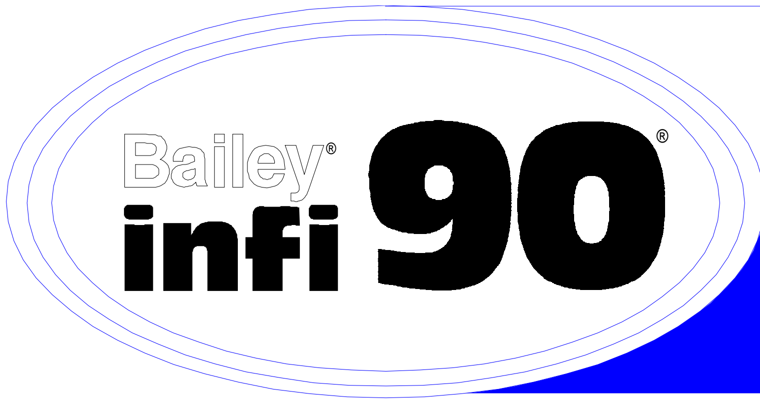
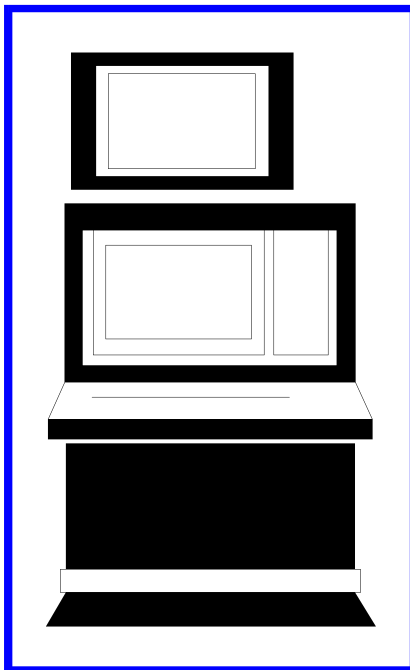
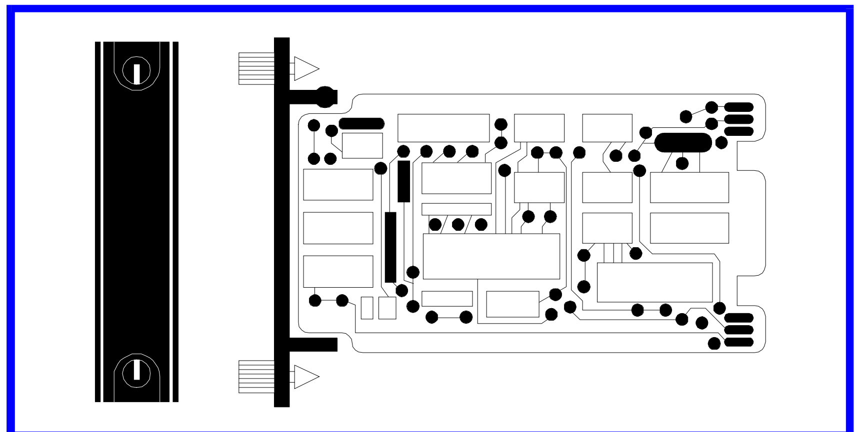
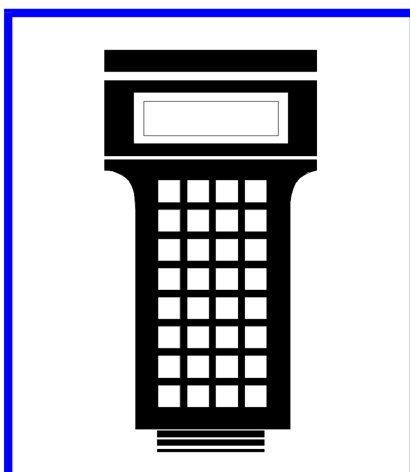
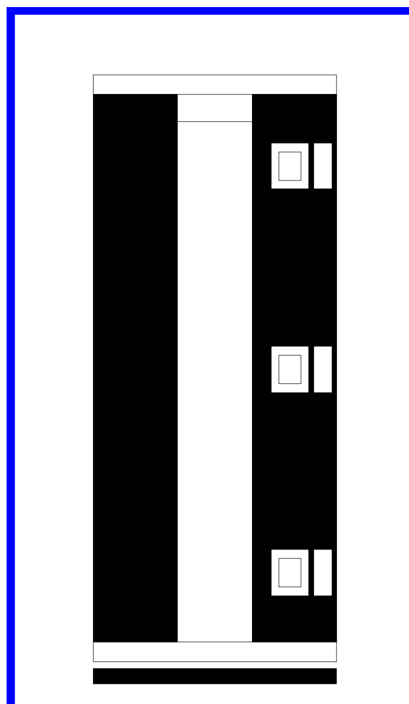
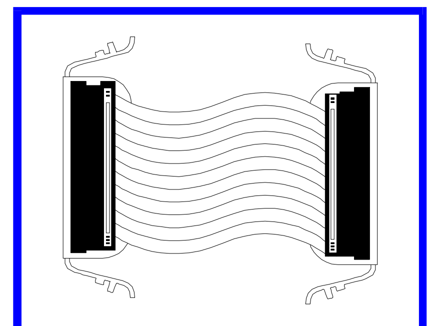
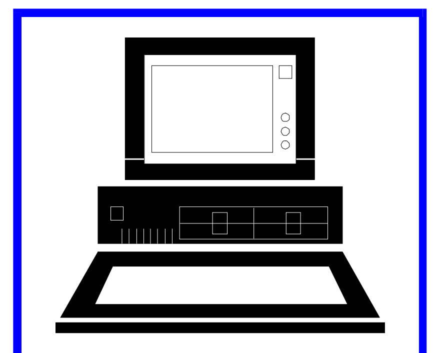


E96-703

Instruction

C Utility Program
(Release 2.0)

Bailey
Process Control and
Automation Solutions
from Elsag Bailey Group

WARNING notices as used in this instruction apply to hazards or unsafe practices that could result in personal injury or death.

CAUTION notices apply to hazards or unsafe practices that could result in property damage.

NOTES highlight procedures and contain information that assists the operator in understanding the information contained in this instruction.

WARNING

INSTRUCTION MANUALS

DO NOT INSTALL, MAINTAIN, OR OPERATE THIS EQUIPMENT WITHOUT READING, UNDERSTANDING, AND FOLLOWING THE PROPER **Elsag Bailey** INSTRUCTIONS AND MANUALS; OTHERWISE, INJURY OR DAMAGE MAY RESULT.

RADIO FREQUENCY INTERFERENCE

MOST ELECTRONIC EQUIPMENT IS INFLUENCED BY RADIO FREQUENCY INTERFERENCE (RFI). CAUTION SHOULD BE EXERCISED WITH REGARD TO THE USE OF PORTABLE COMMUNICATIONS EQUIPMENT IN THE AREA AROUND SUCH EQUIPMENT. PRUDENT PRACTICE DICTATES THAT SIGNS SHOULD BE POSTED IN THE VICINITY OF THE EQUIPMENT CAUTIONING AGAINST THE USE OF PORTABLE COMMUNICATIONS EQUIPMENT.

POSSIBLE PROCESS UPSETS

MAINTENANCE MUST BE PERFORMED ONLY BY QUALIFIED PERSONNEL AND ONLY AFTER SECURING EQUIPMENT CONTROLLED BY THIS PRODUCT. ADJUSTING OR REMOVING THIS PRODUCT WHILE IT IS IN THE SYSTEM MAY UPSET THE PROCESS BEING CONTROLLED. SOME PROCESS UPSETS MAY CAUSE INJURY OR DAMAGE.

AVERTISSEMENT

MANUELS D'OPÉRATION

NE PAS METTRE EN PLACE, RÉPARER OU FAIRE FONCTIONNER L'ÉQUIPEMENT SANS AVOIR LU, COMPRIS ET SUIVI LES INSTRUCTIONS RÉGLEMENTAIRES DE **Elsag Bailey**. TOUTE NÉGLIGENCE À CET ÉGARD POURRAIT ÊTRE UNE CAUSE D'ACCIDENT OU DE DÉFAILLANCE DU MATÉRIEL.

PERTURBATIONS PAR FRÉQUENCE RADIO

LA PLUPART DES ÉQUIPEMENTS ÉLECTRONIQUES SONT SENSIBLES AUX PERTURBATIONS PAR FRÉQUENCE RADIO. DES PRÉCAUTIONS DEVRONT ÊTRE PRISES LORS DE L'UTILISATION DU MATÉRIEL DE COMMUNICATION PORTATIF. LA PRUDENCE EXIGE QUE LES PRÉCAUTIONS À PRENDRE DANS CE CAS SOIENT SIGNALÉES AUX ENDROITS VOULUS DANS VOTRE USINE.

PERTURBATIONS DU PROCÉDÉ

L'ENTRETIEN DOIT ÊTRE ASSURÉ PAR UNE PERSONNE QUALIFIÉE EN CONSIDÉRANT L'ASPECT SÉCURITAIRE DES ÉQUIPEMENTS CONTRÔLÉS PAR CE PRODUIT. L'AJUSTEMENT ET/OU L'EXTRACTION DE CE PRODUIT PEUT OCCASIONNER DES À-COUPS AU PROCÉDÉ CONTRÔLE LORSQU'IL EST INSÉRÉ DANS UNE SYSTÈME ACTIF. CES À-COUPS PEUVENT ÉGALEMENT OCCASIONNER DES BLESSURES OU DES DOMMAGES MATÉREILS.

NOTICE

The information contained in this document is subject to change without notice.

Elsag Bailey, its affiliates, employees, and agents, and the authors and contributors to this publication specifically disclaim all liabilities and warranties, express and implied (including warranties of merchantability and fitness for a particular purpose), for the accuracy, currency, completeness, and/or reliability of the information contained herein and/or for the fitness for any particular use and/or for the performance of any material and/or equipment selected in whole or part with the user of/or in reliance upon information contained herein. Selection of materials and/or equipment is at the sole risk of the user of this publication.

This document contains proprietary information of Elsag Bailey, Elsag Bailey Process Automation, and is issued in strict confidence. Its use, or reproduction for use, for the reverse engineering, development or manufacture of hardware or software described herein is prohibited. No part of this document may be photocopied or reproduced without the prior written consent of Elsag Bailey.

Preface

The purpose of this manual is to provide the information necessary to operate the Bailey C Utility Program release 2.0 and other associated software. These software packages allow utilization of C programming in IMMFP01, IMMFP02, IMMFP03, and IMMFC03 modules. This manual is not intended as a tutorial in C programming techniques.

List of Effective Pages

Total number of pages in this instruction is 108, consisting of the following:

Page No.	Change Date
Preface	Original
List of Effective Pages	Original
iii through vi	Original
1-1 through 1-7	Original
2-1 through 2-4	Original
3-1 through 3-2	Original
4-1 through 4-21	Original
5-1 through 5-42	Original
6-1 through 6-6	Original
7-1 through 7-8	Original
A-1 through A-2	Original
B-1 through B-2	Original
C-1	Original
D-1 through D-5	Original
Index-1 through Index-2	Original

When an update is received, insert the latest changed pages and dispose of the superseded pages.

NOTE: On an update page, the changed text or table is indicated by a vertical bar in the outer margin of the page adjacent to the changed area. A changed figure is indicated by a vertical bar in the outer margin next to the figure caption. The date the update was prepared will appear beside the page number.

Table of Contents

Page

SECTION 1 - INTRODUCTION 1-1
PURPOSE 1-1
MANUAL CONTENT 1-1
REQUIREMENTS 1-1
Hardware 1-1
Software 1-2
Module Initialization 1-2
Function Blocks 1-2
USER QUALIFICATIONS 1-3
MODULE MEMORY 1-4
GLOSSARY OF TERMS AND ABBREVIATIONS 1-4
REFERENCE DOCUMENTS 1-7

SECTION 2 - INSTALLATION 2-1
INTRODUCTION 2-1
Installing the C Utility Program 2-1
Installing the Cross Compiler Program 2-1
Setting C Environment Variables 2-1
INSTALLING CREATED DIRECTORIES AND FILES 2-2
HARDWARE CONNECTIONS 2-2

SECTION 3 - C APPLICATION DEVELOPMENT 3-1
DEVELOPING C PROGRAMS 3-1

SECTION 4 - C UTILITY PROGRAM 4-1
INTRODUCTION 4-1
CREATING/LOADING C SPECIFICATION FILES 4-1
EDITING C SPECIFICATION FILES 4-3
Object File List Editing 4-4
User Memory Specifications Editing 4-4
System Memory Specifications Editing 4-5
Format Specifications Editing 4-6
BUILD LINKER COMMAND FILE 4-8
INVOKING THE LINKER PROGRAM 4-8
MODULE MANAGEMENT 4-8
Setting the Module Mode to Configure 4-8
Setting the Module Mode to Execute 4-8
Resetting the Module 4-9
Formatting the Module 4-9
Adding Function Code 143 to the Configuration 4-9
Erasing a Function Block 4-10
Listing Function Block Assignments 4-10
C PROGRAM MANAGEMENT 4-10
Transferring a C Program from the Work Station to the Module 4-10
Deleting C Programs from the Module 4-11
DATA FILE MANAGEMENT 4-11
MODEM SETUP 4-13
Initiating Modem Connection 4-16
Editing the Telephone Directory 4-16
CUP ERROR MESSAGES 4-17

Table of Contents (continued)

Page

SECTION 5 - MODULE SPECIFIC FUNCTIONS 5-1
INTRODUCTION 5-1
bin 5-1
blk_wrt 5-3
bout 5-6
checkpoint_file 5-8
cpfil 5-10
dadig_in 5-12
dasin 5-14
filxfer 5-17
fltr3 5-21
getargs 5-22
inque 5-24
mb_addr 5-25
outque 5-26
portopen 5-27
putargs 5-31
putmsg 5-32
r3flt 5-33
rfilef 5-34
rfiler 5-37
sysclk 5-40
time_ms 5-41

SECTION 6 - DEBUGGER PROGRAM 6-1
INTRODUCTION 6-1
PROGRAM OPERATION 6-1
The File Menu 6-2
The Search Menu 6-3
The Run Menu 6-3
The Watch Menu 6-4
The Break Menu 6-5
The Module Menu 6-5
The Options Menu 6-6

SECTION 7 - SAMPLE APPLICATION 7-1
INTRODUCTION 7-1
THE SAMPLE PROGRAM 7-2
MAIN.C Source File 7-2
GETNUMB.C Source File 7-2
DISPFILE.C Source File 7-3
WORK STATION BASED TESTING 7-4
Creating an Object File 7-4
Creating an Executable File 7-4
Running and Debugging the Program 7-4
PREPARING FOR MODULE EXECUTION 7-4
Compiling the Program 7-4
Creating the C Specifications File 7-5
Linking and Downloading the Program 7-7
EXECUTING THE PROGRAM 7-8

Table of Contents (continued)

Page

APPENDIX A - FILE SYSTEM CHECK COMMAND A-1
FSCK COMMAND DETAILS A-1

APPENDIX B - MAKE FILE SYSTEM COMMAND B-1
MKFS COMMAND DETAILS B-1

APPENDIX C - REDUNDANT HARD DRIVE COPY OPERATIONS C-1
COPYING PROCEDURE C-1

APPENDIX D - AVAILABLE C LANGUAGE FUNCTIONS D-1
FUNCTION LIST D-1

List of Figures

No.	Title	Page
2-1.	Work Station to INFI-NET Communication Example	2-3
2-2.	Work Station to Plant Loop Communication Example	2-4
4-1.	CUP Main Menu	4-2
4-2.	CSP Title Information Screen	4-3
4-3.	Edit C Specifications Menu	4-3
4-4.	User RAM Specifications Screen	4-4
4-5.	System RAM Specifications Screen	4-6
4-6.	Edit Format Specifications Screen	4-7
4-7.	Module Management Menu	4-9
4-8.	C Program Management Menu	4-10
4-9.	Data File Management Screen	4-14
4-10.	Example Data File Management Shell	4-14
4-11.	Modem Setup Menu	4-16
4-12.	Modem Initialization Screen	4-17
4-13.	Edit Telephone Directory Screen	4-17
6-1.	Debugger Program Main Menu	6-2
6-2.	The File Pull Down Menu	6-3
6-3.	The Search Pull Down Menu	6-3
6-4.	The Run Pull Down Menu	6-4
6-5.	The Watch Pull Down Menu	6-4
6-6.	The Break Pull Down Menu	6-5
6-7.	The Module Pull Down Menu	6-5
6-8.	The Option Pull Down Menu	6-6
7-1.	Sample Hardware Configuration	7-1

List of Tables

No.	Title	Page
1-1.	Module Initialization Procedure	1-3
1-2.	Available Memory for C Programs	1-4
1-3.	Memory Space Usage	1-4
1-4.	Glossary of Terms and Abbreviations	1-5
1-5.	Reference Documents	1-7
2-1.	Directory and File Listing	2-2
3-1.	Suggested Program Development Procedure	3-1
4-1.	Data File Management Commands	4-11
4-2.	CUP Error Messages	4-18
5-1.	Error Status Information	5-19
D-1.	Available C Language Functions	D-1

SECTION 1 - INTRODUCTION

PURPOSE

The IMMFP01, IMMFP02 and IMMFP03 Multi-Function Processor modules and IMMFC03 Multi-Function Controller module are the most versatile members of the INFI 90® controller family. The ability to execute C language programs is a key feature of these controllers. The addition of C language programs to a control strategy provides increased versatility, multi-tasking capability, and access to other module resources.

MANUAL CONTENT

This manual is organized into seven sections.

Introduction Provides an overview of the requirements and preliminary information necessary to use C programs in multi-function processor (MFP) or multi-function controller (MFC) modules.

Installation Explains how to install the C development software and C Utility Program.

C Application Development Provides guidelines on how to write, develop, test and execute C programs in a MFP or MFC module.

C Utility Program Provides instructions on how to use this program.

Module Specific Functions Lists the functions specific to MFP or MFC modules.

Debugger Program Explains the abilities and use of the troubleshooting tool.

Sample Application Carries a sample program through the development process.

REQUIREMENTS

Using C language programs in the INFI 90 Strategic Process Management System requires the following items.

Hardware

- Bailey engineering work station.

or

® INFI 90 is a registered trademark of Elsag Bailey Process Automation.
® IBM is a registered trademark of International Business Machines.

- IBM® compatible 286-based or 386-based personal computer with a minimum of:

  640 kilobytes of RAM.

  Serial port (RS-232-C).

  High density 5.25-inch or 3.5-inch floppy disk drive as drive A.

  Six megabytes of free memory space on a hard disk drive as drive C.

- INFI 90 IMMFP01, IMMFP02 or IMMFP03 Multi-Function Processor module. (The debugger portion of the CUP program requires firmware revision D_0 or greater).

  or

- INFI 90 IMMFC03 Multi-Function Controller module.

- INFI 90 INPCI01 Plant Loop to Computer Interface, INPCI02 Plant Loop to Computer Interface, INICI01 INFI-NET® to Computer Interface, INICI03 INFI-NET to Computer Interface, IMCPM02 Communication Port Module (with required serial cable), or IMSPM01 Serial Port Module (with required serial cable).

Software

The Bailey C Utility Program (Release 2.0) software package contains the following items:

- Bailey C Utility Program (CUP).
- Microtec® ANSI C Cross Compiler.

Module Initialization

Each MFP or MFC module must be initialized prior to downloading any C language program information. Table 1-1 lists the steps required for each type of module according to the communication system it uses.

Function Blocks

The function block configuration in the module must contain at least one invoke C command (function code 143). Function code 143 executes the C program located in the same memory segment as the function code. If the C programs are being developed with the CAD/TXT software package, one C allocation command (function code 144) should be used in the module. Function code 144 informs the CAD/TXT program of the

® INFI-NET is a registered trademark of Elsag Bailey Process Automation.
® Microtec is a registered trademark of Microtec Research, Inc.

amount of RAM and NVRAM memory allocated for C programs by the C Utility Program.

Table 1-1. Module Initialization Procedure

Module Type	Communication System Used	Initialization Procedure[1]
IMMFP01/02/03	Plant Loop	1. Remove module. 2. Locate dipswitch SW2. 3. Set poles 1 and 7 of SW2 to 1. 4. Install module. 5. Wait until LEDs 1 through 6 are on. 6. Remove module. 7. Set poles 1 and 7 to 0. 8. Install module.
IMMFP01/02/03	INFI-NET	1. Remove module. 2. Locate dipswitch SW2. 3. Set poles 1 and 7 of SW2 to 1. 4. Install module. 5. Wait until LEDs 1 through 6 are on. 6. Remove module. 7. Set pole 6 to 1 and 7 to 0. 8. Install module. 9. Wait until LEDs 1 through 6 are on. 10. Remove module. 11. Set poles 1 and 6 to 0. 12. Install module.
IMMFC03	Plant Loop	1. Remove module. 2. Locate dipswitch U72. 3. Set poles 1 and 3 of U72 to 1. 4. Install module. 5. Wait until LEDs 1 through 6 are on. 6. Remove module. 7. Set poles 1 and 3 to 0. 8. Install module.
IMMFC03	INFI-NET	1. Remove module. 2. Locate dipswitch U72. 3. Set poles 1 and 3 of U72 to 1. 4. Install module. 5. Wait until LEDs 1 through 6 are on. 6. Remove module. 7. Set pole 4 to 1 and 3 to 0. 8. Install module. 9. Wait until LEDs 1 through 6 are on. 10. Remove module. 11. Set poles 1 and 4 to 0. 12. Install module.

NOTE: 1. 1 = open (OFF), 0 = closed (ON). Set all remaining poles to 0.

USER QUALIFICATIONS

The developer of C language programs for the INFI 90 Strategic Process Management System must have a working knowledge of process control, Bailey function codes, and C programming language.

MODULE MEMORY

The amount and types of memory available for C programs depends on the module used. Table 1-2 lists the available RAM and NVRAM according to module type. Table 1-3 shows an itemized account of memory space usage.

Table 1-2. Available Memory for C Programs

Module Type	RAM (bytes)	NVRAM (bytes)
IMMFC03	319,488	75,738
IMMFP01	155,648	59,354
IMMFP02	339,968	191,450
IMMFP03	1,564,608	420,828

Table 1-3. Memory Space Usage

<table>
<tr><th>Memory Type</th><th>Area</th><th>Memory Contents</th><th>Size</th></tr>
<tr><td rowspan="10">RAM</td><td colspan="2">C RAM (2 parts)</td><td>User specified (Total RAM allocated to C:)</td></tr>
<tr><td rowspan="4">1. C user RAM</td><td>Idata section</td><td>User specified (Max size of C idata section:)</td></tr>
<tr><td>Udata section</td><td>User specified (Max size of C udata section:)</td></tr>
<tr><td>Segment data</td><td>Derived from segment stack size (((stack size or 2 kbytes) = 2000)* no. of segment entry points)</td></tr>
<tr><td>Dynamic memory pool</td><td>User specified (Min size of dynamic memory pool:)</td></tr>
<tr><td rowspan="4">2. C system RAM</td><td>Checkpoint Buffers</td><td>Derived from (no. of checkpoint buffers*(checkpoint buffer size + 30))</td></tr>
<tr><td>MBF buffers</td><td>Derived from (no. of file buffers* (file buffer size + 40))</td></tr>
<tr><td>Code section</td><td>User specified (Max size of C code section:)</td></tr>
<tr><td>Unused</td><td></td></tr>
<tr><td colspan="2">Function block space</td><td>Derived from (total RAM minus RAM allotted to C language programming)</td></tr>
<tr><td rowspan="4">NVRAM</td><td colspan="2">Module format table</td><td>34 bytes</td></tr>
<tr><td colspan="2">File directory</td><td>Derived from (36*(5 + max no. of data files))</td></tr>
<tr><td colspan="2">File space</td><td>User specified (Total NVRAM allocated to C:)</td></tr>
<tr><td colspan="2">Function block space</td><td>Remaining NVRAM not used.</td></tr>
</table>

GLOSSARY OF TERMS AND ABBREVIATIONS

Table 1-4 list the definitions of terms and abbreviations used in this manual.

Table 1-4. Glossary of Terms and Abbreviations

Term	Definition
Baud	Rate at which data is transmitted over a serial bus in bits per second.
.C File	ASCII text file (***.C*** extension) that contains a file to be read by the compiler.
C Map File	Module resident file (file number 32767) containing memory map of the C program. The C Utility Program creates this file.
C RAM	Continuous block of RAM memory in the module containing all configurable, RAM resident structures of the C environment. It consists of two parts: C user RAM and C system RAM memory.
C System RAM	Part of C RAM memory containing code section, checkpoint buffers and MBF buffers.
C User RAM	Part of C RAM memory containing data structures (segment stacks, C library data, dynamic memory pool, etc.) accessible to the C program.
Checkpoint Buffers	The MFP/MFC module buffers that provide temporary storage for file data transmitted from primary MFP/MFC to backup MFP/MFC.
.CMD File	A temporary file containing C program information used by the Microtec linker program.
Code Section	The executable code of the C program.
Code Section File	Module resident file (file number 32765) containing the C program code section.
.CSP File	Work station resident file (***.CSP*** file extension) containing information about the C environment of the module.
Compiler	A program whose purpose is that of translating high-level language statements into a form that can directly activate the device hardware.
Configuration	The act of setting up equipment to accomplish specific functions or a list of parameters associated with such a setup.
Controlway	High speed, redundant, peer-to-peer communication link. Used to transfer information between intelligent modules within the a process control unit.
CPM	Communication port module. A module in the process control unit that allows the work station to communicate with a module via the Controlway communication link.
CUP	C Utility Program. The software used to specify and manage the C environment in the module.
Dynamic File	A module file that can be modified while the module is in EXECUTE mode.
EWS	Engineering work station.
File Attributes	The read and write access levels specified when a file is created in a module.
File ID	Unique C file identifier (numerical).
Format Operation	Operation performed on the module to format and initialize the user configurable memory.
Format Specifications	The set of specifications that define the module memory format.
Function Block	The occurrence of a function code at a block address of a module.
Function Code	An algorithm that manipulates specific functions. These functions are linked together to form the control strategy.
Idata Section File	Module resident file (file number 32766) containing C program initialized data such as constants, character strings, etc.

Table 1-4. Glossary of Terms and Abbreviations *(continued)*

Term	Definition
INFI-NET	Advanced data communication highway.
Linker (DOS) Program	Work station resident linker program that combines a set of ***.OBJ*** files into one ***.LMS*** file that can be run and tested within the work station.
.LMS File	Work station resident file (***.LMS*** file extension) containing a C program code section and idata section.
.MAP File	Work station resident file (***.MAP*** file extension) containing the memory map of a C program load module.
MBF Buffers	Module resident buffers that provide temporary storage of file data while the file is being accessed via the module bus.
MFC	Multi-function controller module. A multiple-loop controller with data acquisition and information processing capabilities.
MFP	Multi-function processor module. A multiple-loop controller with data acquisition and information processing capabilities.
Module Address	A unique identifier of a specific device or a communication channel. Refers to Controlway or module bus address.
Module Bus	Peer to peer communication link used to transfer information between intelligent modules within a process control unit.
NIU	Network interface unit. Term used for all local and remote interfaces, computer interfaces and console interfaces.
Node	A point of interconnection to a network.
Node Address	A unique identifier of a specific device or a communication channel. Refers to Plant Loop, Superloop or INFI-NET address.
NVRAM	Nonvolatile random access memory. Retains stored information when power is removed.
.OBJ File	Work station resident file (***.OBJ*** file extension) produced by a C compiler.
PCU	Process control unit. A node on the plant-wide communication network containing master and slave modules.
Plant Loop	Network 90® data communication highway.
Segment	A grouping of function blocks separated from other function blocks by function code 82 (segment control).
Segment Entry Point	The function in the C program that is called by an invoke C command. There is one entry point for every segment containing invoke C commands. Multiple invoke C commands in a segment call the same entry point. Two or more segments may share the same entry point.
Segment Stack	An area in RAM used by the C program to store temporary data. There is one stack for every segment that invokes C.
Set Point	Target value for a process variable representing desired performance of the process variable.
SPM	Serial port module; a module in the process control unit that allows the work station to communicate with a module via the module bus.
Udata Section File	Module resident file containing C program uninitialized data such as uninitialized arrays, etc.

REFERENCE DOCUMENTS

Table 1-5 lists the documents referenced in this manual.

Table 1-5. Reference Documents

Number	Document
I-E96-200	Function Code Applications Manual
I-E96-201	Multi-Function Processor Module IMMFP01
I-E96-202	Multi-Function Processor Module IMMFP02
I-E96-203	Multi-Function Processor Module IMMFP03
I-E96-211	Multi-Function Controller Module IMMFC03
I-E96-217	Serial Port Module IMSPM01
I-E96-221	Communication Port Module IMCPM02
I-E96-601	INFI-NET Communications Modules
I-E96-620	Plant Loop to Computer Interface INPCI01
I-E96-621	Plant Loop to Computer Interface INPCI02
I-E96-701	Personal Computer Software Computer-Aided Drawing/Text (CAD/TXT)

® Network 90 is a registered trademark of Elsag Bailey Process Automation.

SECTION 2 - INSTALLATION

INTRODUCTION

There are three steps to installing the Bailey C Utility Program release 2.0 software. The first step is to install the C Utility Program. The second step is to install the Microtec Cross Compiler Program. The third step is to set C environment variables. The C Utility Program resides on one disk. The cross compiler generates module executable C language code and resides on two disks.

Installing the C Utility Program

Install the C Utility Program using the following steps.

1. Insert Bailey Controls C Utility Program disk into Drive A.
2. Type the following:

   INSTALL Enter

3. Press any key.

Installing the Cross Compiler Program

Refer to the Microtec Cross Compiler instruction manual for detailed installation instructions.

Setting C Environment Variables

The ***AUTOEXEC.BAT*** file in the root directory of the work station must contain the following statement.

SET CUP_COMPILER = MICROTEC
(if the Microtec compiler is used)

or

SET CUP_COMPILER = LATTICE
(if the Lattice® compiler is used)

NOTE: If the ***AUTOEXEC.BAT*** file is not modified, the CUP program defaults to the **MICROTEC** compiler.

This statement tells the C Utility Program which compiler and linker to use. The path statement in the ***AUTOEXEC.BAT*** file must be set to look for commands in the C:\CMFC,

® Lattice is a registered trademark of Lattice Incorporated.

C:\MCC68K, and C:\ASM68K directories. Add the following to the existing path statement.

;C:\CMFC; C:\MCC68K; C:\ASM68K;

INSTALLING CREATED DIRECTORIES AND FILES

The installation process creates three directories. These directories are C:\CMFC, C:\MCC68K, and C:\ASM68K. Refer to Table 2-1 for a brief explanation of the contents of these directories.

Table 2-1. Directory and File Listing

Directory	Contents
C:\CMFC	Compiler Batch, Linker Batch, Help, and Bailey Utility files. Also included is the ***CUP.EXE*** and ***CDEBUG.EXE*** files.
C:\MCC68K	Microtec C Compiler files.
C:\ASM68K	Microtec Assembler and Linker files.

HARDWARE CONNECTIONS

There are several ways to connect a work station to INFI-NET or Plant Loop communication highways. Connecting a work station to an INFI-NET communication highway requires an INICI01 or INICI03 INFI-NET to Computer Interface. Refer to Figure 2-1 for an example. Connecting a work station to a Plant Loop communication highway requires an INPCI01 or INPCI02 Plant Loop to Computer Interface. Refer to Figure 2-2 for an example. Connect the work station to these interfaces with a standard RS-232-C serial cable. Connecting a work station to the Controlway communication link requires the IMCPM02 Communication Port Module (see Figure 2-1). Connecting a work station to the module bus communication link requires the IMSPM01 Serial Port Module (see Figure 2-2). Refer to the appropriate instruction manual for more details (refer to Table 1-5).

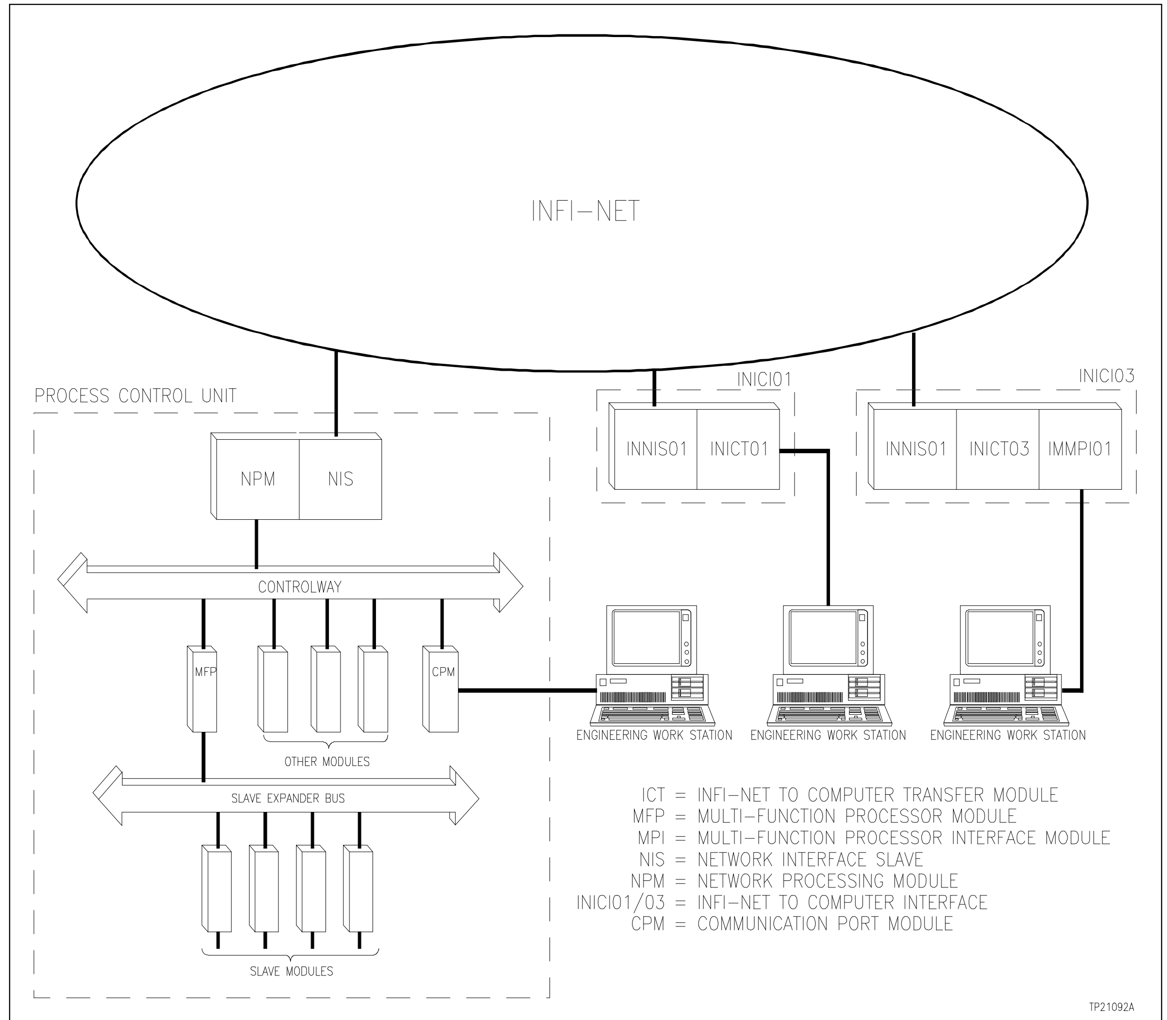


Figure 2-1. Work Station to INFI-NET Communication Example

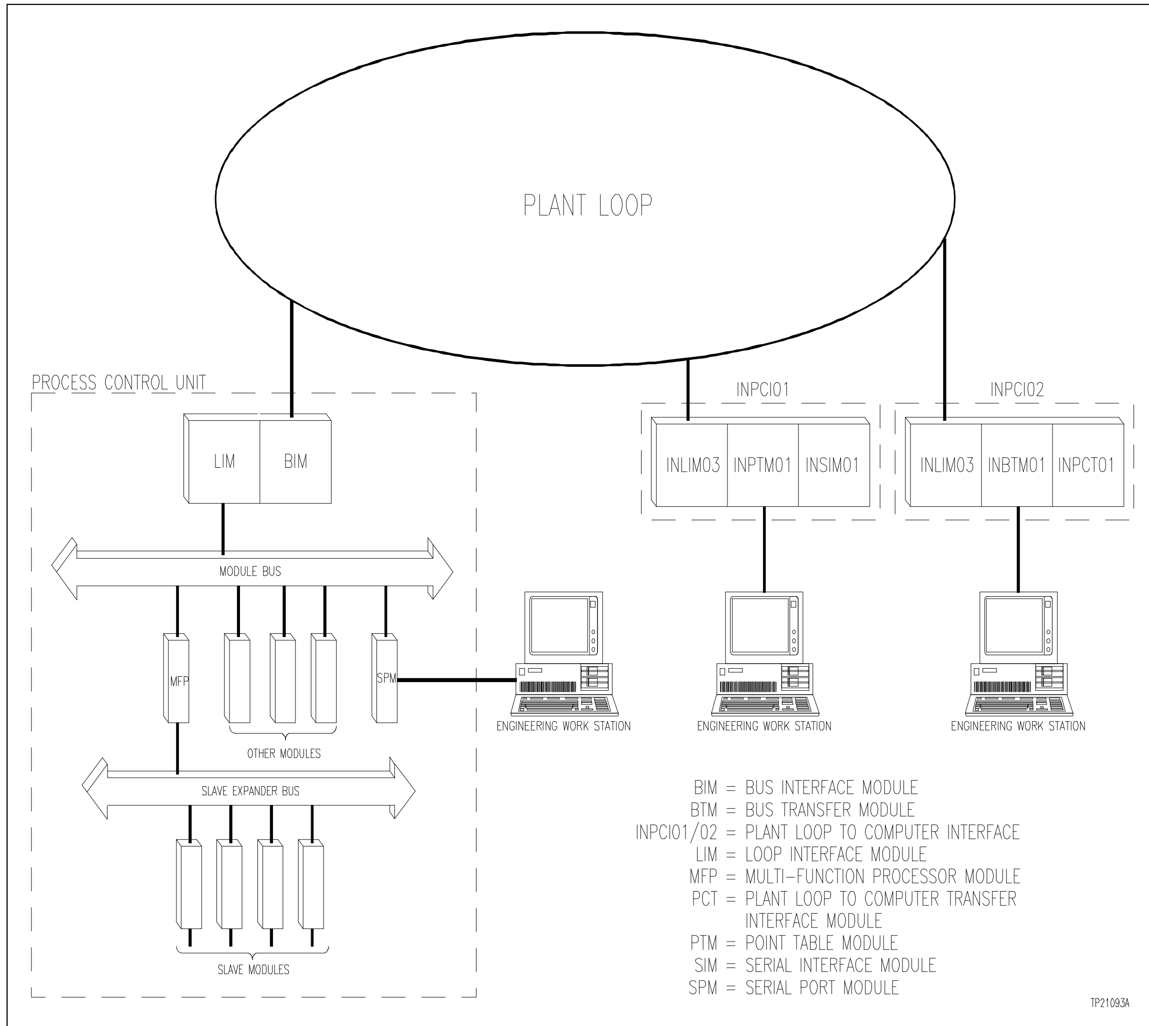


Figure 2-2. Work Station to Plant Loop Communication Example

SECTION 4 - C UTILITY PROGRAM

INTRODUCTION

The C Utility Program (CUP) serves as a bridge between C program development on a host work station and C program execution in a MFP or MFC module. The CUP program can specify C memory partitions in the module as well as managing, linking, and downloading C programs to a target module. Several module file and configuration operations are also available. The description of each option follows. To start the CUP program, type the following at the DOS prompt:

CUP [*port*] [*baud*] [*file_name***.CSP**]

Where:

[*port*]	is the work station port (COM1 or COM2) connected to the INFI 90 network interface unit, CPM module, or SPM module. This number must be a 1 (COM1) or a 2 (COM2) with 1 being the default value.
[*baud*]	is the baud rate for the work station to module interface. Allowable baud rates are 150, 300, 1200, 2400, 4800, 9600, and 19200. The default baud rate is 9600.
[*file_name***.CSP**]	is the name of the C specification file corresponding to a previously created C program.

NOTE: The [*port*], [*baud*], and [*file_name***.CSP**] fields are optional and can be entered in any order.

This command brings up the CUP title page and, after a short delay, the *CUP MAIN MENU* (see Figure 4-1). These menu options are presented in the logical sequence of events needed to prepare a C program for execution in a MFP or MFC module. On-line help is available at any time (while in the CUP program) by simultaneously pressing **Alt** - **H**.

NOTE: The C program should be created, edited, and compiled (as detailed in Section 3) before executing the options listed on the *CUP MAIN MENU*.

CREATING/LOADING C SPECIFICATION FILES

The first step in preparing a C program for execution is to create a new C specification file or edit an existing file. The C specifications file (***.CSP*** file extension) contains general information regarding the C program. This information consists of

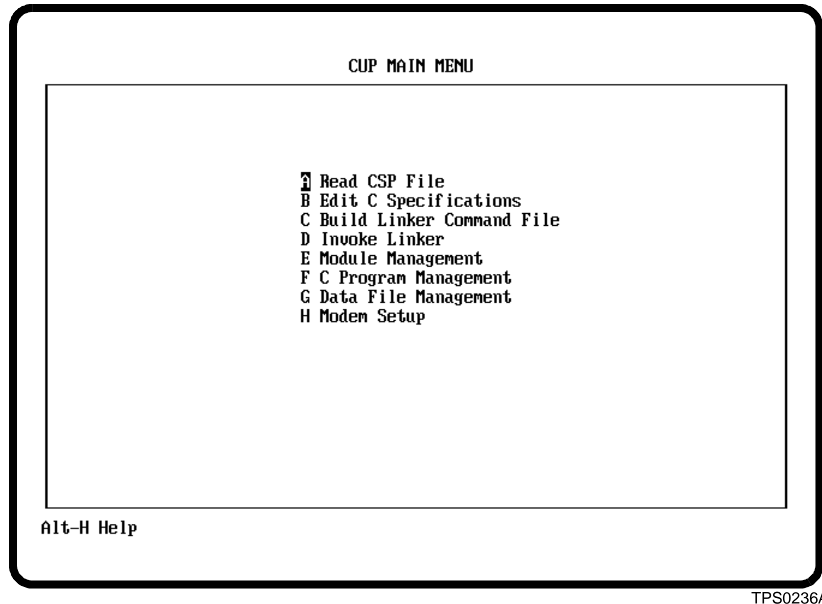


Figure 4-1. CUP Main Menu

user name, module type, loop address, process control unit address, module address, comments, etc.

The C specification file also contains the module partitioning requirements. To execute a C program in a MFP or MFC module, memory must be set aside for program and data space. This memory is partitioned for the program and is separated from the memory that executes the function codes.

To create a new ***.CSP*** file, select *A Read CSP File* from the *CUP MAIN MENU*. Enter the name of the new ***.CSP*** file at the *CSP Filename* prompt. Press F10 to enter the file name. The CUP program searches the current directory for the file. If the file does not exist, the CUP program creates a new file. Entering the name of an existing ***.CSP*** file loads that file. A path may be specified while entering the file name. If a path is specified, the edited ***.CSP*** file and any related files are placed in the specified path. Entering the default *CSP Filename (*.CSP)* starts a wildcard ***.CSP** file search.

Once the **.CSP** file is loaded or created, the *CSP TITLE INFORMATION* screen (See Figure 4-2) will appear. The *Loop Address, PCU Address, Module Address, Module Type, Hardware Ver* (circuit board version), *Software Ver* (module firmware version), and *BATCH present?* fields must be accurate according to the existing INFI 90 system configuration. The values entered in these fields are used as defaults for all subsequent module functions. The user *Name, Plant Site, Contract #, Program Description*, and *Comments* fields are for information only and do not affect program execution.

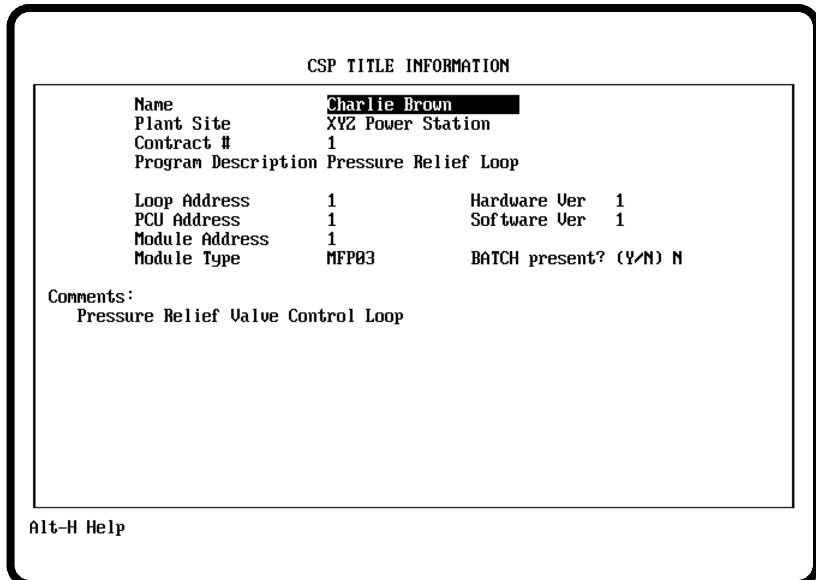


Figure 4-2. CSP Title Information Screen

EDITING C SPECIFICATION FILES

After a **.CSP** file is created or loaded, it can be edited by selecting *B Edit C Specifications* from the *CUP MAIN MENU*. This causes the *EDIT C SPECIFICATIONS* menu (see Figure 4-3) to appear. A menu of four editing options appears. Below these options is a breakdown of the available memory, memory formatted for C programs, memory required for the C program, and unused memory. An explanation of each menu option follows.

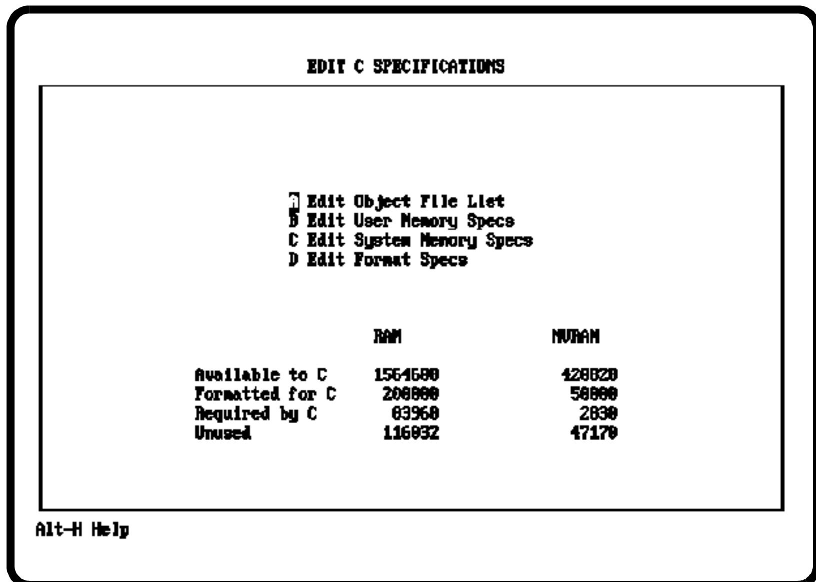


Figure 4-3. Edit C Specifications Menu

Object File List Editing

Selecting *A Edit Object File List* allows the C object files (***.OBJ*** file extension) and user supplied library files that make up the C program to be entered. When entering library files on this list, specify the proper file extension. The Microtec linker program uses this list to determine which files must be linked together for the C program. Up to 100 files can be entered in this list. Pressing **Page Up** or **Page Dn** scrolls the list in groups of ten files.

After building the list and exiting the screen, the CUP program reads each object file and calculates memory (idata, udata, and code) usage. If there are no object files on the list, the *Cannot compute memory requirements* message appears. The calculated memory values can then be transferred to the memory requirement specifications of the C specification file (***.CSP*** file). To transfer the specifications answer **Y** to the *Transfer to CSP? (Y/N)* prompt. Make the appropriate corrections to any noted errors before escaping this screen.

User Memory Specifications Editing

After building an object file list, select *B Edit User Memory Specs*. The *EDIT USER RAM SPECIFICATIONS* screen (see Figure 4-4) will appear. This screen allows segment entry point names, and user RAM specifications to be entered or edited. If memory usage values were transferred to the C specification file (***.CSP*** file), the idata and udata values will already be specified.

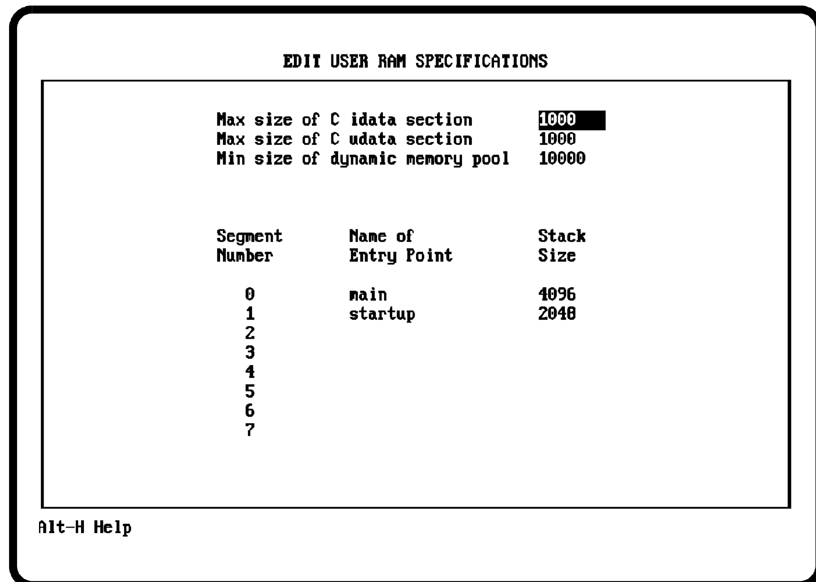


Figure 4-4. User RAM Specifications Screen

The *EDIT USER RAM SPECIFICATIONS* screen requests the following information:

Max size of C idata section

This field limits the maximum size (in bytes) of the initialized data section of memory.

Max size of C udata section

This field limits the maximum size (in bytes) for the uninitialized data section of memory.

Min size of dynamic memory pool

This field limits the minimum size (in bytes) for the section of memory reserved for dynamic memory functions. This section contains any excess C configured program memory.

Name of Entry Point

These eight fields list the names of the executable C programs in each segment that contains a function code 143 (invoke C).

Stack Size

These eight fields specify the size (in bytes) of the entry point stacks.

The C programmer should have some idea of the size requirements for each of these fields. For example, the idata and udata sections should be large when large quantities of static data are used. When using C dynamic memory functions (malloc(), free(), etc.), the dynamic memory pool should be large enough to accommodate the functions. The stack size should be set based on C program flow. If there are several nested function calls in the C program, a lot of local function data, or several recursive function calls, these values should be set larger accordingly.

System Memory Specifications Editing

Selecting *C Edit System Memory Specs* causes the *EDIT SYSTEM RAM SPECIFICATIONS* screen (see Figure 4-5) to appear. This screen requests the following information:

Max size of C code section

This field limits the maximum size (in bytes) of C program. This value should be as large as the program itself plus some extra memory to account for small program changes.

Number of checkpoint buffers

This field specifies the number of buffers used when sending file information to a redundant MFP or MFC module. If there is no redundant MFP or MFC module, set this field to zero.

Size of each checkpoint buffer

This field specifies the size (in bytes) of each checkpoint buffer. For optimum performance, set this field to a value greater than that of the largest file sent to the redundant MFP or MFC module. If there is no redundant MFP or MFC module, set this field to zero.

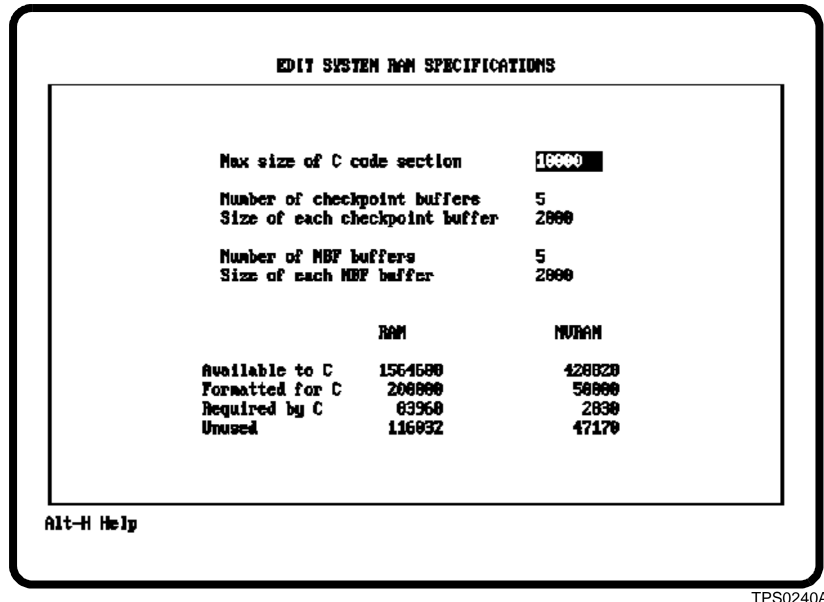


Figure 4-5. System RAM Specifications Screen

Number of MBF buffers

This field specifies the number of buffers that provide temporary storage of file data while accessing a file via the Controlway or module bus communication link.

Size of each MBF buffer

This field specifies the size (in bytes) of each MBF buffer. For optimum performance, set this field to a value greater than that of the largest file to be accessed during file transfers.

Format Specifications Editing

Selecting *D Edit Format Specs* allows the total RAM and NVRAM memory allocated for C program use to be specified. The amount of RAM and NVRAM memory available for C program use depends on the module type and if Batch 90™ programs are present. The *EDIT FORMAT SPECIFICATIONS* screen (See Figure 4-6) requests the following information:

Total RAM allocated for C

This field sets the total RAM memory (in bytes) used for C programming and related files.

Total NVRAM allocated for C

This request appears when no BATCH programs are used. This field sets the total NVRAM memory (in bytes) used for C programming and related files.

or

Total NVRAM allocated for C and BATCH

This request appears when Batch 90 programs are used in the same module as C programs. This field sets the total NVRAM memory (in bytes) used for both C and Batch 90 programs.

™ bATCH 90 is a registered trademark of Elsag Bailey Process Automation.

```
                 EDIT FORMAT SPECIFICATIONS

        Total RAM allocated for C            200000
        Total NVRAM allocated for C          50000
        Number of data files for C           10

                          RAM              NVRAM

         Available to C   1564600          420020
         Formatted for C   200000           50000
         Required by C      83960            2830
         Unused            116032           47170

Alt-H Help
```

TPS0241A

Figure 4-6. Edit Format Specifications Screen

Number of data files for C

This request appears when no Batch 90 programs are used. This field specifies the maximum number of data files in the module directory.

or

Number of data files for C and BATCH

This request appears when Batch 90 programs are used in the same module as C programs. This field specifies the maximum number of data files in the module directory.

Set these three parameters to allow for program growth since a module format command will need to be issued when any one of the three change. Note that the CUP program rounds the C user RAM memory requirements up to the nearest 64 kilobyte increment and then adds its own internal RAM and NVRAM memory requirements. This rounding allows the C system RAM memory to start on an even 64 kilobyte increment.

As the C specifications change, the memory usage table displayed on several of the screens will automatically be updated. Use these totals as a guide to ensure that enough RAM and NVRAM memory are allocated to cover the needs of the C program. It is important to note that memory specifications for sections such as code, idata, and udata must be greater than or equal to the actual amounts of each contained in the C program.

BUILD LINKER COMMAND FILE

After correctly entering the C specifications and memory allocations, select *C Build Linker Command File* from the *CUP MAIN MENU*. This will cause a linker command file (***.CMD*** file extension) to be created using the C specification information. A memory usage table will appear on screen before building the ***.CMD*** file. If any C specification information is missing or out of the valid range, the ***.CMD*** file will not be created. All errors must be corrected before further progress is allowed. If there are no errors, a ***.CMD*** file is created.

INVOKING THE LINKER PROGRAM

After creating a linker command file, select *D Invoke Linker* from the *CUP MAIN MENU*. When invoking the Microtec linker program, a memory map file (***.MAP*** file extension) and download file (***.LMS*** file extension) of the C program are created. After the linker returns from a link, the memory map of the module displays on screen. This map contains the addresses of various C program elements. If the link failed, the map will list the errors.

As an alternative, the linker may be invoked from the MS-DOS prompt by typing the following:

MCL *file_name* **Enter**

Where:

file_name is the name of the linker command file (***.CMD*** file extension) built by the CUP program. This name will be the same as the C specification file (***.CSP*** file extension) name.

MODULE MANAGEMENT

Selecting *E Module Management* from the *CUP MAIN MENU* causes the *MODULE MANAGEMENT* menu (see Figure 4-7) to appear. Explanations of the seven options follow.

Setting the Module Mode to Configure

A CONFIGURE mode places the module specified into configure mode. This mode is necessary for downloading a C program, formatting the module, adding blocks, deleting blocks, or creating module data files.

Setting the Module Mode to Execute

B EXECUTE mode places the module specified into execute mode. After placing the module in this mode, the C program

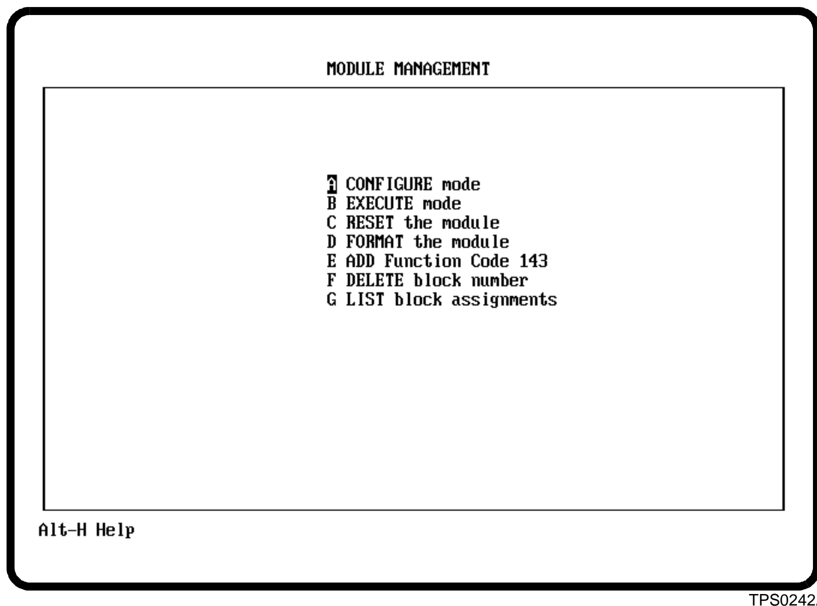


Figure 4-7. Module Management Menu

currently in the module begins execution at the specified entry points.

Resetting the Module

C RESET the module is the software equivalent of pressing the hardware reset switch of the target module.

Formatting the Module

D FORMAT the module installs the current C and Batch 90 (if applicable) memory allocations into the specified module.

NOTE: This command causes all C program, Batch 90 program (if applicable), and function code configurations in the module to be erased.

Adding Function Code 143 to the Configuration

E ADD Function Code 143 adds function code 143 to a specified block in the module. Function code 143 executes the C program located in that function block segment. This option does not permit the altering of any block specification. Change block specifications with the CAD/TXT Personal Computer Software Computer-Aided Drawing/Text program, NCTM01 Configuration and Tuning Module, or Type CTT02 Configuration and Tuning Terminal.

Erasing a Function Block

F DELETE block number deletes a configured function block in the specified module. This command complements the *E ADD Function Code 143* command.

Listing Function Block Assignments

G LIST block assignments lists the configured function blocks and corresponding function codes of the specified module.

C PROGRAM MANAGEMENT

Once a C program links successfully, it can be transferred from the work station to the target module. When selecting *F C Program Management* from the *CUP MAIN MENU*, the *C PROGRAM MANAGEMENT* menu appears. See Figure 4-8. An explanation of each menu option follows.

Transferring a C Program from the Work Station to the Module

A Download C Program From EWS to Module causes the CUP program to compare the C program memory information and the current module memory map. If the memory information matches, the CUP program will display a memory summary screen and download the C program to the module. If there are any discrepancies between the current module memory map and the C program memory information, the CUP program will not allow the download. Most discrepancies can be corrected by editing the C specifications or reformatting the specified

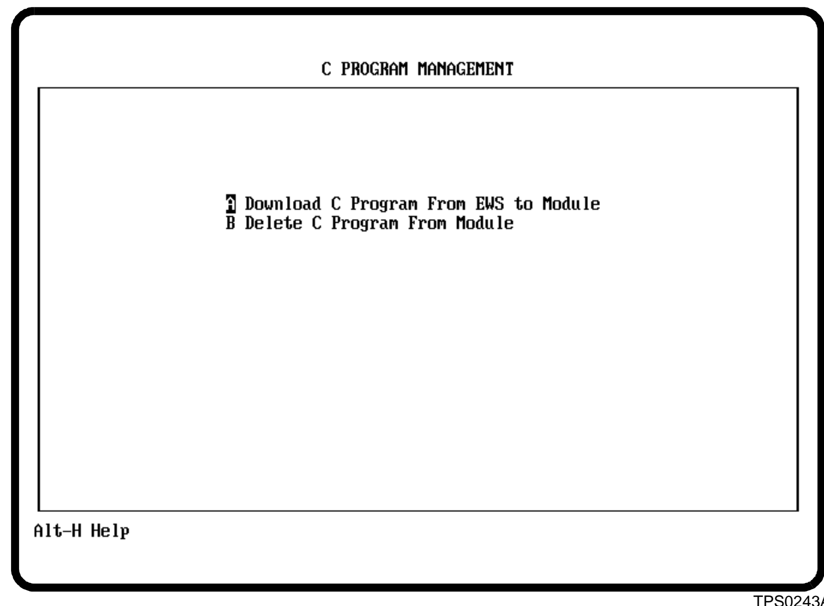


Figure 4-8. C Program Management Menu

module. The module must be in configure mode for the C program to download. This transfer process creates the three NVRAM C program files (file numbers 32765, 32766, and 32767) in the directory of the specified module.

Deleting C Programs from the Module

B Delete C Program From Module removes a C program from the specified module by deleting the three C program files. Always delete the function code 143 (in the module function block configuration) corresponding to the deleted C program. If a configured module (containing function code 143) is placed into execute mode when no C program files are in the module directory, an invoke C error appears.

DATA FILE MANAGEMENT

It is important to know that NVRAM files created for use in a module differ from files created for use in a work station or those found on an IMMFP03 hard drive system. In the NVRAM file system, there can be no subdirectories and files must have numeric file names. In IMMFP03 modules equipped with a hard drive, both a NVRAM and hard drive file system exist. The hard drive file system can have subdirectories and must have alphabetic names. The name, alphabetic or numeric, used in a C program routine such as OPEN is how the destination (NVRAM or hard drive) of the file is determined. Both file systems have access levels (read only, write only, read and write).

The CUP program provides an interactive shell that permits the execution of file management commands. Refer to Table 4-1 for a complete list of commands. Some of these commands apply to IMMFP03 module hard drive file systems only. Other commands apply to NVRAM file systems only. The remaining commands apply to both file systems.

Table 4-1. Data File Management Commands

Command	Description
BKUP[1]	Send a backup copy of the primary module hard drive to the redundant module hard drive.
CALL	Execute a shell command file. The command file is an ASCII file that can contain any number of data file commands found in this section. Example: **CALL** *file* where *file* is the DOS path and file name.
CD[1]	Change the current module hard drive directory. Example: **CD** [*path*] where *path* is the new directory path specification.
CFG	Place the target module into configure mode.
CLS	Clear the shell display window.
CP[1]	Initiate a module hard drive copy operation. Example: **CP** *src* *dst* where *src* is the source path and file name. *dst* is the destination path and file name.

Table 4-1. Data File Management Commands (continued)

Command	Description
CR[1]	Create a module hard drive file. Example: **CR** *file* where *file* is the path and file name to be created.
CRN[2]	Create a module NVRAM file. Example: **CRN** *file* [*access*] [*check*] where *file* is the file number (allowable numbers from 1 to 32,759). *access* is the write access code (0 = no restrictions, 1 = local writes only, 2 = no writes allowed). 0 is the default value. *check* is checkpoint inhibit code (0 = no, 1 = yes). 0 is the default value.
CWD[1]	Get the current module hard drive directory.
DF[1]	Get the number of free hard drive blocks.
DOS	Invoke a DOS shell.
EXE	Place the target module into execute mode.
FIXN[2]	Attempt to recover a bad NVRAM file. Example: **FIXN** [*file*] where *file* is the number of the bad NVRAM file.
FSCK[1]	Check the module hard drive system. Example: **FSCK** *type name1 name2*... where *type* is the type of check to perform. *name* is the file or device name. Refer to Appendix A for more information about this command.
GET[1]	Transfer a module hard drive file to the work station. Example: **GET** *module ews* [*count*] [*offset*] where *module* is the hard drive file path and file name. *ews* is the work station path and file name. *count* is the number of bytes to transfer (default = all). *offset* is the file offset at which to begin transfer (default = 0).
GETN[2]	Transfer a module NVRAM file to the work station. Example: **GETN** *module ews* [*count*] [*offset*] where *module* is the NVRAM file number. *ews* is the work station path and file name. *count* is the number of bytes to transfer (default = all). *offset* is the file offset at which to begin transfer (default = 0).
HELP	Get help on a command. Example: **HELP** [*command*] where *command* is the subject of the inquiry. Entering **HELP** without the command field will display a complete list of file management commands.
LS[1]	List the module hard drive directory. Example: **LS** [*path*] where *path* is the path of the directory to be listed.
LSN[2]	List the module NVRAM directory.
MAP	Display the memory map of the module.
MD	Make a module hard drive directory. Example: **MD** *path* where *path* is the path and name of the new directory.
MKFS[1]	Make the hard drive file system. Example: **MKFS** *num_inodes num_zones num_bufs* [*sync_time*] where *num_inodes* is the number of inode (files and/or directories) blocks. *num_zones* is the number of zone blocks. *num_bufs* is the number of cache buffers. *sync_time* is the time between file system syncs in seconds. Refer to Appendix B for more information about this command.
MOD	Select the target module. Example: **MOD** [*address*] where *address* is the module address.
NOP	Null operation.
PUT[1]	Transfer a work station file to the module hard drive. Example: **PUT** *ews module* [*count*] [*offset*] where *ews* is the path and file name of the work station file. *module* is the path and file name of the module hard drive file. *count* is the number of bytes to transfer (default = all). *offset* is the file offset at which to begin transfer (default = 0).

Table 4-1. Data File Management Commands (continued)

Command	Description
PUTN[2]	Transfer a work station file to the module NVRAM. Example: **PUTN** *ews module* [*count*] [*offset*] where *ews* is the path and file name of the work station file. *module* is the number of the module NVRAM file. *count* is the number of bytes to transfer (default = all). *offset* is the file offset at which to begin transfer (default = 0).
QUIT	Exit the data file command shell.
RD[1]	Remove a module hard drive directory. Example: **RD** *dir* where *dir* is the path and name of the directory to be removed.
REWIND	Restart the execution of a shell command file.The rewind command is used within a shell command file only. It causes execution of the command file to restart at the beginning. Refer to the call command.
RM[1]	Remove a module hard drive file. Example: **RM** *file* where *file* is the path and file name of the file to be removed.
RMN[2]	Remove a module NVRAM file. Example: **RMN** *file* where *file* is the number of the file to be removed.
RST	Reset the target module.
SLEEP	Wait a specified number of seconds. Example: **SLEEP** *seconds* where *seconds* is the length of the delay. This command is used within a shell command file only. It suspends execution of the command file for the specified number of seconds. Refer to the call command.
SYNC	Set hard drive sync time. Example: **SYNC** seconds where *seconds* is the number of seconds between syncs.

NOTES:

1. This command functions in hard drive file systems only.
2. This command functions in NVRAM file systems only.
3. Brackets ([..]) denote an optional parameter. All optional parameters in the same command must be specified when changing even one of the parameters.

To perform data file management commands, select *G Data File Management* from the *CUP MAIN MENU*. This causes the *DATA FILE MANAGEMENT* screen to appear. See Figure 4-9. Enter the appropriate information in the *Loop Address*, *PCU Address*, and *Module Address* fields on this screen and press **F10** At this point, the CUP program provides an interactive shell for data file management. The shell appears as a large empty box with the word *Module* followed by a number in the upper left corner. See Figure 4-10 for an example shell screen. The module specified is the target module. If the desired command is known, enter the command and press **Enter**. If the desired command is not known, type **HELP** and press **Enter**. This will display the complete list of available commands (see Figure 4-10). Refer to Table 4-1 or type **HELP** [*command*], and press **Enter** for further information about a particular command.

MODEM SETUP

One feature of the C Utility Program is the capability of establishing a remote modem communication link. Using this link,

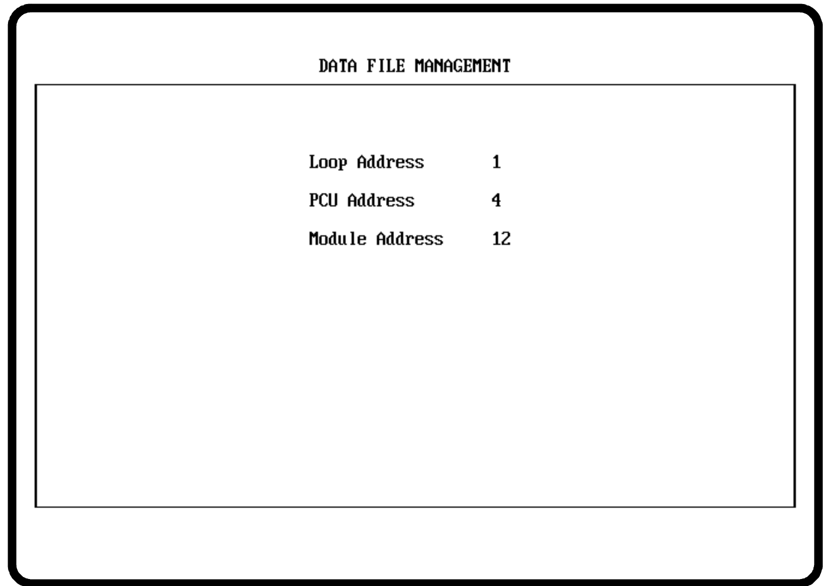


Figure 4-9. Data File Management Screen

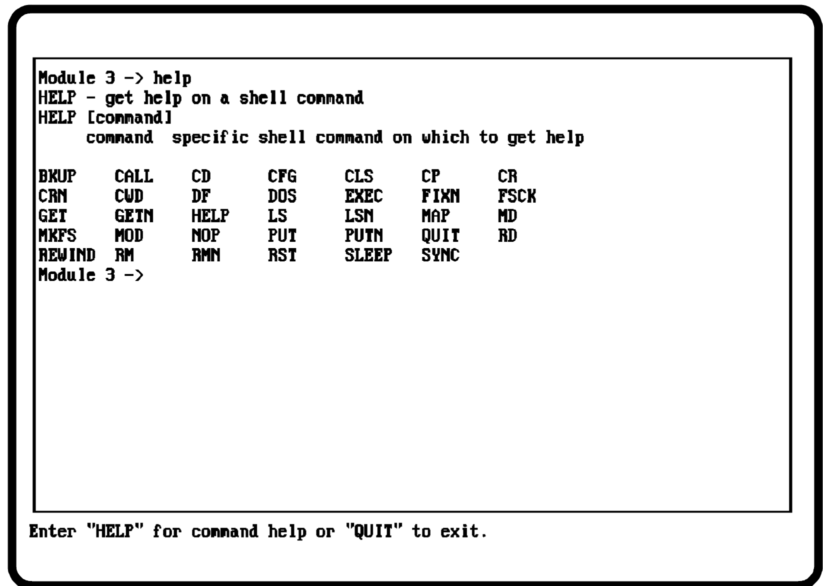


Figure 4-10. Example Data File Management Shell

an INFI 90 system can be accessed from a remote work station (containing the CUP program) and still have the CUP program function as if connected directly to the system. Complete modem support resides in the CUP program eliminating external communication software requirements. The requirements for connecting a remote work station (loaded with the

CUP program) to an INFI 90 system through modems are as follows:

1. Two Hayes® compatible modems are required.

2. The module or interface connecting the work station to the INFI 90 system and both modems must be set to the same baud rate. Baud rates from 300 to 9600 are supported by the INFI 90 system.

3. The modem connected to the work station has the following additional requirements.

   a. Hayes compatible command recognition must be enabled.

   b. The result codes of the Hayes commands must be sent back to the work station as English words.

   c. Character echo must be disabled.

   d. The RS-232-C carrier detect signal must be set to determine if a carrier signal is coming from the INFI 90 system. Note that the work station uses pin 8 of the DB-25 connector for this signal.

   e. Support for the RS-232-C document transmittal record signal should be set to allow the modem to hang up after completing a call. Please note that the station uses pin 20 of the DB-25 connector for this signal.

4. The modem connected to the INFI 90 system has the following additional requirements.

   a. Automatic answering must be enabled.

   b. The modem must be set to ignore RS-232-C document transmittal record signals (pin 20 of the DB-25 connector).

   c. No modem result codes must be sent back to the INFI 90 system.

   d. Character echo must be enabled.

   e. The RS-232-C carrier detect signal must be set to determine if a carrier signal is coming from the work station. Note that the work station uses pin 8 of the DB-25 connector for this signal.

® Hayes is a registered trademark of Hayes Microcomputer Products Inc.

f. The RS-232-C cable connecting the modem to the INFI 90 system must be wired in a null modem configuration.

Refer to the instruction manual for the modem used for exact setup procedures. After making all hardware connections, modem setup can take place. Selecting *H Modem Setup* from the *CUP MAIN MENU* causes the *MODEM SETUP* menu to appear. See Figure 4-11. An explanation of each menu option follows.

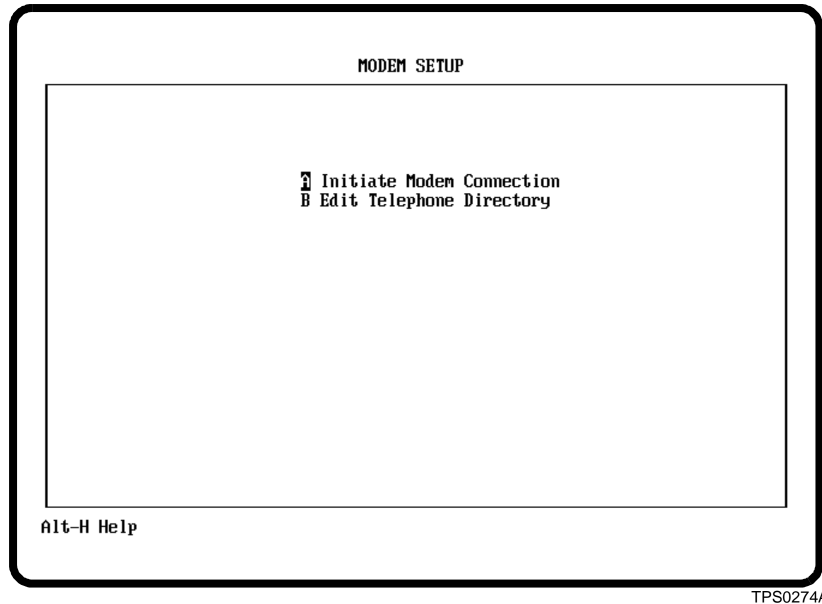


Figure 4-11. Modem Setup Menu

Initiating Modem Connection

A Initiate Modem Connection establishes a communication link to a remote INFI 90 system using the specified information. When the *MODEM INITIALIZATION* screen appears, specify the telephone number, communication port number (1 or 2), and baud rate (300, 1200, 2400, or 9600). See Figure 4-12. When in the *Telephone Number* field, pressing **F8** will cause the telephone directory to be displayed. Select a number or exit the directory and type in the desired number.

Editing the Telephone Directory

B Edit Telephone Directory allows editing of the telephone directory database. See Figure 4-13. Use this database to store frequently called modem numbers. These telephone numbers can be recalled when initiating a modem connection.

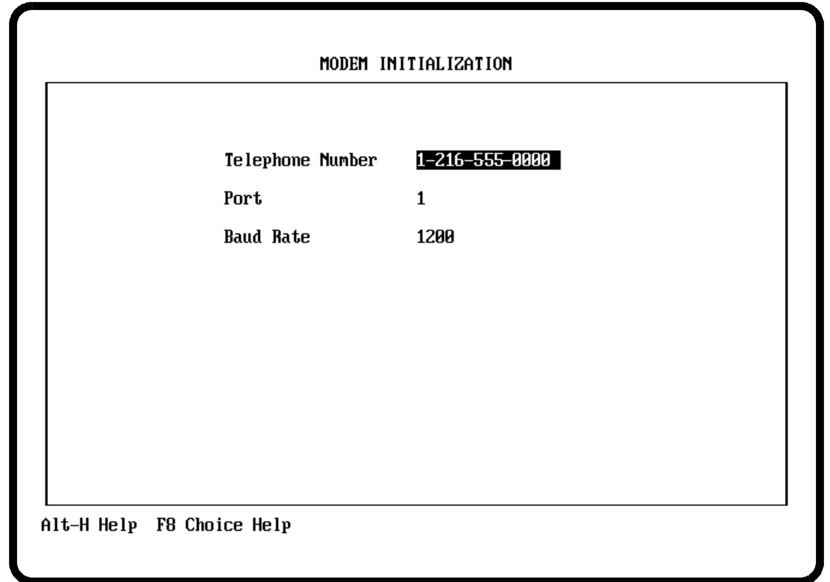


Figure 4-12. Modem Initialization Screen

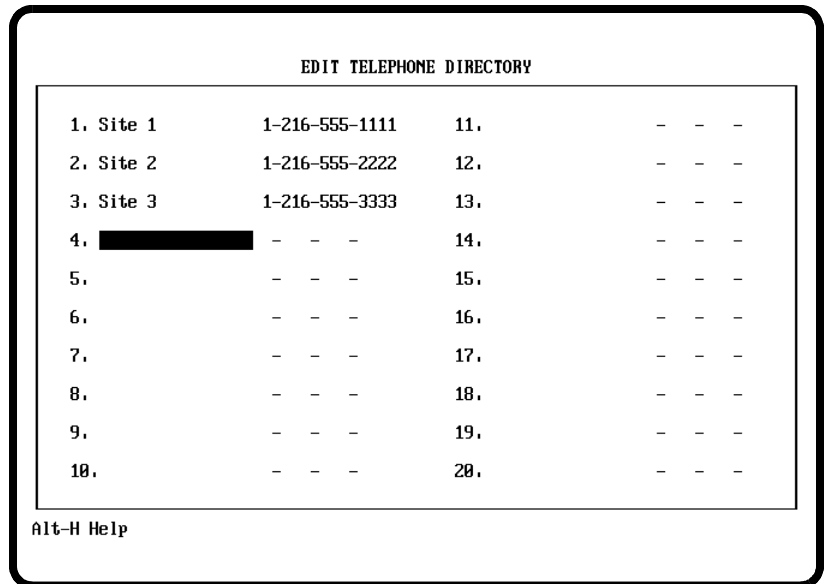


Figure 4-13. Edit Telephone Directory Screen

CUP ERROR MESSAGES

Table 4-2 lists the error messages unique to the C Utility Program and a brief explanation of each. Please note that a list of error messages from INFI 90/Network 90 communication interfaces exists in the interface documentation.

Table 4-2. CUP Error Messages

Error Message	Explanation	Corrective Action
Access denied	Access to the specified file is denied.	Change the permission flags or change the path.
Bad file descriptor	The file descriptor passed to a function was out of range or invalid.	Verify the file descriptor is an acceptable value.
Buffers too small	The module file buffer is too small for the accessed module file. Each buffer should be larger than the largest module file to be accessed.	The size of each buffer should be increased.
Can't create module file	There is no space left for another entry in the module file list.	The number of files in the module directory must be increased.
Cannot find entry point name in map file	While downloading the C program, CUP could not find the entry point name specified.	Make sure that this entry point is correct (i.e., spelling and lower/upper case) and retry.
Cannot open directory	An illegal file name or directory path was specified.	Insert the valid file name or directory path.
Cannot remove directory	The specified directory cannot be removed, or a directory path was specified when attempting to remove a file.	Correct the path name.
Copy command - cannot copy file onto itself	Attempted to copy a file on top of itself.	Change the destination file name to a name other than that of the source file.
Copy command - arguments not valid	Unidentified path in a copy command (This error code should never be seen).	Check the copy command parameters for the appropriate path names. Document this error and consult the technical support department of Bailey Controls Company.
Directory not empty	The contents of a directory have not been removed prior to the removal command.	Delete all files in the directory.
Directory problems detected in getcwd	The (.) or (..) entries in a directory structure are corrupt.	Execute the FSCK command with the repair option enabled. Document this error and consult the technical support department of Bailey Controls Company.
Entry not a directory	The specified directory name is not the name of an existing directory.	Verify the directory name is valid.
Error found in module file	A module file contains a data error.	Correct the error.
Error initializing target files	CUP could not create the module target files while downloading the C program.	Make sure that the module is in configure mode and is formatted with the current memory specifications.
Error occurred due to a parameter being a directory	A file was given the name of an existing directory or an open was attempted on a directory entry with write permission given.	Check the syntax of either the **cp** command or the open command.
Error transferring map/code/data section	CUP encountered an error while downloading one of these three C program files.	Ensure that the module is properly formatted with the current memory specifications and there is enough NVRAM memory space allocated to store these three files.

Table 4-2. CUP Error Messages (continued)

Error Message	Explanation	Corrective Action
File already exists	Attempted to create a file that already exist.	Remove the file that already exists or change the name of the file to be created.
File does not exist	File specified in the RM command does not exist.	Verify the correct file name is specified in the RM command.
File is too large	The size of the specified file exceeds the maximum file size.	Reduce the file size.
Hard drive backup operation failed	Undetermined error occurred during the BKUP operation. The maximum number of retries is exceeded.	Check connections between the MFP module and the MPI module and retry the command.
Hard drive write error	Error writing to the hard drive of the backup module during the BKUP command.	Check connections between the MFP module and the MPI module and retry the command.
Invalid argument	An argument in a command or function is invalid.	Check the arguments and parameters in the command for errors.
Invalid seek	The SEEK that was attemped was out of range or occurred on an invalid file.	Document the error and consult the technical support department of Bailey Controls Company.
Maximum directory level exceeded	The maximum number of subdirectory levels is exceeded.	Limit subdirectory levels to four deep.
Memory specifications and map do not agree	The current memory specifications do not match those in the map file produced for the C program by the linker.	Relink the C program to correct this error.
Memory specifications conflict with .LMS file	The current memory specifications conflict with the C program object file produced by the linker.	Relink the C program to correct this error.
Memory specifications conflict with module	The CUP memory format specifications do not match the current module settings.	Reformat the module or change the CUP specifications accordingly.
Module file does not exist	The specified module file is not in the directory.	Create the file.
Module file is write protected	The module file is write protected.	Change the write access level or write to another file.
Module file not opened	A read or write was attempted on a module file that is not opened.	Open the file.
Module file offset out of range	A read was attempted at an offset beyond the end of a module file.	Correct the file offset value.
Module file system busy	Module is currently updating the desired file.	Repeat the command after a short delay.
Module file system error	The module file directory is corrupt, or there is no free space left for file storage.	Increase the space allotted or repair the corrupt directory.
Name too long	The specified path name is too long.	Correct the path name.
No file buffer available	All the available module file buffers are in use.	Increase the number of module file buffers or repeat the command after a short delay.

Table 4-2. CUP Error Messages (continued)

Error Message	Explanation	Corrective Action
No hard drive on primary module	The BKUP command was issued and the primary module has no hard drive or it is not bootable.	Check the connections between the MFP module and the MPI module. Verify the hard drive is bootable by issuing the MKFS command.
No hard drive on redundant module	The BKUP command was issued and the secondary module has no hard drive or it is not bootable.	Check the connections between the MFP module and the MPI module. Verify the hard drive is bootable by issuing the MKFS command.
No space left on the device	No blocks are left on the device.	Remove files from the disk that are no longer needed.
No such file or directory	The specified file or directory name does not exist.	Verify the proper file or directory name was entered.
Path not a directory	The specified path is not a valid directory path.	Correct the directory path name.
Read only file system	The file system for the device is READ ONLY.	Document the error and consult the technical support department of Bailey Controls Company.
Result is too large	The path for the current working directory is too long.	Reduce the path length by shortening the path or path names.
Too many files to execute	Too many files found in the directory for the command to hold.	Move some files to another directory.
Too many open files	Too many files are open at one time.	Close some of the files that are in use and try again.
Write failure	Write to a file failed.	Check disk space and parameters of the write command.
Wrong mode for operation	A module file create or write was attempted while the module was in execute mode.	Place the module in configure mode.

SECTION 5 - MODULE SPECIFIC FUNCTIONS

INTRODUCTION

Listed in this section are the runtime support functions unique to MFP and MFC modules.

bin

PURPOSE:

This function inputs fixed or configured block output values to the C program.

FORMAT:

```
status = bin(blockno, bufptr);
long int blockno;                    block number of input
struct fbbuf *bufptr;                pointer to buffer structure
long int status;                     return status:
                                     0 = success
                                     -1 = block does not exist
```

REMARKS:

This function inputs fixed or configured block output values to the C program. After this function is called, the fbbuf buffer contains the data type, quality definition, and output value of the block. The buffer structure fbbuf is a two part structure. The info part of the structure defines the block characteristics. The fobval part of the structure defines the block value. The fbbuf structure is defined in the ***n90.h*** file as follows:

```
struct fbbuf
{
    struct                          /* block (FOB) characteristics */
    {
        unsigned type  :  8;        /* data type of output value */
                                    /* (BOOLEAN, I1, I2, or R4) */
        unsigned qdef  :  8;        /* nonzero if quality defined */
                                    /* (NOQUAL or QUAL) */
        unsigned fcdata  :  8;      /* code specific data */
        unsigned q  :  1;           /* quality */
        unsigned ha  :  1;          /* high alarm/Boolean alarm */
        unsigned la  :  1;          /* low alarm */
        unsigned hdev  :  1;        /* high device alarm */
        unsigned ldev  :  1;        /* low device alarm */
        unsigned pad  :  3;         /* unused bits */
    }  info;
    union                           /* block output (FOB) value */
    {
        struct
        {
            unsigned pad1  :  7;
            unsigned val  :  1;     /* Boolean value */
            unsigned pad2  :  8;
        } bo;
        struct
        {
```

bin

continued

```
                        unsigned char val;          /* I1 value */
                        char pad;
                } i1;
                short i2;                           /* I2 value */
                float r4;                           /* REAL 4 value */
        } fobval;
};
```

The constants BOOLEAN, I1, I2, R4 (fbbuf.info.type), NOQUAL, and QUAL (fbbuf.info.qdef) are also defined in the ***n90.h*** file for C program use.

See also: ***blk_wrt***, ***bout***, ***dadig_in***, ***dasin***

EXAMPLE:

```
#include <n90.h>
void main(void)
{
        long int hours, minutes, seconds, time_sync, status;
        struct fbbuf buffer;

        /*
        Get hours, minutes, seconds and time_sync flag from the Extended Executive
        block and display these values on an attached terminal:
        */
        /* get hours */
        status = bin(20, &buffer);
        if (status) return;
        hours = (long int) buffer.fobval.r4;

        /* get minutes */
        status = bin(21, &buffer);
        if (status) return;
        minutes = (long int) buffer.fobval.r4;

        /* get seconds */
        status = bin(22, &buffer);
        if (status) return;
        seconds = (long int) buffer.fobval.r4;
        /* get time sync flag */
        status = bin(23, &buffer);
        if (status) return;
        time_sync = (long int) buffer.fobval.bo.val;

        /* print results to terminal */
        printf("System Time: %02d:%02d:%02d\n",hours, minutes,
        seconds);
        printf("Time Sync Flag: %s\n", (time_sync) ? "NO" : "YES");

}
```

blk_wrt

PURPOSE: This function writes to exception report function blocks.

FORMAT:

```
status = blk_wrt(bufptr);
struct bs_ *bufptr;          pointer to buffer structure
long int status:             return status;
                              0 = success
                             -1 = block does not exist
                             -2 = function code mismatch
                             -3 = block does not allow write
                             >0 = operations not performed
```

REMARKS: This function writes directly to exception report blocks (function codes 30 and 45). Externally sourced timestamps can also be written to the INFI 90 system by the C program. Function code 30 and 45 must have function block input (specification one) configured to a number two for it to be considered program driven. The output of the exception report block will be zero with bad quality until the blk_wrt() function is called. The block still performs the alarming and exception report operations normally, but the input, quality, and timestamp are read from the C program supplied buffer. Data is kept internally in the function blocks after successful completion of the blk_wrt() function. Exception reports are not generated when the blk_wrt() function is executed. Exception reports are only generated by the normal exception report mechanism.

Each function code that supports the blk_wrt() function has a unique buffer structure that contains two parts. The bs_ part of the structure defines the block number, function code, and operation. The other part of the structure contains function code specific information. These parts are defined in the ***blk_wrt.h*** file as follows:

```
struct bs_                          /* structure for the common section */
{
      long block_no;                /* block number */
      short fun_code;               /* function code */
      unsigned long operations;     /* operations */
};
```

The operations field is set by performing a bitwise OR with the desired operation values as defined in the ***blk_wrt.h*** file:

Operation Value	Description
BLK_WRT_TS	Timestamp operation
BLK_WRT_QA	Set/clear quality/alarm states
BLK_WRT_V	Set block output value

blk_wrt

continued

Function code 30 structure as defined in the ***blk_wrt.h*** file:

```
struct bs_fc30
{
        struct bs_  bs;                          /* common section structure */
        unsigned char qa_on;                     /* quality/alarm on control */
        unsigned char qa_off;                    /* quality/alarm off control */
        float val;                               /* analog value */
        unsigned long timestamp;                 /* timestamp value */
};
```

Function code 45 structure as defined in the ***blk_wrt.h*** file:

```
struct bs_fc45
{
        struct bs_  bs;                          /* common section structure */
        unsigned char qa_on;                     /* quality/alarm on control */
        unsigned char qa_off;                    /* quality/alarm off control */
        unsigned char val;                       /* value */
        unsigned char unused;                    /* reserved for future use */
        unsigned long timestamp;                 /* timestamp value */
};
```

The qa_on and qa_off fields are used to control the quality and alarm parameters of the function block. These fields are set by performing a bitwise OR with the desired operation values as defined in the ***blk_wrt.h*** file:

Value (qa_on)	Value (qa_off)	Description
BLK_Q_0	BLK_Q_0	No operation
BLK_Q_0	BLK_Q_1	Clear quality
BLK_Q_1	BLK_Q_0	Set quality
BLK_Q_1	BLK_Q_1	Enable quality
BLK_HA_0	BLK_HA_0	No operation
BLK_HA_0	BLK_HA_1	Clear high alarm
BLK_HA_1	BLK_HA_0	Set high alarm
BLK_HA_1	BLK_HA_1	Enable high alarm
BLK_LA_0	BLK_LA_0	No operation
BLK_LA_0	BLK_LA_1	Clear low alarm
BLK_HA_1	BLK_LA_0	Set low alarm
BLK_HA_1	BLK_HA_1	Enable low alarm
BLK_HD_0	BLK_HD_0	No operation
BLK_HD_0	BLK_HD_1	Clear high deviation
BLK_HD_1	BLK_HD_0	Set high deviation
BLK_HD_1	BLK_HD_1	Enable high deviation
BLK_LD_0	BLK_LD_0	No operation

blk_wrt

continued

Value (qa_on)	Value (qa_off)	Description
BLK_LD_0	BLK_LD_1	Clear low deviation
BLK_LD_1	BLK_LD_0	Set low deviation
BLK_LD_1	BLK_LD_1	Enable low deviation

See also: ***bin***, ***bout***, ***dadig_in***, ***dasin***

EXAMPLE:

```
#include <blk_wrt.h>
void main(void)
{
	long int status;			/* blk_wrt() return status */
	float f = 23.76;			/* miscellaneous program variable */
	struct bs_fc30 buffer;		/* buffer for blk_wrt() parameters */

	/*
	Write value 3.14159 to function code 30 at block 100, assert
	quality bad, and clear the high alarm:
	*/
	buffer.bs.fun_code = 30;
	buffer.bs.block_no = 100;
	buffer.bs.operations = BLK_WRT_V | BLK_WRT_QA;
	buffer.val = 3.14159;
	buffer.qa_on = BLK_Q_1 | BLK_HA_0;
	buffer.qa_off = BLK_Q_0 | BLK_HA_1
	status = blk_wrt(&buffer);
	if (status) return;

	/*
	Write value f to function code 30 at block 200, and set the low
	alarm:
	*/
	buffer.bs.fun_code = 30;
	buffer.bs.block_no = 200;
	buffer.bs.operations = BLK_WRT_V | BLK_WRT_QA;
	buffer.val = f;
	buffer.qa_on = BLK_LA_1;
	buffer.qa_off = BLK_LA_0;
	status = blk_wrt(&buffer);
	if (status) return;
}
```

bout

PURPOSE: This function outputs a value to a function block.

FORMAT:

```
status = bout(blockno, bufptr);
      long int blockno;              block number for output
      struct fbbuf *bufptr;          pointer to buffer structure
      long int status;               return status:
                                      0 = success
                                      1 = overflow occurred
                                     -1 = error
```

REMARKS: This function outputs a value and optional quality/alarm status to a function block. The data type, quality, and alarm parameters are provided by the C program via the fbbuf buffer structure. The output function block must be an output buffer function code (function code 93, 94, 137, or 138). An error occurs when the block number is out of range, the output function block does not exist, the output function block is not an output buffer function code, or the data type specified does not match the output buffer function code type.

Overflow can occur for block outputs that are of type REAL. This happens because the floating point representation used internally by the module differs from C program floating point representations. If the block output value is too large to be represented, the largest value of the same sign is output and the return status of the bout() function is set to one.

The buffer structure fbbuf is a two part structure. The info part of the structure defines the block characteristics. The fobval part of the structure defines the block value. The pad element of fbbuf must be set to zero for proper operation. The fbbuf structure is defined in the ***n90.h*** file. See the ***bin*** function for details.

See also: ***bin***, ***blk_wrt***, ***dadig_in***, ***dasin***

EXAMPLE:

```
#include  <n90.h>

void main(void)
{
      long int status;
      struct fbbuf buffer;

      /*
      Output a real value with quality to block 200, which is configured with a function
      code 137 (output buffer):
      */
      buffer.info.type = R4;                /* output is type REAL */
      buffer.info.qdef = QUAL;              /* output has quality */
      buffer.info.q = 0;                    /* output quality is good */
      buffer.info.ha = 0;                   /* no high alarm */
      buffer.info.la = 0;                   /* no low alarm */
```

bout

continued

```
        buffer.info.hdev = 0;                       /* no high device */
        buffer.info.ldev = 0;                       /* no low device */
        buffer.info.pad = 0;                        /* must zero pad element */
        buffer.fobval.bo.val = 1;                   /* set output value */

        status = bout(200, &buffer);

        /*
        Output a Boolean value without quality to block 300, which is configured with a
        function code 94 (Output Buffer):
        */
        buffer.info.type = BOOLEAN;                 /* output is type boolean */
        buffer.info.qdef = NOQUAL;                  /* output has no quality */
        buffer.info.ha = 0;                         /* no high alarm */
        buffer.info.la = 0;                         /* no low alarm */
        buffer.info.hdev = 0;                       /* no high device */
        buffer.info.ldev = 0;                       /* no low device */
        buffer.info.pad = 0;                        /* must zero pad element */
        buffer.fobval.r4 = 23.6391;                 /* set output value */

        status = bout(300, &buffer);

}
```

checkpoint_file

PURPOSE: This function requests a hard drive or NVRAM file checkpoint operation.

FORMAT:

status = checkpoint_file(file_name,mode);	
char *file_name:	file name (ASCII character string)
short int mode;	error action: 0 = immediate return 1 = wait until checkpoint complete
long int status;	return status: 0 = success -1 = no buffer available -2 = file too big for buffer -3 = file read error -4 = problem with backup -5 = not able to checkpoint file

REMARKS: This function requests that a file be copied to the backup module in a redundant pair configuration. This copy operation allows changes in file data of the primary module to be updated in the backup module. The mode parameter specifies the action to take if a checkpoint buffer error, such as no buffers currently available, occurs.

Enter file names as ASCII character strings regardless of file origin (NVRAM or hard drive). Include the complete path in the file name for hard drive files.

The primary module uses checkpoint buffers to temporarily store the file data while it is being sent to the backup module. The number of checkpoint buffers and size of each are specified via the C Utility Program. When a checkpoint request starts, the module acquires a checkpoint buffer. File data loads into the buffer and transfers to the backup module over the redundancy link. Once the data transfer is complete, the module releases the buffer. For optimum checkpoint performance the checkpoint buffer size should be greater than the largest file to checkpoint.

When using the wait until complete mode to checkpoint a file, approximately 300 bytes of C stack space are used if no checkpoint buffers are available, or the file size is larger than the checkpoint buffer size. Generally, checkpoint buffers of sufficient size and quantity should be allocated to accommodate the total number of checkpoint requests that may occur at one time. If this is not possible, increase the stack size accordingly via the C Utility Program.

When using the return immediate mode to checkpoint a file and there are no buffers available or if they are too small, the

checkpoint_file

continued

checkpoint_file function will return immediately with a zero status and no files checkpointed.

See also: ***cpfil***

EXAMPLE:

```
#define IMMEDIATE_RETURN  0
#define WAIT_UNTIL_COMPLETE  1

char filename[100];

extern long checkpoint_file();

void main(void)
{
	short mode;
	long status;

	mode = WAIT_UNTIL_COMPLETE;

	/* checkpoint hard drive file abc.txt in the directory /test */
	strcpy(filename, "\\test\\abc.txt");
	status = checkpoint_file(filename, mode);
	if (status == 0)
	{
		printf("File %s was successfully checkpointed\n",filename);
	}
	else
	{
		printf("Error checkpointing file %s\n",filename);
	}

	/* checkpoint NVRAM file 2000 */
	strcpy(filename,"2000");
	status = checkpoint_file(filename, mode);
	if (status == 0)
	{
		printf("File %s was successfully checkpointed\n",filename);
	}
	else
	{
		printf("Error checkpointing file %s\n",filename):
	}

	/* checkpoint hard drive file 2000 in the root directory */
	strcpy(filename,"\\2000");
	status = checkpoint_file(filename, mode);
	if (status == 0)
	{
		printf("File %s was successfully checkpointed\n",filename);
	}
	else
	{
		printf("Error checkpointing file %s\n",filename);
	}
}
```

cpfil

PURPOSE: This function requests a NVRAM file checkpoint operation.

FORMAT:

```
status = cpfil(mode, filehandle, fileid);
short int mode;            error action:
                           0 = immediate return
                           1 = request deferred checkpoint
and return
                           2 = wait until checkpoint complete
long int filehandle;       file handle as returned from open()
long int fileid;           file id number
long int status;           return status:
                            0 = success
                           -1 = no buffer available
                           -2 = file too big for buffer
                           -3 = file read error
                           -4 = problem with backup
                           -5 = not able to checkpoint file
```

REMARKS: This function requests that a NVRAM file be copied to the backup module in a redundant pair configuration. This copy operation allows changes in NVRAM file data of the primary module to be updated in the backup module. The mode parameter specifies the action to take if a checkpoint buffer error, such as no buffers currently available, occurs.

If the immediate return mode or wait until complete mode of checkpointing is requested, the NVRAM file to be checkpointed must be opened for read access by the C program prior to the request. Once the checkpointing is completed, the C program can close the NVRAM file. If the C program is holding the NVRAM file open for write access during the request, the checkpoint request will fail.

The C program can hold the NVRAM file open for write access during a deferred mode checkpoint request, but the NVRAM file will not be checkpointed until it is closed by the C program. This close is required to allow the checkpoint function to acquire the NVRAM file data and send it to the backup module. Once checkpointing is complete, the C program can reopen the NVRAM file for write access.

Checkpoint buffers are used by the primary module to temporarily store the NVRAM file data while it is being sent to the backup module. The number of checkpoint buffers and size of each are specified via the C Utility Program. When a checkpoint request starts, a checkpoint buffer is acquired. NVRAM file data is then loaded into the buffer and transferred to the backup module over the redundancy link. Once the data transfer is complete, the buffer is released and can be used to service another checkpoint request. For optimum checkpoint

cpfil

continued

performance it is recommended that the checkpoint buffer size be greater than the largest NVRAM file that is to be checkpointed.

If the wait until complete mode is used to checkpoint a NVRAM file, approximately 300 bytes of C stack space is used if no checkpoint buffers are available, or the NVRAM file size is larger than the checkpoint buffer size. Generally enough checkpoint buffers of sufficient size should be allocated to accommodate the total number of checkpoint requests that may occur at one time. If this is not possible, increase the stack size accordingly via the C Utility Program.

If the return immediate mode is used to checkpoint a NVRAM file and there are no buffers available or they are too small, the cpfil() function will return immediately with a zero status and no checkpointing will be performed.

See also: ***checkpoint_file***

EXAMPLE:

```
#include <io.h>
#include <fcntl.h>

#define DATA_FILE "1000"
#define MESSAGE "This is some file data."

void main(void)
{
      long int filehandle, status;

      /* create a NVRAM file with some data */
      filehandle = open(DATA_FILE, O_WRONLY | O_CREAT | O_TRUNC);
      write(filehandle, MESSAGE, strlen(MESSAGE));
      close(filehandle);

      /* checkpoint the NVRAM file using the wait until complete mode */
      filehandle = open(DATA_FILE, O_RDONLY);
      status = cpfil(2, filehandle, atoi(DATA_FILE));
      close(filehandle);

}
```

dadig_in

PURPOSE: This function reads information from a data acquisition digital block.

FORMAT:

```
status = dadig_in(blockno, bufptr);
long int blockno;                       block number of input
struct dadig_buf *bufptr;               pointer to buffer structure
long int status;                        return status:
                                         0 = success
                                        -1 = block does not exist
```

REMARKS: This function reads the internal structure of a data acquisition digital block (function code 211 or 212). After successful completion, the dadig buffer contains the same information that is sent to the loop in an exception report by the block. The dadig_buf structure is defined in the ***das.h*** file as follows:

```
struct dadig_buf
{
    unsigned char timestamp[6];             /* latched timestamp */
                                            /* (6 byte integer) */
    unsigned quality  : 1;                  /* quality/hardware failure */
    unsigned alarm  : 1;                    /* alarm */
    unsigned time_in_alarm  : 1;            /* time in alarm re-alarm */
    unsigned alarm_suppressed  : 1;         /* alarms suppressed */
    unsigned output_suspect  : 1;           /* output suspect */
                                            /* out of range */
    unsigned no_report  : 1;                /* no report/off scan */
    unsigned red_tagged  : 1;               /* red-tagged */
    unsigned output_value  : 1;             /* output value */
    unsigned ex_stat_latched  : 1;          /* extended status latched */
    unsigned quality_override  : 1;         /* quality override */
    unsigned set_permissive  : 1;           /* set permissive */
                                            /* (state of S2) */
    unsigned pri_input_sel  : 1;            /* primary input select */
                                            /* (S1 bit 1) */
    unsigned alt_input_sel  : 1;            /* alternate input select */
                                            /* (S1 bit 0) */
    unsigned spare1  : 1;                   /* spare bit 1 */
    unsigned spare2  : 1;                   /* spare bit 2 */
    unsigned spare3  : 1;                   /* spare bit 3 */
};
```

See also: ***bin***, ***blk_wrt***, ***bout***, ***dasin***

dadig_in

continued

EXAMPLE:

```
#include <das.h>

void main(void)
{
        struct dadig_buf buffer;
        long int status;
        static char *yes = "YES";
        static char *no = "NO";

        /* get the internals from function code 211 at block 100 */
        status = dadig_in(100, &buffer);

        /* display some status information */
        printf("Alarm: %s\n", (buffer.alarm) ? yes: no);
        printf("Alarm suppressed: %s\n",
        (buffer.alarm_suppressed) ? yes: no);
        printf("Output value: %d.\n", buffer.output_value);

}
```

dasin

PURPOSE: This function reads information from a data acquisition analog block.

FORMAT:

```
status = dasin(blockno, bufptr);
struct dasbuf *bufptr;                 pointer to buffer structure

long int blockno;                      block number of input
long int status;                       return status:
                                       0 = success
                                       -1 = block does not exist
```

REMARKS: This function reads the internal structure of a data acquisition analog block (function code 177 or 178). After successful completion of the function, the buffer contains the same information sent to the loop in an exception report by the block. The dasbuf structure is defined in the ***das.h*** file as follows:

```
struct dasbuf
{
    unsigned short status;             /* point status */
    float val;                         /* point value */
    float higher;                      /* limit or alarm higher than
                                        current output */
    float lower;                       /* limit or alarm lower than
                                       current output */
    union                              /* extended status */
    {
        unsigned long exs_int;         /* extended point status */
        {
            struct
            unsigned nu  : 8;          /* not used */
            unsigned spare  : 8;       /* not used */
            unsigned xrr  : 1;         /* exception report received */
            unsigned rtag  : 1;        /* red-tagged */
            unsigned hwfail  : 1;      /* hardware failure  */
            unsigned orng  : 1;        /* out of range */
            unsigned lim  : 1;         /* limited */
            unsigned aman  : 1;        /* auto/manual (1=auto) */
            unsigned calc  : 1;        /* calculated */
            unsigned spare2  : 1;      /* spare */
            unsigned offs  : 1;        /* off scan */
            unsigned hdeva  : 1;       /* high deviation alarm */
            unsigned ldeva  : 1;       /* low deviation alarm  */
            unsigned hrata  : 1;       /* high rate alarm */
            unsigned lrata  : 1;       /* low rate alarm */
            unsigned varal  : 1;       /* variable alarms */
            unsigned alspr  : 1;       /* alarms suppressed */
            unsigned realr  : 1;       /* time in alarm re-alarm */
        } exs_bin;
    } exstatus;
}
```

The following constants, as defined in the ***das.h*** file, can be used to directly decode the status (dasbuf.status) and extended status (dasbuf.exstatus.exs_int) elements of the pre-

dasin

continued

vious dasbuf structure. This decoding is accomplished by performing a bitwise AND between the dasbuf element and the desired constant.

dasbuf.status:

Constant	Description
QBAD	Point quality
HALRM	High alarm
LALRM	Low alarm
LEVL_2	Level two alarm
LEVL_3	Level three alarm
EXS_CHG	Extended status change
R_TAG	Red-tagged
S_AUTO	Auto/manual (1=auto)

dasbuf.exstatus.exs_int:

Constant	Description	Constant	Description
REALALRM	Time in alarm re-alarm	CALCULA	Calculated
AL_SUPR	Alarms suppressed	AUTO	Auto/manual (auto=1)
VAR_ALR	Variable alarms	LIMITED	Limited
LO_RATE	Low rate alarm	O_RANGE	Out of range
HI_RATE	High rate alarm	HW_FAIL	Hardware failure
LO_DEVI	Low device alarm	RED_TAG	Red-tagged
HI_DEVI	High device alarm	XR_RECV	Exception report received
OFFSCAN	Off scan		

See also: ***bin***, ***blk_wrt***, ***bout***, ***dadig_in***

EXAMPLE:

```
#include <das.h>

void main(void)
{
    struct dasbuf buffer;
    long int status;
    static char *yes = "YES";
    static char *no = "NO";

    /*
    This example uses a pointer to a bitfield structure to directly access the elements
    of dasbuf.status instead of performing bitwise AND operations with the defined
    constants from the das.h file. This example structure is also defined in the
    das.h file as struct "s".
    */
    struct sts_bits                          /* defined in das.h as "s" */
```

dasin

continued

```
{
        unsigned unused : 8;
        unsigned q  : 1;                /* point quality */
        unsigned ha  : 1;               /* high alarm */
        unsigned la  : 1;               /* low alarm */
        unsigned alev  : 2;             /* alarm level: */
                                        /* 00 = normal */
                                        /* 01 = unused */
                                        /* 10 = level 2 */
                                        /* 11 = level 3 */
        unsigned exsc  : 1;             /* extended status changed  */
        unsigned rtag  : 1;             /* red-tagged */
        unsigned a_m  : 1;              /* auto/manual (1=auto) */
} *sts;

/* get the internals from function code 177 at block 100 */
status = dasin(100, &buffer);

/* point sts to the dasbuf status bits */
sts = (struct sts_bits *) &buffer.status;

/* display some dasbuf.status information */
printf("High alarm: %s\n", (sts-ha) ? yes: no);
printf("Low alarm: %s\n", (sts-la) ? yes: no);
printf("Auto/Manual: %s\n", (sts-a_m) ? "AUTO": "MANUAL");

/* display some dasbuf.exstatus.exs_int information */
printf("Re-alarm: %s\n",
        (buffer.exstatus.exs_int & REALARM) ? yes: no);
printf("Hardware failure: %s\n",

(buffer.exstatus.exs_int & HW_FAIL) ? yes: no);

}
```

filxfer

PURPOSE: This function transfers file data between two modules. This function transfers NVRAM and hard disk files.

FORMAT:

```
void filxfer(spec);
struct fxspec *spec;                file copy specifications structure
```

REMARKS: This function transfers file data between two modules. The caller specifies the file, direction of transfer, file offset, and number of bytes in the file to transfer. Any part of either file can be transferred to any part of the other file and both files must exist prior to the transfer. The specifications of the transfer operation are passed to this function in a structure of type fxspec. The status is also returned to the C program in this structure. The format of the fxspec structure is defined in the ***n90.h*** file as follows:

```
struct fxspec
{
    short int op;               /* operation code: 0=import, 16=export */
    char *sfnp;                 /* source file string pointer */
    char *dfnp;                 /* destination file string pointer */
                                /* format of string: */
                                /*   fnp = "@Lxx @Nxx @Mxx file" */
                                /*   @Lxx    remote loop */
                                /*   @Nxx    remote node */
                                /*   @Mxx    remote module */
                                /*   file    file name or number */
    long int sofs;              /* source offset (0=start of file) */
    long int dofs;              /* destination offset (0=start of file)*/
    long int nb;                /* bytes to transfer (-1=entire file)*/
    short int tim;              /* operation timeout value in seconds */
    short int wait;             /* wait mode: */
                                /* 0 = immediate return */
                                /* 1 = wait until complete */
    short int sta;              /* operation return status: */
                                /*  1 = busy */
                                /*  0 = normal completion */
                                /*  -1 = invalid specification */
                                /*  -3 = local memory overflow */
                                /*  -5 = timeout */
                                /*  -6 = invalid operation */
                                /*  -7 = interface unit error */
                                /*  -8 = can't get request to interface unit
*/
                                /*  -? = other error; see Table 5-1 */
    long int tc;                /* number of bytes transferred */
    long int spr;               /* reserved */
};
```

If the immediate return mode (fxspec.wait set to zero) is used, the fxspec structure ***must*** be declared globally (outside of all C functions) or locally with the static keyword. This declaration is required so that the contents of fxspec are preserved during the entire file transfer process.

filxfer

continued

By setting fxspec.sofs, fxspec.dofs, and fxspec.nb to the appropriate values it is possible to transfer an entire file or any continuous block within a file. To read an entire file, for example, set fxspec.sofs and fxspec.dofs to zero and fxspec.nb equal to negative one.

Note that an import operation (fxspec.op = zero) of specific source (fxspec.sfnp) and destination (fxspec.dfnp) specifications is equivalent to switching the source and destination specifications and performing an export operation (fxspec.op = 16).

Multiple transfer requests may be active at one time by using the immediate return option and/or by using multiple tasks. The number of requests that can be active at one time is limited only by the size of the dynamic memory pool of the local module. It is recommended that the number of requests active at one time be regulated so that the process control unit will not become saturated with excessive amounts of file transfer data.

Each active transfer request uses 200 bytes of local dynamic memory. The dynamic memory pool of the local module must be set accordingly or the operation will fail with the fxspec.sts element set to negative three. The size of the dynamic memory pool is set using the C Utility Program.

If the file transfer is occurring between two different process control units, this function sends its file transfer request to the local network interface unit. The interface unit is responsible for establishing a data route to the remote module. Once a module submits a transfer request, it waits until completion or returns to the C program, depending on the setting of fxspec.wait. The interface unit manages the actual transfer operation and signals the local module when it is complete. The destination file will not be updated until the transfer has successfully completed, thus if errors are encountered during the transfer process, the file will not become corrupt.

Since the interface unit may do the actual file transfer, the fxspec.sta offers an extended error status to indicate several other transfer problems to the C program. Table 5-1 defines the bits of the fxspec.sta structure element.

The remote module uses one module bus file (MBF) buffer for each active transfer operation. If the selected section of the file fits into an MBF buffer (i.e. fxspec.nb is less than the MBF buffer size), the remote file will lock the file only long enough to

filxfer

continued

read its contents into the MBF buffer. Otherwise, the remote module will lock the file during the entire transfer operation. It is recommended that the size of the MBF buffers is set large enough to hold the size of the largest amount of data that is to be transferred. The local module also uses an MBF buffer for the transfer operation and its size should be set equal to the size of the MBF buffer of the remote module. The MBF buffers are allocated using the C Utility Program.

See also: ***rfilef***, ***rfiler***

EXAMPLE:

```
#include <n90.h>
void main(void)
{
    static struct fxspec fx;
    static char *sfnp = "@L1 @N2 @M10 1000";
    static char *dfnp = "@L5 @N4 @M12 2000";

    /*
    transfer file 1000 at loop 1, node 2, module 10 to file 2000 at loop 5, node 4,
    module 12
    */
    fx.op = 0;                  /* import file data operation */
    fx.sfnp = sfnp;             /* set source spec */
    fx.dfnp = dfnp;             /* set destination spec */
    fx.sofs = 0;                /* start of source file */
    fx.dofs = 0;                /* start of destination file */
    fx.nb = -1;                 /* transfer entire file */
    fx.tim = 60;                /* timeout value */
    fx.wait = 1;                /* wait until complete */

    filxfr(&fx);

    /* display return status on attached terminal */
    printf("Return status of filxfr(): %d\n", fx.sta);
    printf("Bytes transferred: %d\n", fx.tc);

}
```

Table 5-1. Error Status Information

Status information format word ESSS OOOO CCCC CCCC

Type of Code	Availible Code	Explanation
E = Error Code	0x8001	Error not listed
	0x8002	Invalid file specifications
	0x8003	Invalid mode
	0x8004	Invalid operation
	0x8005	Invalid parameter
	0x8006	Write protected

filxfer

continued

Table 5-1. Error Status Information (continued)

Status information format word ESSS OOOO CCCC CCCC

Type of Code	Availible Code	Explanation
E = Error Code (continued)	0x8007	Read protected
	0x8008	Hardware fault
	0x8009	File not open/too many opens
	0x800A	File needs to be fixed
	0x800B	No reply
	0x800C	Reply length wrong
	0x800D	Busy reply limit exceeded
	0x800E	Memory/buffer allocation
	0x800F	Send error
	0x8010	Receive error
	0x8011	Disconnect error
	0x8012	Connect error
	0x8013	Listen error
S = Subsystem Code	0x1000	Module bus file I/O
	0x2000	Transport
	0x3000	Network file system
	0x4000	Host
O = Operation Code	0x0100	Read
	0x0200	Write
	0x0300	Open
	0x0400	Close
	0x0500	Seek/position
	0x0600	Lock/exclusive
	0x0700	Unlock/share
	0x0800	Copy file
	0x0900	Directory
	0x0A00	Delete
	0x0B00	Rename
	0x0C00	Move file
	0x0D00	Status/ completion
C = Status Code	0x0C00	OK
	0x0001	Busy
	0x0002	End-of-file

fltr3

PURPOSE: This function converts a floating point value to INFI 90 REAL_3 format value.

FORMAT:

```
status = fltr3(fptr, r3ptr);
float *fptr;          pointer to float value to convert
char *r3ptr;          pointer to buffer for results
long int status;      conversion results:
                      0 = success
                      -1 = underflow
                      1 = overflow
```

REMARKS: This function converts an IEEE floating point value to its INFI 90 REAL 3 format equivalent. C programs use this function to convert an internal floating point variable to the REAL 3 format that is usable by a network interface unit.

Because a four byte representation is being converted to a three byte representation, overflow and underflow are possible. If either occurs, the maximum (overflow) or minimum (underflow) REAL 3 value is assigned. The return status will indicate the error.

See also: ***r3flt***

EXAMPLE:

```
void main(void)
{
    float flt;
    char r3[3];
    long int status;

    /*
    convert a floating point value to a REAL 3 format value and display the status
    on an attached terminal.
    */

    flt = 3.1415926;
    status = fltr3(&flt, r3);

    printf("Conversion results: ");
    switch(status)
    {
        case -1:
            printf("underflow\n");
            break;
        case 0:
            printf("successful\n");
            break;
        case 1:
            printf("overflow\n");
            break;
    }
}
```

getargs

PURPOSE: This function reads information from the invoke C function block.

FORMAT:

```
pp = getargs(void);
struct parblock *pp;  return value is a pointer to the invoke C parameter block of
                      the current segment
```

REMARKS: This function allows a C program to read the Invoke C (function code 143) block number, its tune flag (set to one if the block has been tuned), and readable parameters of the block (specifications three through eight). The C program also reads the block output and a private checkpointed data area.

The parblock structure is defined in the ***n90.h*** file. The C_program does not need to declare memory for this structure. A pointer variable is all that is required. The parblock structure is not updated automatically. A call to the getargs() function must be made each time that the latest block values are required by the C program.

```
struct parblock
{
    short blockno;          /* block number of Invoke C */
    short tflag;            /* non zero if the block has been tuned */
    char S3, S4;            /* program-readable character specs */
    short S5, S6;           /* program-readable short specs */
    float S7, S8;           /* program-readable float specs */
    char S9;                /* spare character parameter - not used */
    short S10;              /* spare short parameter - not used */
    float S11;              /* spare float parameter - not used */
    struct                  /* private checkpointed data */
    {
        short qual;         /* block quality and alarm status */
        float bout;         /* block output value */
    } blockout;
    char cpsave[8];         /* free-format checkpointed data buffer */
};
```

The following constants are defined in the ***n90.h*** file and can be used to test or set each bit of the parblock.blockout.qual structure element:

Constant	Description
QBAD	Used to set or test quality
HALRM	Used to set or test high alarm
LALRM	Used to set or test low alarm

See also: ***putargs***

getargs

continued

EXAMPLE:

```
#include  <n90.h>

void main(void)
{
        struct parblock *pp;

        /*
        get the block number, tune status and quality of the Invoke C block (function
        code 143) and display the information on an attached terminal
        */

        pp = getargs();

        printf("The Invoke C block number is %d.\n", pp->blockno);
        printf("This block has %s been tuned.\n",
                (pp->tflag) ? " " : " not ");
        printf("The quality is %s/n",
                (pp->blockout.qual & QBAD) ? "bad" : "good");

}
```

inque

PURPOSE: This function returns the number of bytes in a serial port receive buffer.

FORMAT:

```
bytes = inque(port);
```

long int blockno;	port number: 0 = terminal port 1 = printer port
long int bytes;	bytes in receive buffer of port

REMARKS: This function examines the state of a serial port receive buffer and returns the number of bytes currently in the buffer. The inque() function should be called prior to calling the read() function to determine the number of bytes in the receive buffer. A call to the read() function will return only after the number of bytes received by the port equals the number of bytes specified to be read. In this manner, calling the inque() function eliminates C program suspension.

See also: ***outque***, ***portopen***

EXAMPLE:

```
void main(void)
{
    long int bytes;

    /* get number of bytes in the printer port receive buffer */
    bytes = inque(1);

    /* display the byte count on an attached terminal */
    printf("Printer port receive buffer:  %d bytes\n", bytes);

}
```

PURPOSE: This function returns the bus address of the module.

FORMAT:

```
address = mb_addr();
long int address;                    address of module
```

REMARKS: This function returns the setting of the module address dipswitches. This enables a C program to determine the address of its module.

EXAMPLE:

```
void main(void)
{
    long int address;

    /* get my address using mb_addr() */
    address = mb_addr();

    /* display it on an attached terminal */
    printf("My bus address is %d.\n", address);

}
```

outque

PURPOSE:

This function returns the number of bytes in a serial port transmit buffer.

FORMAT:

bytes = outque(port);	
long int port;	port number 0 = terminal port 1 = printer port
long int bytes;	bytes in transmit buffer of port

REMARKS:

This function examines the state of a serial port transmit buffer and returns the number of bytes currently in the buffer. The outque() function should be called prior to calling the write() function to verify that there is enough space in the transmit buffer to store the bytes to be written. A call to the write() function will return only after the number of bytes to be written is placed in the transmit buffer. In this manner, calling the outque() function eliminates the C program suspension.

See also: ***inque***, ***portopen***

EXAMPLE:

```
void main(void)
{
      long int bytes;

      /* get number of bytes in the printer port transmit buffer */
      bytes = outque(1);

      /* display the byte count on an attached terminal */
      printf("Printer port transmit buffer:  %d bytes\n", bytes);

}
```

portopen

PURPOSE: This function allows a serial port to be opened with a user specified buffer.

FORMAT:

phandle = portopen(port, mode, bufptr, bufsize);	
long int port;	port number to open: 0 = terminal port 1 = printer port
unsigned short mode;	port protocol: 0 = PMODE (default) at 9600 baud, eight data bits, no parity, and one stop bit. All other numbers represent a bitwise OR of the selected protocol values below.
char *bufptr;	pointer to buffer; NULL if the default buffer is to be used, and only a port mode change is required.
long int bufsize;	size in bytes of the buffer; this value must be evenly divisible by four.
long int phandle;	return value: -1 = portopen() failed ≥0 = handle of the port

REMARKS: This function allows a serial port to be opened with a buffer or protocol format other than the default settings. The buffer is divided into three pieces: 38 bytes of overhead and two circular queues (one for input and one for output). The default buffer is 550 bytes (256 bytes for each input and output) and cannot be reclaimed once the portopen() function is called. The default protocol (PMODE) is 9600 baud, eight data bits, no parity, and one stop bit. This function will fail if the specified port is already open. It is necessary to issue calls to the fclose() function for stdin, stdout, and stderr prior to a portopen() function call for port 0.

If a port is to remain open between scans of the segments containing function code 143, the variables used for the port handle and buffer ***must*** be declared globally (outside of all C functions) or locally with the static keyword. This declaration is required so that the port handle and buffer contents are preserved between scans. This declaration is the only way to guarantee that the port handle value and buffer contents are not corrupted by other module processes between executions of the C program.

Once a port is open, the inque() and outque() functions are used to obtain the number of bytes in the receive and transmit buffers respectively. The read() and write() functions are used to get data from and put data to the port. When the read() and

portopen

continued

write() functions are used following a portopen() function call, the following situations should be considered:

- The size of the portopen() buffer is greater than the number of bytes to read: A call to the read() function will return only after the number of bytes received by the port equals the number of bytes that were specified to read. While the read() function is waiting for characters to be received, C program execution is suspended. The call to the read() function will never return if the sending device does not transmit enough characters as specified to read.

- The size of the portopen() buffer is smaller than the number of bytes specified to read: A call to the read() function will never return since the number of bytes to read can never be received without port buffer overrun occurring.

- The number of free bytes in the portopen() buffer is larger than the number of bytes to write: A call to the write() function will return normally after the characters to write are placed in the portopen() buffer.

- The number of free bytes in the portopen() buffer is smaller than the number of bytes to write: A call to the write() function suspends C program execution until enough bytes have been transmitted so that the remaining bytes to write can be placed in the buffer. Indefinite module suspension can occur if the receiving device holds the port handshaking signals. This prevents the module from transmitting any characters.

The main point to consider when using the port I/O functions is to use the inque() and outque() functions to monitor the receive and transmit buffer space. The inque() function should be used prior to a read() call to accurately determine the number of bytes to read. The outque() function should be used prior to a write() function call to verify that enough free space exists in the transmit buffer to write the new data. Using these functions instead of relying on fixed byte reads and writes will always guarantee that module execution is not suspended.

The following constants (as defined in the ***ios1.h*** file) can be used to specify the mode argument for the portopen() function when an MFP is used. In this case, a "#define MFP" statement must appear in the C program prior to including the ***ios1.h*** file. The value for mode is constructed by performing a bitwise OR of the desired protocol values.

portopen

continued

Constants	Description
Baud rates:	
BAUD75	75 baud
BAUD110	110 baud
BAUD134	134.5 baud
BAUD150	150 baud
BAUD300	300 baud
BAUD600	600 baud
BAUD1200	1200 baud
BAUD1800	1800 baud
BAUD2000	2000 baud
BAUD2400	2400 baud
BAUD4800	4800 baud
BAUD9600	9600 baud
BAUD19200	19.2K baud
Data bits:	
DATA7	Seven data bits
DATA8	Eight data bits
Parity:	
EVENPAR	Even parity
ODDPAR	Odd parity
LOWPAR	Low parity
HIGHPAR	High parity
NOPAR	No parity
Stop bits:	
STOP1	One stop bit
STOP2	Two stop bits
Miscellaneous:	
BREAKON	Enables break detect
XONXOFF	Enables ^S/^Q flow control
RTSCTS	Enables RS-232 RTS and CTS handshaking
NOECHO	Disables character echoing and translation

The BREAKON, XONXOFF, and RTSCTS options are all implemented in module hardware and are ***disabled*** by default.

The NOECHO option (also known as O_RAW) is implemented in software and is disabled by default. Thus carriage returns are expanded to carriage return/line feed sequences. Also, characters received into the port are immediately echoed out of the port. Control characters (characters with ASCII values less than 32) are expanded and echoed as two character sequences. For example, control I (ASCII value nine) is echoed as "^I". When interfacing with binary devices the NOECHO option should be enabled to suppress this character expansion and echoing.

See also: ***inque***, ***outque***

portopen

continued

EXAMPLE:

```
#define MFP                                   /* required for ios1.h */
#include <ios1.h>

/* declare global variables and constants */
#define PORT 1                                /* open the printer port */
#define BUFSIZ 4096                           /* evenly divisible by four */
#define XMTBUF 2029                           /* transmit buffer size:(BUFSIZ-38)/2 */

char buff[BUFSIZ];                            /* buffer space for port */
int phandle;                                  /* file handle for port */

void main(void)
{
    char scratch[255];                        /* local scratch buffer */
    long int i;                               /* loop variable */
    unsigned short mode;                      /* port mode value */

    /* set the mode by ORing desired values from ios1.h */
    mode = BAUD2400 | ODDPAR | DATA7 | STOP2 | NOECHO;

    /* call portopen() with the new mode and buffer */
    phandle = portopen(PORT, mode, buff, BUFSIZE);

    /* check for portopen() failure */
    if (phandle < 0)
    {
        fprintf(stderr, "portopen() failed!\n");
        return;
    }

    /* setup data to send */
    for(i='A'; i<='Z'; scratch[i - 'A'] = (char) i++);

    /* NULL terminate the data */
    scratch[i] = '\0';

    /* make sure there is enough space in the transmit buffer */
    if ((XMTBUF - outque(PORT)) > strlen(scratch))
    {
        /* send data to port */
        write(phandle, scratch, strlen(scratch));
    }

    /*
    The following code waits until all characters have been transmitted or until five
    seconds have elapsed. This timed wait insures that the C program will exit even
    if the data transmission gets suspended at the serial port.
    */
    /* get current time in milliseconds */
    i = sysclk();

    /* loop to wait */
    while((outque(PORT)) && (sysclk() - i < 5000));

    /* close the port and return */
    close(phandle);
}
```

putargs

PURPOSE: This function writes information to the invoke C function block.

FORMAT:

```
putargs(pp);
struct parblock *pp;              argument is a pointer to the Invoke C
                                  parameter block of the segment
```

REMARKS: This function allows a C program to write to the invoke C (function code 143) block output and private checkpointed data area. The parblock pointer is obtained by first calling the getargs() function before a putargs() function call is made. Its structure is defined in the ***n90.h*** file.

See also: ***getargs***

EXAMPLE:

```
#include  <n90.h>

void main(void)
{
        struct parblock *pp;

        /*
        Write data to the private checkpointed space of the parblock structure. This data
        will then be sent to the backup in a redundant module configuration.
        */

        /* first get a pointer to the parblock */
        pp = getargs();

        /* update output value */
        pp->blockout.bout = 5.234;

        /* setup some private data to checkpoint */
        pp->cpsave[0] = 1;
        pp->cpsave[1] = 5;

        /* call putargs() to update this data */
        putargs(pp);

}
```

putmsg

PURPOSE: This function writes information to a serial port with gapless transmission.

FORMAT:

```
status = putmsg(phandle, bufptr, length);
long int phandle;        handle of port as returned by the portopen() function
char *bufptr;            pointer to message data
long int length;         length of message data in bytes
long int status;         return value:  -1 if bad port handle;  otherwise number of
                         bytes transmitted
```

REMARKS: This function behaves much like the write() function for output to the serial port. If gapless transmission of a message is required, then the putmsg() function must be used. Note that the target serial port must first be opened using open() or portopen() function.

There must be enough free space in the serial port transmit buffer for the entire length of the message or C program execution will be suspended until enough space is available. The outque() function can be used to test the available buffer space.

See also: ***inque***, ***outque***, ***portopen***

EXAMPLE:

```
#define MFP                                  /* required for ios1.h */
#include <ios1.h>
 /* declare global variables and constants */
#define PORT 1                               /* open the printer port */

int handle;                                  /* file handle for port */

void main(void)
{
    char * message = "This is a test message!\n";

    /* call portopen() using the default mode and buffer */
    phandle = portopen(PORT, PMODE, NULL, 0);

    /* check for portopen() failure */
    if (phandle < 0)
    {
        printf("portopen() failed!\n");
        return;
    }

    /* send a message to the port */
    putmsg(phandle, message, strlen(message));

    /* close the port and exit */
    close(phandle);
}
```

r3flt

PURPOSE: This function converts an INFI 90 REAL 3 format value to a floating point value.

FORMAT:

```
void r3flt(r3ptr, fptr);
char *r3ptr;              pointer to REAL 3 value
float *fptr;              pointer to float result
```

REMARKS: This function converts an INFI 90 REAL 3 value to its IEEE floating point format equivalent. A REAL 3 value received from a network interface unit can be converted to an internal floating point variable that can be used in arithmetic operations by the C program. There is no return value.

See also: ***fltr3***

EXAMPLE:

```
‘void main(void)
{
      float flt;
      char r3[3];
      long int status;

      /*
      Convert a REAL 3 value to a floating point value and display the value on an
      attached terminal.
      */

      flt = 3.1415926;
      status = fltr3(&flt, r3);
      if (status ! = 0) return;
      r3flt(r3, &flt);

      printf("Conversion value: %f\n"' flt);
}
```

rfilef

PURPOSE: This function copies a file from a module in the local process control unit to a file in a module in the same local process control unit.

FORMAT:

```
void rfilef(spec);
struct rffspec *spec;      file copy specification structure
```

REMARKS: This function copies a file residing in another module into a local file. Both modules must reside in the same process control unit. This function copies whole files only. The specifications of the copy operation are passed to this function in a structure of type rffspec. The status is also returned to the C program in this structure. The format of this structure is defined in the ***n90.h*** file as follows:

```
struct rffspec
{
      short int loop;            /* loop of remote module (set to 0) */
      short int pcu;             /* process control unit of remote module (set to 0) */
      short int mod;             /* remote module address */
      short int rfid;            /* remote file number */
      short int lfid;            /* local file number */
      short int lfdscr;          /* local file descriptor of file opened by */
                                 /* caller or -1 if not opened by caller */
      short int wait;            /* wait mode: */
                                 /*  0 = immediate return */
                                 /*  1 = wait until complete */
      short int maxtime;         /* operation timeout value in seconds */
      short int sts;             /* operation return status: */
                                 /*  1 = busy */
                                 /*  0 = normal operation */
                                 /*  -1 = invalid specification */
                                 /*  -2 = cannot access remote file */
                                 /*  -3 = local memory overflow */
                                 /*  -4 = cannot write local file */
                                 /*  -5 = timeout */
                                 /*  -6 = communications error */
};
```

If the immediate return mode (rffspec.wait set to zero) is used, the rffspec structure ***must*** be declared globally (outside of all C functions) or locally with the static keyword. This declaration is required so that the contents of rffspec are preserved during the entire file transfer process.

If the destination file is opened by the caller, it must be opened for read/write access. Also, the caller must not access the file while the rffspec.sts element is "busy" (set to one). When the transfer completes, the file position is indeterminate, thus the caller must seek to the desired position.

rfilef

continued

If the caller does not open the file, it will be opened and closed inside the function. The function obeys the same file access rules as the C program.

Multiple copy requests may be active at one time by using the immediate return option and/or by using multiple tasks. The number of requests that can be active at one time is limited only by the size of the dynamic memory pool of the local module. It is recommended that the number of requests that are active at one time be regulated so that the process control unit will not become saturated with excessive amounts of file transfer data.

The entire file is read into local dynamic memory before it is written to the local file. If the read operation is not successful, the local file is not modified. The dynamic memory pool of the local module must be set large enough to accommodate the size of the largest file that is to be transferred plus 200 bytes of memory for each active request. The operation will fail with the rffspec.sts element set to negative three if the dynamic memory pool is not large enough. For example, assume three transfer requests are to be active at one time and the largest file size is 10000 bytes. The size of the dynamic memory pool should be set to 10000 + 200 * 3 or 10600 bytes. The size of the dynamic memory pool is set using the C Utility Program.

Errors detected by this function are classified as fatal (i.e.: the specified remote file does not exist) or non-fatal (i.e.: the remote module is busy). When a fatal error is detected, this function will terminate immediately. When a non-fatal error is detected, the operation that caused the error is retried periodically until it succeeds, or the elapsed time (measured from the time of the initial call to this function) exceeds the rffspec.maxtime setting.

The remote module uses one module bus file (MBF) buffer for each active transfer operation. If the entire file fits into an MBF buffer, the remote file will lock the file only long enough to read its contents into the MBF buffer. Otherwise, the remote module will lock the file during the entire transfer operation. It is recommended that the size of the MBF buffers is set large enough to hold the size of the largest file to be transferred. The local module does not use an MBF buffer for the transfer operation. The MBF buffers are allocated using the C Utility Program.

rfilef

continued

The file transfer operation is performed by the module bus I/O task (see function code 90, specification two). A file transfer request will require at least three execution cycles of this task. One cycle is required to open the remote file, one or more cycles to read the remote file, and one cycle to close the remote file. The number of read cycles depends on the file size. Each read cycle on module bus will transfer 270 bytes. Each read cycle on the Controlway communication link will transfer 4,200 bytes. For example, if a 20,000 byte file is transferred on module bus, 77 cycles will be required. One cycle is required to open the remote file, 75 cycles to read its contents, and one cycle to close the remote file. The same transfer on Controlway communication link will require seven cycles. One cycle is required to open the remote file, five cycles to read its contents, and one cycle to close the remote file. If the command of any cycle fails due to excessive process control unit traffic or the remote module being busy, it is retried on the following cycle, thus the total transfer process is increased by one cycle.

The file transfer rate can be regulated by adjusting specification two of function code 90. It should be noted that several simultaneous file transfers can cause a significant increase in process control unit traffic. Use of file transfers should be carefully regulated so as not to saturate the process control unit and degrade overall process control unit message throughput.

See also: ***filxfer***, ***rfiler***

EXAMPLE:

```
#include  <n90.h>

void main(void)
{
      static struct rffspec rf;

      /* setup remote file transfer */
      rf.loop = 0;                        /* must be set to 0 */
      rf.pcu = 0;                         /* must be set to 0 */
      rf.mod = 12;                        /* address of remote module */
      rf.rfid = 1000;                     /* remote file number */
      rf.lfid = 1000;                     /* local file number */
      rf.lfdscr = -1;                     /* local file is not opened */
      rf.wait = 1;                        /* wait until complete */
      rf.maxtime = 60;                    /* timeout in 60 seconds */

      /* call remote file transfer function */
      rfilef(&rf);

      /* display return status on attached terminal */
      printf("Return status of rfilef(): %d\n", rf.sts);

}
```

rfiler

PURPOSE: This function copies a file from a module to the RAM memory buffer of the local module in the same process control unit.

FORMAT:

```
void rfiler(spec);
struct rfrspec *spec;        file copy specification structure
```

REMARKS: This function copies all or part of a file residing in another module into the RAM buffer of the local module. Both modules must reside in the same process control unit. The specifications of the copy operation are passed to this function in a structure of type rfrspec. The status is also returned to the C program in this structure. The format of this structure is defined in the ***n90.h*** file as follows:

```
struct rfrspec
{
    short int loop;          /* loop of remote module (set to zero) */
    short int pcu;           /* process control unit of remote module (set to 0) */
    short int mod;           /* remote module address */
    short int rfid;          /* remote file number */
    long int ofs;            /* remote file offset */
    long int cnt;            /* maximum number of bytes to read */
    char *ptr;               /* pointer to destination buffer */
    short int wait;          /* wait mode: */
                             /*  0 = immediate return */
                             /*  1 = wait until complete */
    short int maxtime;       /* operation timeout value in seconds */
    short int sts;           /* operation return status */
                             /*  1 = busy */
                             /*  0 = normal operation */
                             /*  -1 = invalid operation */
                             /*  -2 = cannot access remote file */
                             /*  -3 = local memory overflow */
                             /*  -5 = timeout */
                             /*  -6 = communications error */
    long int nb;             /* number of bytes read */
};
```

If the immediate return mode (rfrspec.wait set to 0) is used, the rfrspec structure ***must*** be declared globally (outside of all C functions) or locally with the static keyword. This declaration is required so that the contents of rfrspec are preserved during the entire file transfer process. The local buffer pointed to by rfrspec.ptr must also be declared in this manner.

By setting rfrspec.ofs and rfrspec.cnt to the appropriate values it is possible to read all of a file or any continuous block within a file. For example, to read an entire file rfrspec.ofs is set to zero and rfrspec.cnt is set equal to or greater than the file size. The local buffer, pointed to by rfrspec.ptr, must be large enough to accommodate rfrspec.cnt number of bytes.

rfiler

continued

Multiple copy requests may be active at one time by using the immediate return option and/or by using multiple tasks. The number of requests that can be active at one time is limited only by the size of the dynamic memory pool of the local module. It is recommended that the number of requests that are active at one time be regulated so that the process control unit will not become saturated with excessive amounts of file transfer data.

The specified remote file data is read into the local buffer, (pointed to by rfrspec.ptr) as the transfer occurs. If the transfer operation is not successful, the contents of the local buffer are indeterminate. Each active transfer request uses 200 bytes of local dynamic memory. The dynamic memory pool of the local module must be set accordingly or the operation will fail with the rfrspec.sts element set to negative three. The size of the dynamic memory pool is set using the C Utility Program.

Errors detected by this function are classified as fatal (i.e.: the specified remote file does not exist) or non-fatal (i.e.: the remote module is busy). When a fatal error is detected, this function will terminate immediately. When a non-fatal error is detected, the operation that caused the error is retried periodically until it succeeds, or the elapsed time (measured from the time of the initial call to this function) exceeds the rffspec.maxtime setting.

The remote module uses one module bus file (MBF) buffer for each active transfer operation. If the selected section of the file fits into an MBF buffer (i.e.: rfrspec.cnt is less than the MBF buffer size), the remote file will lock the file only long enough to read its contents into the MBF buffer. Otherwise, the remote module will lock the file during the entire transfer operation. It is recommended that the size of the MBF buffers is set large enough to hold the largest amount of data to be transferred. The local module does not use an MBF buffer for the transfer operation. The MBF buffers are allocated using the C Utility Program.

The file transfer operation is performed by the module bus I/O task (see function code 90, specification two). A file transfer request will require at least three execution cycles of this task. One cycle is required to open the remote file, one or more cycles to read the selected remote file data, and one cycle to close the remote file.

See also: ***filxfer***, ***rfilef***

rfiler

continued

EXAMPLE:

```
#include  <n90.h>

void main(void)
{
      static struct rfrspec rf;
      static char buffer[2000];

      /* setup remote file transfer */
      rf.loop = 0;                    /* must be set to zero */
      rf.pcu = 0;                     /* must be set to zero */
      rf.mod = 12;                    /* address of remote module */
      rf.rfid = 1000;                 /* remote file number */
      rf.ofs = 5000;                  /* offset at which to begin read */
      rf.cnt = 2000;                  /* number of bytes to read */
      rf.ptr = buffer;                /* local buffer to hold data */
      rf.wait = 1;                    /* wait until complete */
      rf.maxtime = 60;                /* timeout is 60 seconds */

      /* call remote file transfer function */
      rfiler(&rf);

      /* display return status on attached terminal */
      printf("Return status of rfiler(): %d\n", rf.sts);
      printf(Bytes read: %d\n", rf.nb);

}
```

sysclk

PURPOSE: This function reads the one millisecond resolution clock.

FORMAT:

```
value = sysclk();
long int value;                    /* clock value */
```

REMARKS: This function returns the one millisecond resolution clock value. The value returned is a 31 bit positive integer, thus clock rollover occurs approximately every 596 hours. The value is reset to zero when the module is reset and increases (in one millisecond intervals) to its maximum value. When roll-over occurs, the cycle repeats. All time interval applications that rely on sysclk() must take rollover into account.

See also: ***time_ms***

EXAMPLE:

```
void main(void)
{
    long int clock;

    /* get current millisecond resolution clock */
    clock = sysclk();

    /* display it on an attached terminal */
    printf("Current millisecond clock is %d.\n", clock);

}
```

time_ms

PURPOSE: This function allows access to the absolute time of the module.

FORMAT:

status = time_ms(buff);	
char *buff;	address of 12 byte buffer where bytes 0 through 5 are the absolute time in milliseconds, and bytes 6 through 11 are the signed wall clock offset.
long int status;	return status: 0 = success -1 = timestamp not valid

REMARKS: This function allows a C program to access the absolute time of the module. The six byte absolute millisecond count and the six byte signed wall clock offset are placed in the user supplied buffer.

Timestamping of point data in exception reports is accomplished by using the INFI-NET absolute time. This time value is a six byte integer count of milliseconds that have elapsed since March 1, 1980. If a module is configured to timestamp its exception reports, the absolute time is saved with the point value when it changes. The value and timestamp are then transmitted together to the network when the module is polled for exception reports. If the module is not configured to generate timestamps, each point is stamped externally when it is received by the polling process.

This function will return a negative one if the module is not properly configured for INFI-NET or if the module has not been timesynched. The timestamp is still available even if the module is not configured to generate timestamps.

When the module is properly configured for timestamp generation, a C program can timestamp exception report blocks by using the blk_wrt() function. Note that blk_wrt() accepts a four byte timestamp value, but the time_ms() provides a six byte value. The exception report contains a four byte timestamp that is later converted to a six byte value. A C program must supply the four low order bytes for the exception report (bytes two through five in the array buffer). The wall clock offset value can be used to generate time-of-day variables for display purposes, but it has no function in the timestamping process. The time-of-day is equal to the absolute time plus the wall clock offset.

A C program can verify that timestamping will be supported by the module by reading the block output from block 9,999. The output can be read using bin() to obtain a real value, that must then be coerced to an integer. Bit zero of this integer will be set

time_ms

continued

if the module is formatted for INFI-NET. Bit one will be set to indicate timestamp generation. Both of these bits must be set for C block timestamping.

See also: ***bin***, ***blk_wrt***, ***sysclk***

EXAMPLE:

```
#include <n90.h>
#include <blk_wrt.h>

void main(void)
{
    struct fbbuf bin_buff;              /* buffer for bin() */
    struct bs_fc30 buffer;              /* buffer for blk_wrt() */
    char timestamp[12];                 /* buffer for timestamp data */
    long int status, config;

    /*
    Verify that module is formatted for INFI-NET communications and supports
    timestamping. Then timestamp function code 30, an exception report block, at
    block 100.
    */

    /* verify module configuration using bin() */
    bin(9999, &bin_buff);
    config = (long int) bin_buff,fobval.r4;
    if ((!config & 0x01) || (config & 0x02))
    {
        printf("Module not properly configured!\n");
        return;
    }

    /* call time_ms() to get timestamp information */
    status = time_ms(timestamp);
    if (status)
    {
        printf("Timestamp not valid!\n");
        return;
    }

    /* setup and call blk_wrt */
    buffer.bs.fun_code = 30;
    buffer.bs.block_no = 100;
    buffer.bs.operations = BLK_WRT_TS;
    buffer.val = 3.14159;
    buffer.qa_on = 0;
    buffer.qa_off = 0;
    memcpy((char *) &buffer.timestamp, &timestamp[2], 4);
    status = blk_wrt(&buffer);
    if (status)
    {
        printf("Error calling blk_wrt()!\n");
    }

}
```

SECTION 6 - DEBUGGER PROGRAM

INTRODUCTION

The main purpose of the C debugger program is to assist in the troubleshooting of MFP module C language programs. The debugger allows the debugging of a C program from a work station as it executes on an MFP module. This is possible through the use of debugging tools such as single step execution, breakpoint execution, and watch variable options. Benefits derived from using the debugger include:

1. Reduced time and effort spent determining C program problems.

2. Increased flexibility determining such problems through the use of debugging tools such as breakpoints and watch variables.

3. The ability to inspect and evaluate the past course of C program flow.

PROGRAM OPERATION

To activate the debugger program, perform the following steps.

> **NOTE:** The work station must be connected to the INFI-NET or Plant Loop communication highway and the C program must be loaded into the target module. Specification one of the function code 143 located in the memory segment containing the C program must be set to two before proceeding.

1. Change directory to the directory containing the ***.CSP*** file for the program to be debugged.

2. Type:

CDEBUG [*port*] [*baud*] *file_name***.CSP** **Enter**

Where:

[*port*] is the work station port (COM1 or COM2) connected to the INFI 90 network interface unit, CPM module, or SPM module. This number must be a 1 (COM1) or a 2 (COM2) with 1 being the default value.

[*baud*] is the baud rate for the work station to module interface. Allowable baud rates are 150, 300, 1200, 2400, 4800, 9600, and 19200. The default baud rate is 9600.

file_name.**CSP** is the name of the C specification file corresponding to the program to be debugged.

NOTE: The [*port*] and [*baud*] fields are optional and can be entered in any order.

After a few seconds, the debugger main screen appears (see Figure 6-1). The text on the screen is the first (alphabetically) source for the program to be debugged. Along the top of the screen are seven options, each displaying a pull down menu when selected. Select these options by simultaneously pressing **Alt** and the first letter of each option.

```
File  Search  Run  Watch  Break  Module  Options          Module 3: no response
                                  Source
void main(void)
        {
        short i, j;

        printf("This is a test C program.\n");
        for(i=0; i<100; i++)
                {
                for(j=0; j<100; j++)
                        {
                        printf("%d/%d", i, j);
                        }
                }
        }

File: C.C  Line: 1  Col: 0
```

TPS0394A

Figure 6-1. Debugger Program Main Menu

The File Menu

Pressing **Alt** - **F** causes the pull down menu in Figure 6-2 to appear. Explanations of each menu option follow.

Open... Select this option and enter the desired source file name. The screen displays the specified file. Any source file that is part of the program can be opened.

View... Select this option and specify any DOS or ASCII file in the module directory. The screen displays the specified file.

DOS shell Selecting this option creates a DOS shell.

About... This option displays the program title page.

Quit The debugger program is exited with this option.

Figure 6-2. The File Pull Down Menu

The Search Menu

Pressing Alt - **S** causes the pull down menu in Figure 6-3 to appear. Explanations of each menu option follow.

Find... Select this option and specify a string of up to 20 alphanumeric characters. Press Enter and the debugger will search for the first occurrence of the text string. This line of code will be highlighted.

Next This option searches for the next occurrence of the text string. This line of code is highlighted.

Previous This option searches for and highlights the last occurrence of the text string.

Figure 6-3. The Search Pull Down Menu

The Run Menu

Pressing Alt - **R** causes the pull down menu in Figure 6-4 to appear. Explanations of each menu option follow.

Segment... This option specifies a segment that has function code 143 specification one set to two and starts execution of the C program in that segment. Activating the debugger for the first time stops all C programs.

Go All This option starts the execution of all C programs no matter what segment they are in.

Go to Cursor Selecting this option causes the program to execute up to and including the line highlighted.

Step Over This option executes a function without stepping through each line of the function.

Step Into This option is used to step through each line of a function before proceeding to the next line of the program.

Halt This option halts the execution of a program.

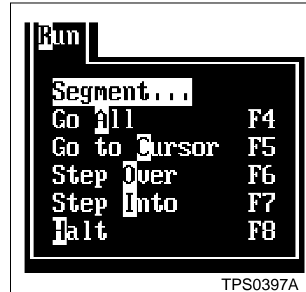


Figure 6-4. The Run Pull Down Menu

The Watch Menu

Pressing Alt - **W** causes the pull down menu in Figure 6-5 to appear. Watched variables appear in a window on the bottom of the screen. The information presented in the watch window is selectable. Refer to the *Options* menu for details. Up to 10 variables can be watched at a time. Explanations of each menu option follow.

Global add... This option displays a listing of all global variables and allows selection of one variable.

Local add... This option displays a listing of all local variables and allows selection of one variable.

Remove... This option removes the specified variable from the watch window.

Delete all This option removes all variables from the watch window.

Global variables are updated every time a breakpoint is encountered or at the global variable poll rate. Local variables of a function are updated when a breakpoint is encountered within the function.

Figure 6-5. The Watch Pull Down Menu

The Break Menu

Pressing Alt - **B** causes the pull down menu in Figure 6-6 to appear. Explanations of each menu option follow.

Status... This option displays a listing of information about each breakpoint. Breakpoint number, name of files and segments containing breakpoints, line number of each breakpoint, and the state of each breakpoint.

Toggle This option sets or clears a breakpoint at the present location of the cursor. At the *Enter segment number (0 - 7) or "*" for all:* prompt, enter the number of the segment where this breakpoint will be active.

Delete all This option removes all breakpoints.

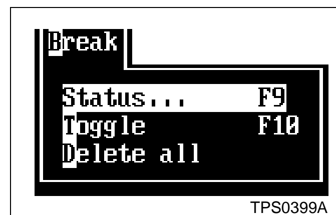


Figure 6-6. The Break Pull Down Menu

The Module Menu

Pressing Alt - **M** causes the pull down menu in Figure 6-7 to appear. Explanations of each menu option follow.

Reset module This option resets the module.

Configure module This option puts the module into configure mode.

Execute module This option puts the module into execute mode.

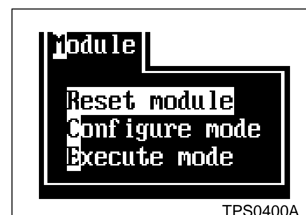


Figure 6-7. The Module Pull Down Menu

The Options Menu

Pressing **Alt** - **O** causes the pull down menu in Figure 6-8 to appear. Explanations of each menu option follow.

Breakpoint poll rate: Specifiy the rate (in seconds) at which the debugger queries the module for breakpoint status changes.

Global watch poll rate: Specify the rate (in seconds) at which the debugger queries the module for global variable value changes.

Jump to breakpoint: This option moves the cursor to the breakpoint location in the program. Specify on or off.

Single step segment: Specify the desired segment (0 - 7) for single step execution.

Full Watch information: This option controls the amount of information displayed in the watch window. Answer *on* and the watch widow displays the file name, scope, segment number, name, type, and value of each variable selected. If applicable, the function being executed is displayed also. Answer *off* and only the variable name and value are displayed.

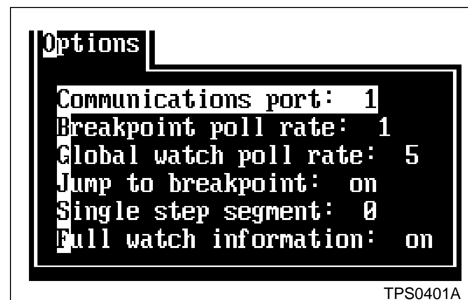


Figure 6-8. The Option Pull Down Menu

SECTION 7 - SAMPLE APPLICATION

INTRODUCTION

This section illustrates the C program development process by carrying a sample program through that process. For this example, a Bailey engineering work station connected (through communication port 1) to an INFI-NET communication highway through an INICI01 INFI-NET to Computer Interface. The target module is an IMMFP01 Multi-Function Processor module located in loop 1, process control unit 2, and module address 3. A computer terminal is connected to the MFP module through a NTMP01 Multi-Function Processor Termination Unit. See Figure 7-1 for the hardware setup.

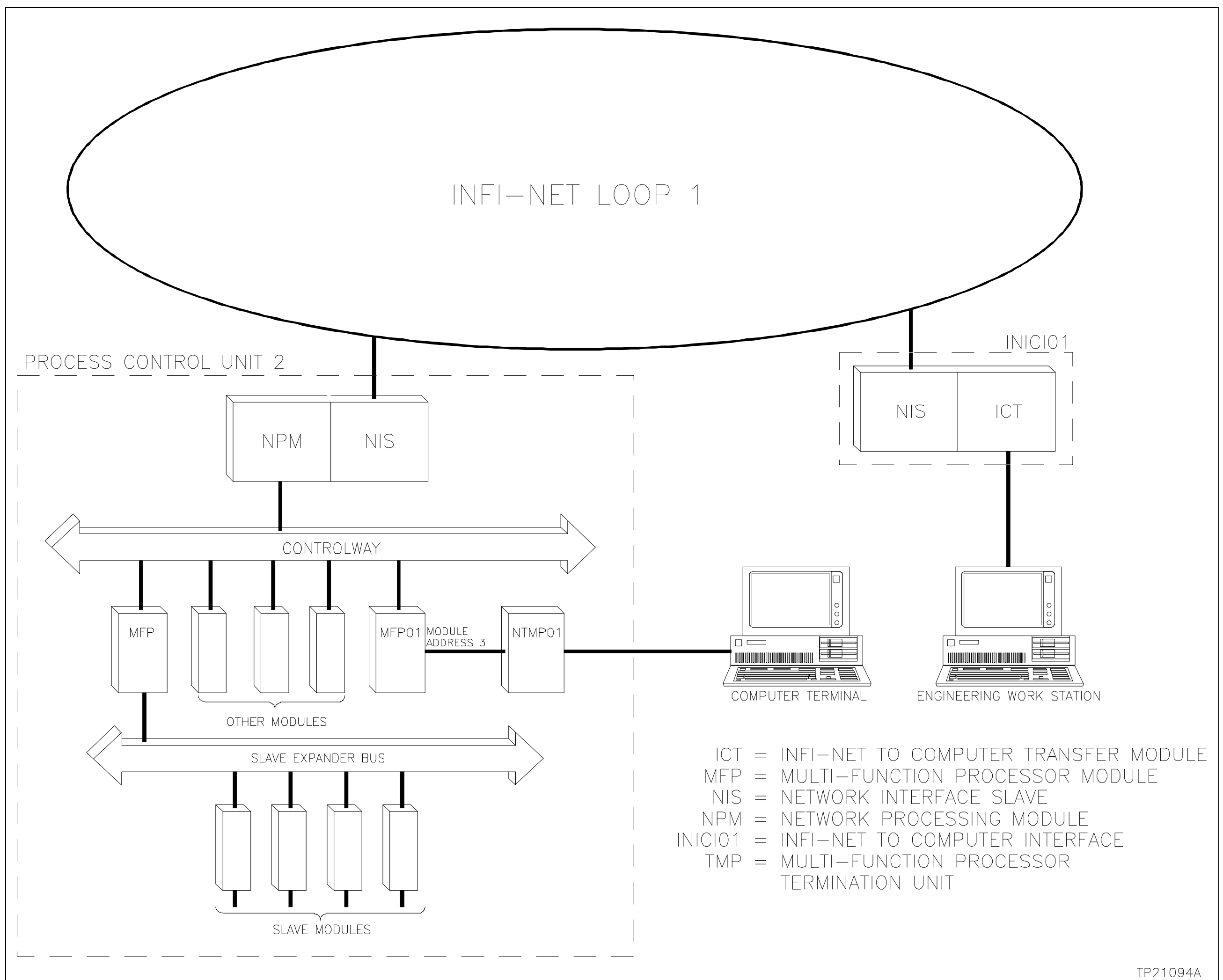


Figure 7-1. Sample Hardware Configuration

The sample program displays (on the computer terminal) the contents of a text file stored in the NVRAM memory section of the MFP module. At the computer terminal, the program prompts for the desired file number.

THE SAMPLE PROGRAM

The sample program consists of three source files. They are ***MAIN.C***, ***GETNUMB.C***, and ***DISPFILE.C***. The ***MAIN.C*** file contains the entry point of the program. The ***GETNUMB.C*** file gets a valid number from the user. The ***DISPFILE.C*** file displays the selected file on the computer terminal. The C:\CMFC directory contains these files.

MAIN.C Source File

The function main() is the entry point of the sample program. A C program can have up to eight entry points for multitasking capabilities. For simplicity, this sample program uses only one entry point. When the program executes (execute function code 143), the main() function is entered. The main() function calls the get_number() function to allow a file number to be entered. Also called up from the main() function is the display_file() function that displays the contents of the selected file on the computer terminal. The ***stdio.h*** file prototypes the computer terminal and file I/O functions. The ***MAIN.C*** source file consists of the following code.

```
#include <stdio.h>                  /* included for I/O specs */

int get_number();                   /* prototype from GETNUMB.C */
void display_file(int)              /* prototype from DISPFILE.C */

void
main()                              /* entry point of program */
{
    int filenumb;                   /* stores the file number */

    filenumb = get_number();        /* get entry from user */

    display_file(filenumb);         /* display file on the terminal */
}
```

GETNUMB.C Source File

The function get_number() loops until a valid file number is entered. After entering a file number, the get_number() function tries to open the file specified. If the open operation fails, an error message displays and the get_number() function waits for the next file number. If the open operation is successful, the specified file number is returned. The ***stdio.h*** file prototypes the computer terminal and file I/O functions. The ***GETNUMB.C*** source file consists of the following code.

```
#include <stdio.h>                          /* included for I/O specs */

int
get_number()
{
    FILE *fp;                               /* used to verify file */
    char buf[20];                           /* buffer for user entry */

    for(;;)                                 /* loop until valid entry */
    {
        printf("\nEnter file number to display:  ");
        gets(buf);
        fp = fopen(buf, "rt");/* check file existence */
        if (fp == NULL)
        {
                printf("Error: File not found.\n");
        }
        else
        {
                fclose(fp);
                break;
        }
    }

    return(atoi(buf));                      /* return valid File */
}
```

DISPFILE.C Source File

The function display_file() opens and displays the selected file on the computer terminal. This function provides a count of the program lines. Line lengths greater than the BUFLEN specification are truncated. The ***stdio.h*** file prototypes the computer terminal and file I/O functions. The ***DISPFILE.C*** source file consists of the following code.

```
#include  <stdio.h>                     /* included for I/O specs */

#define BUFLEN 128                      /* maximum file line length */

void
display_file(filenumb)
int filenumb;
{
    FILE *fp;                           /* pointer to specified file */
    int count = 0;                      /* rolling line count */
    char fil[20];                       /* file name for fopen() */
    char buf[BUFLEN];                   /* buffer for each file line */

    sprintf(fil, "%d", filenumb);       /* set file name for fopen() */

    printf("\nContents of file %s:\n", fil);
    printf("------------------------\n");

    fp = fopen(fil, "rt");/* open specified file */

    while(fgets(buf, BUFLEN, fp))/* loop while not EOF */
    {
```

```
                printf("Line %4d: %s/n",
                   ++count, buf);
        }

        printf("------------------------\n");
        fclose(fp); /* done, so close file */
}
```

WORK STATION BASED TESTING

Test the program on the work station before execution in the module. This process requires several steps. The program must be compiled and linked before it will execute in the work station.

Creating an Object File

Only compiled source files execute on the work station. Compile the ***MAIN.C***, ***GETNUMB.C***, and ***DISPFILE.C*** source files with a standard ANSI C compiler (not provided). Compiled source files have the same name as the source file but with a ***.OBJ*** file extension.

Creating an Executable File

Link the compiled files into one program file using a standard DOS linker program (not provided). Name this linked program file SAMPLE. This creates a ***SAMPLE.EXE*** file.

Running and Debugging the Program

Execute the program by typing the following:

SAMPLE.EXE [Enter]

Make any necessary changes to the source files (***.C*** files) and repeat the compilation and linking steps.

PREPARING FOR MODULE EXECUTION

After testing the program on the work station, it is ready for execution in the module. This requires several steps. The source files must be compiled for use in the module, linked into one program file, and downloaded into the module. The modules must also be initialized and formatted. Perform all of these steps on the work station.

Compiling the Program

The ***MAIN.C***, ***GETNUMB.C***, and ***DISPFILE.C*** source files must be compiled for use in the module. Perform this task with a C cross compiler. The compiler generates an object file (***.OBJ*** file

extension). To compile the program source files, perform the following steps:

1. Change directory to the directory containing the *MAIN.C*, *GETNUMB.C*, and *DISPFILE.C* source files.

2. Type:

   MCC *program_name* Enter

Where:

program_name is MAIN, GETNUMB, or DISPFILE.

Creating the C Specifications File

Only the C Utility Program (CUP) can create a C specifications file. To create the file, perform the following steps.

1. Start the C Utility Program by typing the following at the DOS prompt.

   CUP Enter

2. From the *CUP MAIN MENU*, select *A Read CSP File*.

3. At the *CSP Filename* prompt, type:

   SAMPLE F10

4. Press **Y** at the *File Does Not Exist. Create (Y/N)* prompt.

5. When the *CSP TITLE INFORMATION* screen appears, enter the information after the following prompt.

   Loop Address = **1**

   PCU Address = **2**

   Module Address = **3**

   Module Type = **MFP01**

6. Press F10 and then Esc.

7. Select *B Edit C Specifications* from the *CUP MAIN MENU.*

8. From the *EDIT C SPECIFICATIONS* menu, select *A Edit Object File List.*

9. Enter the following information into the object file name list.

No. 1 = **C:\CMFC\MAIN**

No. 2 = **C:\CMFC\GETNUMB**

No. 3 = **C:\CMFC\DISPFILE**

10. Press **F10**.

11. Press **N** at the *Transfer to CSP? (Y/N)* prompt.

12. Select *B Edit User Memory Specs* from the *EDIT C SPECIFICATIONS* menu.

13. Enter the following information into the *EDIT USER RAM SPECIFICATIONS* screen.

Max size of C idata section = **1024**

Max size of C udata section = **1024**

Min size of dynamic memory pool = **0**

Segment 0 Entry Point Name = **SAMPLE**

Segment 0 Stack Size = **2048**

14. Press **F10**.

15. Select *C Edit System Memory Specs* from the *EDIT C SPECIFICATIONS* menu.

16. Enter the following information into the *EDIT SYSTEM RAM SPECIFICATIONS* screen.

Max size of C code section = **1024**

Number of checkpoint buffers = **0**

Size of each checkpoint buffer = **0**

Number of MBF buffers = **0**

Size of each checkpoint buffer = **0**

17. Press **F10**.

18. Select *D Edit Format Specs* from the *EDIT C SPECIFICATIONS* menu.

19. Enter the following information into the *EDIT FORMAT SPECIFICATIONS* screen.

    Total RAM allocated for C = **70,000**

    Total NVRAM allocated for C = **10,000**

    Number of data files for C = **3**

20. Press **F10**.

Linking and Downloading the Program

The compiled source files must be linked into one program file. The module must be in configure mode and have been reformatted with the format specifications. Finally, the program file needs to be downloaded into the module. To link and download the program file, perform the following steps.

1. Select *C Build Linker Command File* from the *CUP MAIN MENU*.

2. Press **Y** at the *Continue? (Y/N)* prompt.

3. From the *CUP MAIN MENU*, select *D Invoke Linker*.

4. When the link is complete, the program memory map appears. Press **Esc**.

5. Select *E Module Management* from the *CUP MAIN MENU*.

6. From the *MODULE MANAGEMENT* menu, select *A CONFIGURE mode*.

7. Specify the loop 1, process control unit 2, and module address 3 on the screen.

8. Press **F10**.

9. From the *MODULE MANAGEMENT* menu, select *D FORMAT the module*.

10. Specify loop 1, process control unit 2, and module address 3 on the screen.

11. Press **F10**.

12. Select *F C Program Management* from the *CUP MAIN MENU*.

13. From the *C PROGRAM MANAGEMENT* menu, select *A Download C Program From EWS To Module*.

14. Specify loop 1, process control unit 2, and module address 3 on the screen.

15. Press F10.

EXECUTING THE PROGRAM

After the program downloads to the module, put the module into EXECUTE mode to run the program. To do this, perform the following steps on the work station.

1. From the *CUP MAIN MENU*, select *E Module Management*.

2. From the *MODULE MANAGEMENT* menu, select *E ADD Function Code 143*.

3. Specify loop 1, process control unit 2, module address 3, and function block number 100 on the screen.

4. Press F10.

5. Select *B EXECUTE mode* from the *MODULE MANAGEMENT* menu.

6. Specify loop 1, process control unit 2, and module address 3 on the screen.

7. Press F10 and Esc.

8. Press **Y** at the *Exit CUP? (Y/N)* prompt.

9. The program should be executing and ready to accept file numbers from the computer terminal.

APPENDIX A - FILE SYSTEM CHECK COMMAND

FSCK COMMAND DETAILS

The file system check function is included in the set of command functions to give C programmers and field support personnel the ability to detect the presence of file system corruptions and to correct them. The information output by this command is highly technical requiring the programmer to understand the makeup of the file system. However, this command does include an option to repair the file system automatically (to the best of its ability) based on the error information received.

NOTE: Interpretation of the output of this command may require assistance from Bailey Controls field support engineers or technical support personnel. It is hoped that such a diagnostic tool is rarely, if at all, needed. However, it is included as part of an ongoing effort to build self-diagnostics into all areas of the INFI 90 system.

FSCK *type name_1 name_2...name_x*

Where:

type is the operation option. These options are described later in this section.

name is the file, directory, inode, zone or device name.

The file system check function provides six options to the programmer. All options perform an entire file system check. In addition, each option provides supplemental, specialized information or actions. The options are as follows:

1. Perform automatic correction of file system errors.
2. List the attributes of the inodes specified in this command.
3. Verify the inodes specified in this command exist.
4. Verify the zones specified in this command exist.
5. List the files and directories in the file system.
6. List the superblock of the file system.

It is not necessary to be familiar with all of these options. The important options are options 1 and 5. When file system corruption is suspected, issue the FSCK command with option 5 selected. Enter the command as follows:

FSCK 5 \dev\hd0 Enter

This command tests all files and directories for corruption. A list of all the files and directories on the hard drive along with the test results is displayed. If the test results show an error, issue the FSCK command with option 1 selected. Enter the command as follows:

FSCK 1 \dev\hd0 Enter

This command attempts to automatically correct the errors detected with the original FSCK command. Any actions taken to correct the errors. are displayed. Reissue the FSCK command with option 5 selected as follows:

FSCK 5 \dev\hd0 Enter

If more errors are found, reissue the FSCK command with option 1 selected. Repeat this sequence of commands until no errors exist.

Because of the mechanical nature of hard drives, failures in hard drive file systems are more prevalent than in NVRAM file systems. Given the nature of such corruptions, it is highly probable that data will be lost during such failures. There is no guarantee that this command will be able to correct all detected errors. Thus, it is important to regularly back up hard drive files. The FSCK command is not intended as a replacement for regular backups. It is intended as a tool that may be able to save data lost due to unexpected power failures or error conditions.

APPENDIX B - MAKE FILE SYSTEM COMMAND

MKFS COMMAND DETAILS

The MKFS command formats the IMMFP03 module hard drive to the programmers' specifications. This command must be the first command sent to the file system. Any commands issued prior to the MKFS command will fail. The format for the MKFS command is as follows:

MKFS *num_inodes num_zones num_buf* [*sync_time*]

Where:

num_inodes is the number of inodes defined for the hard drive. The number of inodes specified tells the file system the maximum number of files and directories permitted to reside on the hard drive. Each file requires one inode and each directory requires one inode.

num_zones is the number of zones defined for the hard drive. Zone blocks are used by the file system to hold a file's data, directory entries, and file access blocks.

num_buf is the number of cache buffers defined for the hard drive. Cache buffers allow for RAM storage of files and help speed up the process of file accesses. The maximum number of buffers that can be allocated is 320. The number chosen is subtracted from the MFP modules RAM pool and thus impacts overall RAM memory availability. Each buffer consumes 530 bytes of RAM memory.

sync_time is the time between file system syncs in seconds. A sync is an action taken by the file system to purge all information in cache memory to disk. A file is only written to disk at the sync time. This parameter is defaulted to 30 seconds. Times other than 30 seconds can be specified using this optional parameter.

The hard drive is limited to the 33,554,432 bytes of information. This number is determined by the 65,536 logical blocks of 512 bytes each. From the total, the hard drive can be viewed as containing three partitions. The three parts are the file system

information blocks, inodes, and zones. The formula to determine the actual number of blocks used would be as follows:

$$\text{Total Blocks} = 2 + \frac{x}{4096} + \frac{y}{4096} + \frac{x}{32} + y$$

Where:

x The number of inodes.

y The number of zones.

Once this command is issued, no further MKFS commands are needed. Issuing this command a second time ***erases all data on the hard drive*** leaving only the default file system. If at some time it is decided that a larger memory partition is needed (i.e. more inodes and/or zones), all files on the MFP hard drive need to be copied to the work station and then copied back to the MFP hard drive once the new format is run.

It is recommended that the programmer leave room for future growth by not using all available memory space when formatting the hard drive. However, because of the checkpointing scheme used, wasteful allocation of memory space will only result in lengthy checkpointing times. Hard drive memory space should be allocated on a need basis leaving enough memory space for future growth.

Example: An MFP module hard dive is to be formatted to support 1,000 files and/or directories and ten megabytes of memory space for file data. Converting ten megabytes to zones (512 byte blocks of memory) reveals that 20,000 zones are required. Also, the file system is to have 200 cache buffers. The MKFS command would look like the following:

MKFS *10000 20000 200*

Where:

$$\text{Total blocks} = 2 + \frac{10,000}{4096}(\textit{rounded up}) + \frac{20,000}{4096}(\textit{rounded up})$$

$$+ \frac{10,000}{32}(\textit{rounded up}) + 20,000$$

$$= 2 + 3 + 5 + 313 + 20,000$$

= 20,323 blocks used (31 percent of hard drive used).

APPENDIX C - REDUNDANT HARD DRIVE COPY OPERATIONS

COPYING PROCEDURE

Copy the contents of a primary IMMFP03 Multi-Function Processor module hard drive to the secondary IMMFP03 Multi-Function Processor module hard drive using the following procedure. This requires a redundant pair configuration. Make sure that the redundancy link cable is connected correctly.

1. Select the G *Data File Management* option from the C utility program main menu.

2. Enter the loop, process control unit, and module address on the screen and press F10.

3. Verify the primary module is in configure mode by typing the following at the data file management shell:

   CFG Enter

4. Start the copying operation by typing the following:

   BKUP Enter

The copy operation takes up to 25 minutes (depending on the size of the hard drive partition allotted to the file system) to complete.

5. When the copy operation is complete, the primary module can be put into EXECUTE mode. To put the module into EXECUTE mode, type the following:

   EXE Enter

When this command is received, the primary module verifies the secondary module hard drive is ***reasonably*** current. If the hard drive contents are ***reasonably*** current, the secondary module mode changes to execute.

APPENDIX D - AVAILABLE C LANGUAGE FUNCTIONS

FUNCTION LIST

Table D-1 lists all the C language functions that are available to the IMMFC03, IMMFP01, IMMFP02, and IMMFP03 modules. Some of the functions listed are not available in all firmware revisions of the specified modules.

Table D-1. Available C Language Functions

Function	Availability			
	IMMFC03 Module	IMMFP01 Module	IMMFP02 Module	IMMFP03 Module
abs	X	X	X	X
acos	X	X	X	X
asin	X	X	X	X
atan	X	X	X	X
atan2	X	X	X	X
atof	X	X	X	X
atoi	X	X	X	X
atol	X	X	X	X
bin	X	X	X	X
blk_wrt	X	X	X	X
bout	X	X	X	X
bsearch[1]	X	X	X	X
calloc	X	X	X	X
ceil	X	X	X	X
checkpoint_file	—	—	—	X
close	X	X	X	X
clrerr	X	X	X	X
cos	X	X	X	X
cosh	X	X	X	X
cot	X	X	X	X
cpfil	X	X	X	X
creat	X	X	X	X
dadig_in	—	X	X	X
dasin	X	X	X	X
ecvt	X	X	X	X
eprintf[1]	X	X	X	X
except	X	X	X	X

Table D-1. Available C Language Functions (continued)

Function	Availability			
	IMMFC03 Module	IMMFP01 Module	IMMFP02 Module	IMMFP03 Module
exp	X	X	X	X
fabs	X	X	X	X
fclose	X	X	X	X
fdopen	X	X	X	X
feof	—	X	X	X
ferror	—	X	X	X
fflush	—	X	X	X
fgetc	X	X	X	X
fgetpos[1]	X	X	X	X
fgets	X	X	X	X
file-	X	X	X	X
filxfer	X	X	X	X
floor	X	X	X	X
fltr3	X	X	X	X
flushall	X	X	X	X
fmod	X	X	X	X
fopen	X	X	X	X
fprintf	X	X	X	X
fputc	X	X	X	X
fputs	X	X	X	X
fread	X	X	X	X
free	X	X	X	X
freopen	—	X	X	X
frexp	X	X	X	X
fscanf	X	X	X	X
fseek	X	X	X	X
fsetpos[1]	X	X	X	X
ftell	X	X	X	X
ftoa	—	X	X	X
fwrite	X	X	X	X
getargs	X	X	X	X
getc	—	X	X	X
getchar	—	X	X	X
getl[1]	X	X	X	X
gets	X	X	X	X
getw[1]	X	X	X	X
inque	X	X	X	X

Table D-1. Available C Language Functions (continued)

Function	Availability			
	IMMFC03 Module	IMMFP01 Module	IMMFP02 Module	IMMFP03 Module
isalnum	X	X	X	X
isalpha	X	X	X	X
isascii	—	X	X	X
iscntrl	X	X	X	X
isdigit	X	X	X	X
isgraph	—	X	X	X
islower	X	X	X	X
isprint	—	X	X	X
ispunct	—	X	X	X
isspace	X	X	X	X
isupper	X	X	X	X
isxdigit	X	X	X	X
itoa	—	X	X	X
itostr	—	X	X	X
labs[1]	X	X	X	X
ldexp	X	X	X	X
log	X	X	X	X
log10	X	X	X	X
longjmp	—	X	X	X
lseek	X	X	X	X
ltoa	—	X	X	X
ltostr	—	X	X	X
malloc	X	X	X	X
mb_addr	X	X	X	X
memccpy	—	X	X	X
memchr	—	X	X	X
memclr	—	X	X	X
memcmp	—	X	X	X
memcpy	—	X	X	X
memmove[1]	X	X	X	X
memset	—	X	X	X
modf	X	X	X	X
movmem	X	X	X	X
open	X	X	X	X
outque	X	X	X	X
perror[1]	X	X	X	X
portopen	X	X	X	X

Table D-1. Available C Language Functions (continued)

Function	Availability			
	IMMFC03 Module	IMMFP01 Module	IMMFP02 Module	IMMFP03 Module
pow	X	X	X	X
pow2	X	X	X	X
printf	X	X	X	X
putargs	X	X	X	X
putc	—	X	X	X
putchar	—	X	X	X
putl[1]	X	X	X	X
putmsg	X	X	X	X
puts	X	X	X	X
putw[1]	X	X	X	X
qsort[1]	X	X	X	X
r3flt	X	X	X	X
rand[1]	X	X	X	X
read	X	X	X	X
realloc	—	X	X	X
remove[1]	X	X	X	X
repmem	X	X	X	X
rewind[1]	X	X	X	X
rfilef	X	X	X	X
rfiler	X	X	X	X
scanf	X	X	X	X
setbuf	X	X	X	X
setjmp	—	X	X	X
setmem	X	X	X	X
sin	X	X	X	X
sinh	X	X	X	X
sprintf	X	X	X	X
sqrt	X	X	X	X
srand[1]	X	X	X	X
sscanf	X	X	X	X
stcd_i	X	X	X	X
stcd_l	X	X	X	X
stci_d	X	X	X	X
stcu_d	X	X	X	X
stpblk	X	X	X	X
stpchr	X	X	X	X
strcat	X	X	X	X

Table D-1. Available C Language Functions (continued)

Function	Availability			
	IMMFC03 Module	IMMFP01 Module	IMMFP02 Module	IMMFP03 Module
strchr	X	X	X	X
strcmp	X	X	X	X
strcoll[1]	X	X	X	X
strcpy	X	X	X	X
strcspn	—	X	X	X
strerror[1]	X	X	X	X
strlen	X	X	X	X
strncat	X	X	X	X
strncmp	X	X	X	X
strncpy	X	X	X	X
strpbrk	—	X	X	X
strrchr	X	X	X	X
strspn	—	X	X	X
strstr[1]	X	X	X	X
strtod[1]	X	X	X	X
strtok	—	X	X	X
strtol	—	X	X	X
strtoul[1]	X	X	X	X
swab	—	X	X	X
sysclk	X	X	X	X
tan	X	X	X	X
tanh	X	X	X	X
time_ms	X	X	X	X
toascii	—	X	X	X
tolower	X	X	X	X
toupper	X	X	X	X
trunc	X	X	X	X
ungetc	X	X	X	X
unlink	X	X	X	X
vfprintf[1]	X	X	X	X
vprintf[1]	X	X	X	X
vsprintf[1]	X	X	X	X
write	X	X	X	X
zalloc	—	X	X	X

NOTE:

1. These functions are linked in from the ***BAILEY.LIB*** file on the workstation. These functions consume user program memory space.

Index

A

Available C functions D-1

B

bin function 5-1
blk_wrt function 5-3
bout function 5-6

C

C program management 4-10
C specification files 4-1
checkpoint_file function 5-8
cpfil function 5-10
Created directroies and files 2-2
CUP
- C program management 4-10
- C specification files 4-1
- Data file management 4-11
- Error messages 4-17
- Format specification 4-6
- Linker command file 4-8
- Main menu 4-2
- Modem usage 4-13
- Module management 4-8
- Object file list 4-4
- System memory specifications 4-5
- User memory specifications 4-4

CUP error messages 4-17

D

dadig_in function 5-12
dasin function 5-14
Data file management 4-11
Debugger program
- Break menu 6-5
- File menu 6-2
- Main menu 6-2
- Module menu 6-5
- Operation 6-1
- Options menu 6-6
- Run menu 6-3
- Search menu 6-3
- Watch menu 6-4

E

Editing C specification files 4-3

F

filxfer function 5-17
fltr3 function 5-21
Format specifications 4-6
Function block requirements 1-2
Functions
- bin 5-1
- blk_wrt 5-3
- bout 5-6
- checkpoint_file 5-8
- cpfil 5-10
- dadig_in 5-12
- dasin 5-14
- filxfer 5-17
- fltr3 5-21
- getargs 5-22
- inque 5-24
- mb_addr 5-25
- outque 5-26
- portopen 5-27
- putargs 5-31
- putmsg 5-32
- r3flt 5-33
- rfilef 5-34
- rfiler 5-37
- sysclk 5-40
- time_ms 5-41

G

getargs function 5-22
Glossary 1-4

H

Hardware connections 2-2

I

inque function 5-24
Installation
- Cross compiler 2-1
- CUP 2-1

L

Linker command file 4-8

M

Manual content 1-1
mb_addr function 5-25

Index (continued)

Memory usage 1-4
Modem usage 4-13
Module initialization procedures 1-3
Module management 4-8

O

Object file list 4-4
outque function 5-26

P

portopen function 5-27
putargs function 5-31
putmsg function 5-32

R

r3flt function 5-33
Reference documents 1-7
Requirements 1-1
  Function blocks 1-2
  Hardware 1-1
  Module initialization 1-2
  Software 1-2
rfilef function 5-34
rfiler function 5-37

S

Sample program 7-2
Suggested program development 3-1
sysclk function 5-40
System memory specifications 4-5

T

time_ms function 5-41

U

User memory specifications 4-4
User qualifications 1-3

For prompt, personal attention to your instrumentation and control needs or a full listing of Bailey representatives in principal cities around the world, contact the Bailey location nearest you.

Australia
Elsag Bailey Pty. Limited
Regents Park, NSW
Phone: 61-2-645-3322
Telefax: 61-2-645-2212

Brazil
Bailey do Brasil
São Paulo
Phone: 55-11-548-4122
Telefax: 55-11-547-0315

Canada
Elsag Bailey (Canada), Inc.
Burlington, Ontario
Phone: 1-905-639-8840
Telefax: 1-905-639-8639

France
Elsag Bailey S.A.
Massy
Phone: 33-1-64-47-2000
Telefax: 33-1-64-47-2016

Germany
Bailey-F & P Automation GmbH
Overath
Phone: 49-220-473-90
Telefax: 49-220-473-979

Italy
Elsag Bailey
Genoa
Phone: 39-10-658-1
Telefax: 39-10-658-2941

Japan
Bailey Japan Company, Ltd.
Tagata-Gun, Shizuoka-Ken
Phone: 81-559-49-3311
Telefax: 81-559-49-1114

Jordan
Bailey Controls Jordan
Amman
Phone: 962-6-788-116
Telefax: 962-6-756-908

Mexico
Bailey Mexico S.A. de C.V.
Naucalpan
Phone: 52-5-557-6100
Telefax: 52-5-557-7022

Norway
Bailey Norge A.S.
Bergen
Phone: 47-55-222-000
Telefax: 47-55-222-010

People's Republic of China
Bailey Beijing Controls
Beijing
Phone: 86-10-401-0651
Telefax: 86-10-401-1643

Singapore
Elsag Bailey Pte. Ltd.
Singapore
Phone: 65-442-3200
Telefax: 65-442-2700

United Kingdom
Bailey Automation plc
Telford, Shropshire
Phone: 44-1952-670-477
Telefax: 44-1952-670-455

United States
Bailey Controls Company
Wickliffe, Ohio
Phone: 1-216-585-8500
Telefax: 1-216-585-8756

Venezuela
Bailey de Venezuela SA
Valencia
Phone: 58-41-329-196
Telefax: 58-41-327-632

Form I-E96-703A Litho in U.S.A. 1192
Copyright © 1992 by Elsag Bailey Process Automation, As An Unpublished Work
® Registered Trademark of Elsag Bailey Process Automation
™ Trademark of Elsag Bailey Process Automation

Elsag Bailey
Process Automation

29801 Euclid Avenue • Wickliffe, Ohio 44092 • (216) 585-8500
Telex: 980621 • Telefax: (216) 585-8756 or (216) 943-4609