C Language Implementation Guide For The Multi-Function Controller (NMFC03)
NOTICE

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

BAILEY CONTROLS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
WHO SHOULD READ THIS MANUAL

Users of Multi Function Controller Modules (NMFC03) who want to augment NETWORK 90 control strategies with the addition of C language programs will find this manual helpful and informative. A background in process control, C and BASIC programming languages is helpful.

This manual is intended to provide the user with the necessary information to utilize C programming with the NMFC03 module. It is not intended as a tutorial in C programming techniques.

MANUAL ORGANIZATION

This manual is divided into five sections: Introduction, Installation, Getting Started, The MFC, and Functions. An Appendix explaining NVM structure and a Glossary of unique terms is included in the rear of this manual.

Section 1, Introduction provides an overview of the C language requirements for the MFC.

Section 2, Installation provides step-by-step instructions for software (native and cross compilers) and hardware installation.

Section 3, Getting Started explains the procedures for compiling, linking, and executing C programs as well as the purpose of the C Utility Program (CUP).

Section 4, The MFC explains features such as the File System, Dynamic Memory Management, and function codes. A table of errors and corrective actions is also provided.

Section 5, Functions explains the unique MFC ROM resident functions.
PREFACE

REFERENCE DOCUMENTS

The user should refer to the listed documents as needed

E93 900 20 Function Code Application Manual
E93 905 1 Serial Port Module
E93 905 2 CIU Hardware Manual
E93 905 9 CIU Programmer’s Manual
E93 906 7 Multi Function Controller Module (NMFC03) Product Instruction

The C Programming Language
by Brian W. Kernighan and Dennis M. Ritchie
Prentice Hall Software Series

LATTICE C Compiler Documents (supplied with software)
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>General</td>
<td>1</td>
</tr>
<tr>
<td>Programming Components</td>
<td>1</td>
</tr>
<tr>
<td>BASIC versus C</td>
<td>2</td>
</tr>
<tr>
<td>Installation</td>
<td>3</td>
</tr>
<tr>
<td>Native C Compiler</td>
<td>3</td>
</tr>
<tr>
<td>Cross Compiler</td>
<td>4</td>
</tr>
<tr>
<td>Hardware Installation</td>
<td>5</td>
</tr>
<tr>
<td>Getting Started</td>
<td>7</td>
</tr>
<tr>
<td>General</td>
<td>7</td>
</tr>
<tr>
<td>Write the C Program</td>
<td>7</td>
</tr>
<tr>
<td>Compile the Program for the Workstation</td>
<td>7</td>
</tr>
<tr>
<td>Link the Program</td>
<td>7</td>
</tr>
<tr>
<td>Run the Program</td>
<td>8</td>
</tr>
<tr>
<td>Compile for the MFC</td>
<td>8</td>
</tr>
<tr>
<td>C Utility Program</td>
<td>8</td>
</tr>
<tr>
<td>Using CUP</td>
<td>9</td>
</tr>
<tr>
<td>Edit Format Specifications</td>
<td>13</td>
</tr>
<tr>
<td>Edit C Memory Specifications</td>
<td>15</td>
</tr>
<tr>
<td>Edit Object Name List</td>
<td>18</td>
</tr>
<tr>
<td>Build Linker Command File</td>
<td>19</td>
</tr>
<tr>
<td>Link for the MFC</td>
<td>20</td>
</tr>
<tr>
<td>Configure the Module</td>
<td>20</td>
</tr>
<tr>
<td>C Program Management Download</td>
<td>22</td>
</tr>
<tr>
<td>Data File Management</td>
<td>23</td>
</tr>
<tr>
<td>The MFC</td>
<td>25</td>
</tr>
<tr>
<td>General</td>
<td>25</td>
</tr>
<tr>
<td>Module Format</td>
<td>25</td>
</tr>
<tr>
<td>MFC File System</td>
<td>25</td>
</tr>
<tr>
<td>File Names/Device Names</td>
<td>26</td>
</tr>
<tr>
<td>Dynamic Memory Management</td>
<td>26</td>
</tr>
<tr>
<td>C Function Codes</td>
<td>26</td>
</tr>
<tr>
<td>Function Code 143 Invoke C</td>
<td>26</td>
</tr>
<tr>
<td>Function Code 144 Allocate C</td>
<td>28</td>
</tr>
<tr>
<td>CPU LEDS</td>
<td>28</td>
</tr>
<tr>
<td>Functions</td>
<td>31</td>
</tr>
<tr>
<td>bin</td>
<td>31</td>
</tr>
<tr>
<td>bout</td>
<td>31</td>
</tr>
<tr>
<td>cpfile</td>
<td>32</td>
</tr>
<tr>
<td>fltr 3</td>
<td>33</td>
</tr>
<tr>
<td>getargs</td>
<td>33</td>
</tr>
<tr>
<td>inque/outline</td>
<td>34</td>
</tr>
<tr>
<td>portopen</td>
<td>34</td>
</tr>
<tr>
<td>putargs</td>
<td>35</td>
</tr>
<tr>
<td>puumsg</td>
<td>36</td>
</tr>
<tr>
<td>rfilef</td>
<td>36</td>
</tr>
<tr>
<td>rfiler</td>
<td>38</td>
</tr>
<tr>
<td>r3flt</td>
<td>40</td>
</tr>
<tr>
<td>sysclk</td>
<td>40</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS - continued

Appendix A MFC Memory Structures..........................43
  C RAM ..................................................43
  Function Block RAM ..................................43
  Idata Section ..........................................43
  Udata Section ..........................................43
  Segment Stacks and Library Data ......................44
  Dynamic Memory Pool .................................44
  Checkpoint Buffers ...................................44
  MBF Buffers ...........................................44
  Code Section ...........................................45
  NVM Structure ........................................46
  Module Format Table ..................................46
  File Directory .........................................46
  File Space .............................................46
  Function Block Space .................................46

Glossary....................................................49

LIST OF FIGURES

Figure 3 1 CLP Opening Menu ............................10
Figure 3 2 CSP Enter Filename ............................11
Figure 3 3 CSP Title Page ................................12
Figure 3 4 Initial Edit Menu ............................13
Figure 3 5 Main Edit Menu ...............................14
Figure 3 6 Memory Format Specifications ...............14
Figure 3 7 Edit C Memory Specifications
  Menu C User RAM page 1 ............................16
Figure 3 8 Edit C Memory Specifications
  Menu C System RAM page 2 ..........................17
Figure 3 9 Edit C Memory Specifications
  Menu Memory Totals page 3 ..........................18
Figure 3 10 Edit Object Name List Menu .................19
Figure 3 11 Build Linker File Menu .....................20
Figure 3 12 Configure Module Displays ................21
Figure 3 13 C Program Download Menu ..................22
Figure 3 14 Display Module's Directory .................23

LIST OF TABLES

Table 4 1 MFC BASIC vs MFC C ..........................2
Table 4 2 Function Code 143 Invoke C Specifications ..27
Table 4 3 Function Code 144 Allocate C Specifications ..26
Table 4 4 LED ERROR Codes ..............................29
SECTION 1

INTRODUCTION

GENERAL

The Multi Function Controller Module is the most versatile of the NETWORK 90 Controller family. A key module feature is the ability to execute both BASIC and C language programs. This document addresses C language applications. The addition of C to a control strategy provides the user with increased versatility, greater execution speed than with BASIC, and multi tasking capability (only one task at a time can run BASIC).

PROGRAMMING COMPONENTS

To use C, the following hardware is required.

1. Bailey Workstation
   RS 232 serial port
   640 Kbytes of RAM
   Hard Disk Drive

2. Multi Function Controller Module (NMFC03)
   68020 microprocessor
   256 Kbytes EPROM
   512 Kbytes RAM
   80 Kbytes Non volatile RAM (NVM)

3. Computer Interface Unit (NCIU02 or NCIU03) or Serial Port Module (NSPM01)

The following software for implementing C is supplied by Bailey:

1. Lattice® Software
   - iAPX86 native C Compiler (4 floppy disks)
   - MS DOS>68000 C Cross Compiler (2 floppy disks)
   - LMB68K Linker (included on Cross Compiler disks)

2. Workstation Resident Software
   C Utility Program (CUP) (1 floppy disk) with the following files.
   CUP.EXE
   MFCLIB
   PCLIB
   Support Files

@Lattice registered trademark of Lattice Incorporated
INTRODUCTION

BASIC Versus C
The NMFC03 is able to simultaneously execute BASIC and C programs. Table 1.1 provides the similarities and differences between the two languages.

<table>
<thead>
<tr>
<th>BASIC</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Develop program on MFC</td>
<td>Develop program on EWS</td>
</tr>
<tr>
<td>Use dumb terminal connected to MFC terminal port</td>
<td>Download to MFC via CIU/SPM</td>
</tr>
<tr>
<td>BASIC editor</td>
<td>User's choice of source editor</td>
</tr>
<tr>
<td>Program executed by interpreter</td>
<td>Program executed as machine code (approx. 5 times faster than BASIC)</td>
</tr>
<tr>
<td>Memory requirements specified in Function Code</td>
<td>Memory requirements specified in CUP</td>
</tr>
<tr>
<td>Multiple invocations from 1 function block segment</td>
<td>Multiple invocations from all function block segments</td>
</tr>
<tr>
<td>Access to function block output values via BIN/BOUT</td>
<td>Same</td>
</tr>
<tr>
<td>Access to both serial ports</td>
<td>Same</td>
</tr>
<tr>
<td></td>
<td>Dynamic memory allocation</td>
</tr>
<tr>
<td></td>
<td>Dynamic data files</td>
</tr>
</tbody>
</table>
INSTALLING THE C ENVIRONMENT

Native C Compiler
The Native C compiler (iapx86) generates Workstation executable code. This compiler is installed first. The compiler files are on four floppy disks. To install, follow the steps below:

1. Insert disk 1 of 4 in drive A of the Workstation.
   
   **NOTE:** Disk 1 has a README file explaining the installation procedure and other notes of interest to the user. Refer to this file for clarification.

2. Type INSTALL.

3. Answer the system prompts. You will be asked which version(s) of the compiler you would like installed. For maximum flexibility you should select A (for all memory models). If disk space is limited on your Workstation, the S option (small program and data) should be appropriate for most applications. Reinstallation of additional models can be done later. Refer to the Lattice Programmer's Reference Volume I for a discussion of the various memory models.

As the files are copied from floppy to hard disk, file names scroll on the display (this is true for all installation procedures).

The system generates an audible signal when the files are copied and prompts you to insert disk 2.

4. Remove disk 1; insert disk 2.

5. Press any key.

After disk 2 files have been copied

6. Remove disk 2; insert disk 3.

7. Press any key.

After disk 3 files have been copied

8. Remove disk 3; insert disk 4.

INSTALLATION

After disk 4 files have been copied, the **Native C** compiler installation is complete. Proceed to the next installation steps.

**Cross Compiler**
The second step in the installation procedure is to install the Cross Compiler. The Cross Compiler generates MFC executable code.

To install the Cross Compiler:

1. Insert the SCCU IPC C Utility Disk into drive A
2. Type INSTALL
3. Press ENTER.

You will be prompted to insert the Lattice Cross Compiler diskettes into the drive.

As in the previous installation procedure, file names scroll on the display as they are copied onto the hard disk. The system prompts you when the installation is complete.

The INSTALL.BAT program installs the utilities and library functions necessary for MFC interpretation. This program creates two directories: C:\LCC and C:\CMFC. Cross compiler executable files are copied into LCC; Bailey utilities into CMFC.

**Setting Environment Variables**
The AUTOEXEC.BAT file in the root directory needs to be edited so that the include environment variable can be added:

```
SET INCLUDE C \LC
```

Include instructs LATTICE compilers to look in the C \LC directory for files specified for inclusion by the '#include' directive. Note that include files (e.g., stdio.h, math.h) are different for MFC compilation than for Workstation compilation. The include files for the Workstation reside in \LC, while the include files for the MFC reside in \CMFC (this occurs during the installation procedure).

Additionally, set the path environment variable to look for commands in the \LC, \LCC, and \CMFC directories.

```
SET PATH C:\LC;C \LCC,C \CMFC
```
HARDWARE INSTALLATION

Either a Computer Interface Unit (CIU) or Serial Port Module (SPM) can be used as the interface between the Workstation and the NPC. Refer to related product instructions for CIU or SPM information.

When a CIU is used, connect the HCBLO4 serial cable from the CIU to the terminal port 0 (terminal port 0) of the Workstation COM1 or COM2 port to the RS-232 female connector on the faceplate of the Serial Port Module.

When an SPM is used, connect the HCBLO4 serial cable from the Workstation COM1 or COM2 port to the RS-232 female connector.
SECTION 3
GETTING STARTED

GENERAL

This section provides the progression of user actions required to create a C program for ultimate execution by the MFC. The first part of this section explains the compilation and linking procedures on the Workstation. The second part explains the procedures for the MFC.

WRITE THE C PROGRAM

Using standard C Language types, operators, expressions; write the program.

Any text editor that operates on the Bailey Workstation can be used to write the program. The program source file should have .C as a file extension and be stored in one or more text files.

COMPILE THE PROGRAM FOR THE WORKSTATION

After the program has been written and stored, a test execution is run for debugging purposes. To do so, the program must be compiled with the Native (Lattice) compiler. This compiler requires two passes to complete the process. The first pass (LC1.EXE) checks the syntax, etc. and generates an intermediate file. The second pass (LC2.EXE) generates the object file. The compilation procedure follows:

Type LCx Program name. Press ENTER

x is S, P, D, or L memory models. Each memory model contains space for program and data addresses, library, and object files. When compilation starts, the compiler searches the directory in x above and pulls the appropriate library and object files.

When the compilation is complete, the system prompts:

TOTAL FILES: , SUCCESSFUL COMPILATIONS. 

LINK THE PROGRAM

After the source files have been compiled, they must be linked to produce an executable program file.

Type LINKx Program name Press ENTER.

NOTE: x is the same memory model used in compilation.
GETTING STARTED

An executable program file with the .EXE extension is created after a successful link. When the program is invoked, three buffered files are opened and assigned to the workstation: stdin, stdout, and stderr. The buffers stdin and stdout can be redirected from the command line.

Resolution at link time of MFC specific functions such as 'bin' and 'bout' can be achieved by linking the PCLIB library to your program. PCLIB is included on the Workstation software disk.

RUN THE PROGRAM

After successful compilation and linking, run the program for testing and debugging on the Workstation.

COMPILE FOR THE MFC

The Workstation compiled version of the program must now be compiled for the MFC module. This task is accomplished with the Lattice C Cross Compiler. This compiler, like the Native compiler, requires two passes to complete the process. The first pass (LC1M EXE) checks the syntax, etc. and generates an intermediate file. The second pass generates the Object file. The compilation procedure follows:

1. Change to directory in which the C source file resides.
2. Type CMFCC followed by program name

NOTE: CMFCC is a batch file which invokes the LCM compiler, using the appropriate options.

3. Press ENTER

When compilation is done, the display shows the hex values for idata, udata, and text. Use these values when working with the Edit C Memory Specifications menu in CUP, which is invoked next.

C UTILITY PROGRAM

The C Utility Program (CUP) is a Bailey designed program that enables the user to specify the memory and linking requirements for the MFC module to execute C programs. This program is loaded into the \CMFC directory during the installation procedure. CUP is invoked only after the C program has been compiled by the cross compiler.
CUP provides the following capabilities to the user:

- specify MFC module format
- specify C program memory requirements
- specify C program linking requirements (creates linker specification file (*.ink)
- communicate with the MFC
  
  format module
  change module mode
  monitor module status
  create, initialize, read data files on module
  download C programs to module

Using CUP
Before cross compiling the C program for the MFC, it is strongly recommended that the program be executed and tested on the Workstation. After successful completion of execution and testing, the program is ready for the first CUP session which accomplishes the following:

a. Specifies module format requirements
b. Specifies program memory requirements
c. Creates linker specification files

To access CUP:
1. Type CUP.
2. Press ENTER.

Refer to Figure 3.1 for the CUP opening menu.
FIGURE 3.1  CUP Opening Menu

The C specification file (CSP) must be created first. The CSP file resides in the Workstation and contains information that describes the C environment for the MFC module. Included in this information is

- a  format specifications
- b  memory specifications
- c  list of object files that comprise the C program

Note that one CSP file must exist for each target MFC.

From the CUP opening menu

1  Press the F3 key to create a new CSP file. The display of Figure 3.2 appears on the CRT.

2  Enter the name of the CSP file
FIGURE 3-2 --- CSP   Enter Filename

3 Press the F1 key  The display of Figure 3 3 appears on the CRT

NOTE: PCU and module addresses specified here are subsequently used by CUP for communication with the MFC module

4 Key in applicable entries into the fields of this display

5 Press Escape.
GETTING STARTED

FIGURE 3.3 CSP Title Page

6Press Escape again. The entries you made in step 4 will be written to the disk. The display of Figure 3.4 is now on the terminal. The first task is to edit memory and link specs, select F1. The display shown in Figure 3.5 comes up.

From this display, the user has several options. Each option is explained in subsequent paragraphs.
FIGURE 3.4  Initial Edit Menu

Edit Format Specifications
Format specifications are used to define the MFC's memory 'format.' Through this editing function, the user defines the amount of RAM, NVM, and number of data files the C program uses. To enter this function, press F1
GETTING STARTED

FIGURE 3.5  Main Edit Menu

FIGURE 3.6  Memory Format Specifications
GETTING STARTED

Finding appropriate values for the format specs depends very much on the programming application at hand. If the user requires guidance in selecting these values, it is recommended that the 'Edit Memory Specs (F3)' chore be done first. Then return to this menu afterwards.

When done with the menu, press F1 to save entries.

Edit C Memory Specifications
Select F3 from the Main Edit Menu to access Edit C Memory Specifications. This menu is used to define the RAM requirements for the C program. There are three pages to this menu. On page 1, the user defines User RAM area, including idata, udata, and dynamic memory pool sizes idata and udata values are displayed on the console after compilation. These values are entered into their respective fields on page 1. (NOTE: The cross compiler outputs idata and udata in hexadecimal. Convert to decimal when using CUP.)

Page 1 also requires assignment of dynamic memory pool and runtime stacks. (C program invocations receive an entry point and runtime stack based on the function block segment where the invoke C function code is configured. The user provides the name of the C function which serves as the entry point for each segment task which invokes C.)

Dynamic memory pool and runtime stacks are part of the C User RAM area, which has 64K boundaries. The user can configure excess memory into the dynamic memory pool or runtime stacks. Excess memory is assigned to the dynamic memory pool, by default. If the C program does not use dynamic memory functions, this default can be overridden by the user by specifying values in the stack fields to consume any excess memory (up to the next 64K boundary).

The user specifies the maximum RAM size as a result of compilation. The values should be based on those output by the cross compiler to the console. (NOTE: The cross compiler outputs in hexadecimal; CUP in decimal.) The user can specify larger than actual values to allow for program growth.

On the first page of this display (Figure 3.7), the user identifies the idata, udata, and dynamic memory pool sizes.
GETTING STARTED

<table>
<thead>
<tr>
<th>Maximum number of tasks</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Code Section</td>
<td>20</td>
</tr>
<tr>
<td>Minimum Dynamic Memory</td>
<td>1000</td>
</tr>
</tbody>
</table>

**FIGURE 3.7 Edit C Memory**

*Specifications Menu C User RAM page 1*

Press the PgDn key to access page two. Page two (Figure 3.8) is for entry of the System RAM area including the Code Section, Checkpoint Buffer and Module Bus File (MBF) I/O specifications. These optional buffers are configured into System RAM. Checkpoint buffers are used when file data is sent to the backup MFC in a redundant configuration.

MBF buffers are used for copying file data to respond to module bus requests for file operations. This occurs when CPU performs file data reads at user's request, and for C callable Module Bus File transfers (see rfile and rfill in Functions section).
Press the PgDn key to access page three. Page three (Figure 3-9) shows the totals resulting from the input of the previous pages. User RAM is rounded up to a 64K boundary. The sum of User RAM and System RAM should not exceed the Total RAM specification entered in the Edit Format Specs menu. It is important to note that the NVM requirement shown does not account for data file space. If you expect to use data files, estimate the limits of data file growth and add that amount to the given value.

When done with page three, press the Escape key to save your entries.
GETTING STARTED

FIGURE 3-9  Edit C Memory
Specifications Menu  Memory Totals  page 3

Edit Object Name List
From the Main Edit Menu, select F4 to access this menu
Edit Object Name List (Figure 3-10) is used to tell the utility
which object files (those created during the second pass of
the LCM compiler) to include in the linking process. Enter
the name of all the O files you want included in the link. The
linker program is LMB68K EXE. This linker searches the
MFCLIB for MFC ROM resident functions, and LCL and LCML
for Lattice runtime support functions not found in MFC ROM
Additionally, the user can request CUP to include user writer
linker specification files. To do so, put a tilde (~) in front
of the filename. Refer to section 4.2.2 of Lattice Cross
Compiler document for instructions on how to create a linker
specification file.
FIGURE 3.10  *Edit Object Name List Menu*

**Build Linker Command File**

From the Main Edit Menu, select F9 to access this menu. This menu is used to create the linker command file (LNK). Values here are based on the previous entries. Discrepancies between format specs and memory requirements are shown in the Memory Summary Display (Figure 3.11). Negative values in the row labeled unused flash on the display. This indicates an error condition. When this happens, either the format specs must be increased, or the C memory specs must be decreased to correct the problem.

Upon completion of this display, the C program is linked using the CMFCL Batch file.
GETTING STARTED

MEMRY & MMARY

<table>
<thead>
<tr>
<th>RAM</th>
<th>NVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1270C</td>
<td>74 38 (total for C &amp; 1B)</td>
</tr>
<tr>
<td>Formatted for</td>
<td>74000</td>
</tr>
<tr>
<td>Required</td>
<td>41574</td>
</tr>
<tr>
<td>Used</td>
<td>7424</td>
</tr>
</tbody>
</table>

(1) This is the amount of NVM available for data file.

FIGURE 3 11 Build Linker File Menu

Link for the MFC

To link the program, exit the C Utility by pressing Escape twice. At the system prompt, type CMFCL and the CSP filename you want to link; this creates the appropriate output files.

After completing this task, go back to CUP. The next section explains the procedures for configuring the module and downloading the program.

Configure the Module

From the CUP opening menu, press F2 (read existing CSP file). Key in the CSP filename, press F1. The first menu of Figure 3 12 appears. From this menu, select F1 Module Management. The second menu of Figure 3 12 appears. This menu is used to initiate the communication between CUP and the MFC. The PCU and module addresses, and CIU/SPM specifications determine how CUP communicates with the MFC. CLP continuously checks the status of the MFC and displays it through a window on the console. If addresses are incorrect, cable connections improper, the module is halted, etc., the message 'CAN'T COMMUNICATE WITH MODULE' appears.
FIGURE 3-12  Configure Module Displays
GETTING STARTED

From the second menu of Figure 3.12, press F1 (select CONFIGURE mode). Press the space bar to continue. Next, press F4 to format the module. It is important to note that doing so will ERASE everything in the module's memory.

C Program Management Download
The linked program can now be downloaded to the MFC. Press Escape, the first menu of Figure 3.12 is back on the display. Select F2 C program management. This menu (Figure 3.13) is used to download the C program. The C program at this point exists in three files: C MAP, code section, and idata section. When F1 is pressed, CUP verifies the file specifications (see the second display of Figure 3.13). Upon successful verification, the download begins. When the download is complete, the system prompts the user.

NOTE: Any existing program files in the MFC are deleted when the download begins.

![Figure 3.13 C Program Download Menu](image-url)
GETTING STARTED

Data File Management
After the program is downloaded to the MFC, the Data File Management (F3 of Figure 3.12) can be used to perform a variety of operations (see Figure 3.14). For example, press F1 to display the module's directory. This directory shows what files, their size, access, etc are currently in the module's NVM file system. Other operations such as copy files, create files, etc. are available. Figure 3.15 is an example of a module directory.

FIGURE 3-14 Display Module's Directory
SECTION 4
THE MFC

GENERAL

This section explains how to set up the MFC for the C environment.

MODULE FORMAT

The MFC has configurable memory of 320 Kbytes of RAM and
80 Kbytes of NVM. This memory is divided between function
blocks and C program structures. The Module Format Table
handles this system structure.

The format operation involves initializing all function block
space, writing values to the module format table, and creating
a new file system and directory based on the entries in the
format table.

The format operation can be done one of two ways, Hardware
or Software. The Hardware (hard format) method is done with
the Option dipswitch (U72 on the CPU Board refer to MFC
Product Instruction for switch information). Set switches 1
and 3 to the "1" position. When this is done, the MFC writes
a default format table and initializes NVM space. The
defaults are:

- 72 bytes of directory (free space and last entry)
- 512 bytes of file data space
- 0 bytes of C User RAM

The Software (soft format) method is done with the CUP
program. From the CSP Edit Menu, press F2 Configure Module
On the ensuing display, press F1 Module Management Select
FORMAT MODULE from this display. The prompt 'SENDING
FORMAT MESSAGE' appears. It is important to note that this
deletes ALL existing configuration data in the NVM.

MFC FILE SYSTEM

The dynamic data file system for C programs and the C Utility
Program resides in the MFC's NVM. File space size and number
of entries available in the directory is determined by the user
during the Module Format task.

File names must be in the range 1 through 32759. When
specified from a C program, filenames are character strings
(refer to open or creat functions in LATTICE manuals) Files
can be opened for reading simultaneously by multiple tasks,
but write access is exclusive. If files need to be shared
between readers and writers, the user must provide some
synchronization. The system open file table has space for
twenty entries.
THE MFC

For error detection purposes, file data and directories have associated checksum fields. Data integrity checks are performed twice: when the MFC is powered up/reset, and when it goes from Configure mode to Execute mode.

The MFC's two serial ports are treated like files by the C program. Therefore, they can be opened for reading/writing by supplying the correct reserved filename.

**File Names/Device Names**

Valid dynamic file names are in the range of 1 to 32759 in ASCII string representation (e.g., '1', '2', '12345').

Valid device names are Serial Port 0, Serial Port 1, and Null Device. The recognized strings for these devices are:

- **Serial Port 0**  'p0'  'P0'  'CON'  'CON'
- **Serial Port 1**  'p1'  'P1'  'PRN'  'PRN'
- **Null Device**  'nul'  'NUL'  'null'  'NULL'

**NOTE:** Functions to access files reside in the MFC's ROM.

**DYNAMIC MEMORY MANAGEMENT**

The MFC03 supports a dynamic memory management scheme. This scheme allows C programs to acquire and free blocks of memory at runtime. Memory is acquired by functions *malloc* or *calloc*, and returned to the heap through the *free* function. The heap (or dynamic memory pool) is sized by the CUP program (see GETTING STARTED, Edit C Memory Specifications for explanation).

**C FUNCTION CODES**

There are two function codes that must be assigned to function blocks in the module's configuration. They are:

- **Function Code 143**  Invoke C
- **Function Code 144**  C Allocation

Function Code 143 must be in the configuration or the C program will not operate.

**Function Code 143  Invoke C**

The purpose of this function block is to invoke the C program. This code calls a function designated as the segment entry point in the C program.
The program can have up to eight segment entry points. Each (e.g., entry point 0 with function block segment 0). A function name (the name of one of the functions that compose the C program) is assigned to each entry point. The same function name may be assigned to more than one entry point.

When an Invoke C block is encountered during function block execution, control is transferred to the C program at the associated entry point. When that function returns, function block execution continues normally at the next logical block. The entry point function may call other functions of the C program before returning.

Invoke C function blocks can be configured in any of all function block segments. Since each function block segment is a separate task, the C program may also be executed by more than one task. When the program is subject to multitasking, all shared functions among tasks must be coded so that they can be reentered.

Table 4.1 lists the specifications for this function code; Table 4.2 the outputs.

**TABLE 4.1  Function Code 143 Invoke C Specifications**

<table>
<thead>
<tr>
<th>Spec No.</th>
<th>Tune</th>
<th>Default Value</th>
<th>Data Type</th>
<th>Range Min Max</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Yes</td>
<td>0</td>
<td>I1</td>
<td>Full</td>
<td>Inhibit Execution Flag #1</td>
</tr>
<tr>
<td>S2</td>
<td>No</td>
<td>0</td>
<td>I2</td>
<td>0-9998</td>
<td>Block Address of Inhibit Execution Flag #2</td>
</tr>
<tr>
<td>S3</td>
<td>Yes</td>
<td>0</td>
<td>I1</td>
<td>Full</td>
<td>Program readable parameter</td>
</tr>
<tr>
<td>S4</td>
<td>Yes</td>
<td>0</td>
<td>I1</td>
<td>Full</td>
<td>Program readable parameter</td>
</tr>
<tr>
<td>S5</td>
<td>Yes</td>
<td>0</td>
<td>I2</td>
<td>Full</td>
<td>Program readable parameter</td>
</tr>
<tr>
<td>S6</td>
<td>Yes</td>
<td>0</td>
<td>I2</td>
<td>Full</td>
<td>Program readable parameter</td>
</tr>
<tr>
<td>S7</td>
<td>Yes</td>
<td>0.000</td>
<td>R(3)</td>
<td>Full</td>
<td>Program readable parameter</td>
</tr>
<tr>
<td>S8</td>
<td>Yes</td>
<td>0.000</td>
<td>R(3)</td>
<td>Full</td>
<td>Program readable parameter</td>
</tr>
<tr>
<td>S9</td>
<td>Yes</td>
<td>0</td>
<td>I1</td>
<td>Full</td>
<td>Spare</td>
</tr>
<tr>
<td>S10</td>
<td>Yes</td>
<td>0</td>
<td>I2</td>
<td>Full</td>
<td>Spare</td>
</tr>
<tr>
<td>S11</td>
<td>Yes</td>
<td>0 000</td>
<td>R(3)</td>
<td>Full</td>
<td>Spare</td>
</tr>
</tbody>
</table>

Module Utilization: 28 bytes NVM, 106 bytes RAM
THE MFC

Function Code 143 has one output with a real data type. Its value may be set by the C program using the putargs function.

Function Code 144 Allocate C
The purpose of this function block is to define the amount of RAM and NVM allocated for C programs.

The amount of RAM is given in specification S1. It should be greater than or equal to the Total RAM allocated to C specification for Memory Format in CUP.

The amount of NVM is given in specification S2. It should be greater than or equal to the Total NVM allocated to C specification for Memory Format in CUP.

<table>
<thead>
<tr>
<th>Spec No.</th>
<th>Tune</th>
<th>Default Value</th>
<th>Data Type</th>
<th>Range Min</th>
<th>Range Max</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>No</td>
<td>0</td>
<td>I2</td>
<td>0</td>
<td>320</td>
<td>RAM Allocation (in 1K increments)</td>
</tr>
<tr>
<td>S2</td>
<td>No</td>
<td>0</td>
<td>I2</td>
<td>0</td>
<td>80</td>
<td>NVM Allocation (in 1K increments)</td>
</tr>
</tbody>
</table>

Module Utilization 12 + (S2 * 1,024) bytes NVM, 36 + (S1 * 1,024) bytes RAM

Function Code 144 has one output with a Boolean data type. This output is unused at the present time.

CPU LEDS

There are eight front panel CPU LEDs that relay error code information. If an error occurs, the LEDs will illuminate in a pattern indicating the specific error. See Table 4.3 for error codes and corrective actions.
<table>
<thead>
<tr>
<th>LED</th>
<th>Hex</th>
<th>Meaning</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 0 0 0</td>
<td>20</td>
<td>Inconsistent Format Table data caused by a configuration restore operation</td>
<td>Put MFC in Configure mode and retry the Restore operation.</td>
</tr>
<tr>
<td>0 0 1 0 0 0 0 1</td>
<td>21</td>
<td>File system error</td>
<td>Put MFC in Configure mode, check module’s file directory for the file in error.</td>
</tr>
<tr>
<td>0 0 1 0 0 0 1 0</td>
<td>22</td>
<td>Invoke C error. Invoke C function code is configured but program files are missing, or Invoke is configured in the wrong function block segment.</td>
<td>Check configuration, check module’s file directory for program files.</td>
</tr>
<tr>
<td>0 0 1 0 0 0 1 1</td>
<td>23</td>
<td>User write violation. The C program attempted to write outside the allowable user address space</td>
<td>Check C program for writes to null pointers, etc.</td>
</tr>
<tr>
<td>0 0 1 0 0 1 0 0</td>
<td>24</td>
<td>Stack overflow detected. The C program wrote past the end of the stack</td>
<td>Check C program for unbounded stack growth</td>
</tr>
</tbody>
</table>
SECTION 5
FUNCTIONS

GENERAL

This section provides a listing of unique MFC runtime support functions.

**bin** input function block output

This function inputs fixed or configured block output values. On return, the caller’s buffer will indicate the block’s data type, whether quality is defined for the block (if defined, the quality and alarm fields are set), and the block’s output value.

**Synopsis:**

```c
#include <n90.h>

status   bin(blockno,bufptr),
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockno</td>
<td>block number to input</td>
</tr>
<tr>
<td>bufptr,</td>
<td>pointer to buffer structure</td>
</tr>
<tr>
<td>int status</td>
<td>return status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ok</td>
</tr>
<tr>
<td>1</td>
<td>block does not exist</td>
</tr>
</tbody>
</table>

**bout** output a value to a function block

This function outputs a value and optional quality/alarm status to a function block. At entry to **bout**, the data type and an indication of whether quality is defined are provided by the caller in the user structure. An error occurs if the block number is out of range or does not exist, if the block is not an OUTPUT BUFFER function code (function code 93, 94, 137, or 138), or if the data type specified does not match with the function block type.

Overflow can occur for type real block outputs because the floating point representation used internally by MFC firmware differs from C program's representation. If the value is too large to be represented, then the largest value of the same sign is output and the return status is set to 1.

**Synopsis:**

```c
#include <n90.h>

status   bout(blockno,bufptr),
```
FUNCTIONS

int blockno;
struct fbuf *bufptr,
int status,

block number to output to
pointer to user structure
return status:

0  ok
1  overflow occurred
1  Error

cpfil  request a file checkpoint operation

This function allows the user to request that a given
file be copied to the backup MFC in a redundant pair.
It is by this mechanism that changes in file data can be
updated in the backup. The mode parameter specifies
the action to take if a buffer error occurs, such as no
buffers being currently available

Note that the file must currently be held open by the
caller. If deferred checkpointing is requested, the MFC
task responsible for sending the copy must gain access
to the file. This means that if the file is open for
writing by the C program, then it must be closed by the
C program after the request has been made so that the
task can acquire the file

synopsis:

status  cpfil(mode, filedescr, fid);

short mode,  error action (no buffer or file too big
  for buffer)

  0  immediate return
  1  request deferred checkpoint and
      return
  2  wait until checkpoint complete

long filedescr, file descriptor (file must be opened)
long fid,  file id number
long status,  return status

  0  ok
  1  no buffer available
  2  file too big for buffer
  3  file read error
  4  problem with backup
**FUNCTIONS**

**fltr 3** convert a float variable to NETWORK 90 real 3 format

The floating point variable pointed to by pr4 will be converted to NETWORK 90 real 3 format. The C program can use this function to convert its internal floating point variables to a format usable by a Computer Interface Unit.

Because a 4 byte representation is being converted to a 3 byte representation, overflow and underflow are possible. If either occurs, default maximum and minimum values are assigned and an indication is returned in the function.

**synopsis**

```c
error fltr3(pr4,pr3),
```

float *pr4; pointer to value to convert
char *pr3; pointer to buffer to fill
int error; overflow/underflow indicator

```c
0 ok
1 underflow, *pr3 LOWLIM
1 overflow, *pr3 HILIM
```

**getargs** get information from Invoke C function block

This function allows the C program to read the Invoke C function code's block number, program readable parameters (S3 through S8), and its tune flag (set to 1 if the block has been tuned). The C program can also read the block output and a private checkpointed save area which the C program can write by the putargs function.

The parblock structure is defined in the file n90.h. Note that the C program need not declare memory for the structure itself; a pointer variable is all that is required.

**synopsis:**

```c
#include <n90.h>

pp = getargs();
struct parblock *pp;
```

pointer to this Invoke's parblock
FUNCTIONS

inque/outque examine the state of the serial port buffers.

The inque function returns the number of bytes (or characters) waiting to be read from an MFC serial port input. The outque function returns the number of bytes waiting to be sent to an MFC serial port for transmission. In both cases, a value of zero indicates the buffer is empty.

These functions provide a means of sensing the state of the serial ports in order to avoid suspension of execution, if other operations can be performed.

Note that the interrupt-driven port handlers in the MFC firmware queue up all characters as they are received (or sent). The default buffer size for C programs is 256 bytes, but can be overridden by user buffer allocation using the portopen function.

synopsis

nbytes inque(portno),
inbytes outque(portno);

int nbytes, number of bytes in buffer
int portno, port number (0 or 1)

portopen open a serial port with user buffer.

This function allows the user to provide buffering for a serial port of a larger size than the default size achieved from the open function. Note that the buffer provided is divided into three pieces: 3810 bytes of overhead and two circular queues (1 for input, 1 for outputs). The default buffer configured by open is 550 bytes (256 bytes for each input and output), and cannot be reclaimed.

synopsis

fd portopen(pn,mode,bufptr,bufsiz)

int fd; > 0 file descriptor
         1 FAIL
int pn; port # (0 or 1)
unsigned short
mode, 0 PMODE (default)
         10 specifies port control
FUNCTIONS

chart *bufptr, user buffer address

int bufsize; must be 50 or more (38 bytes of overhead)

The mode parameter is used to configure the 6850 UART as follows:

FLOW CONTROL
   Hardware Only 0
   Xon/Xoff Enable 128 0 x 80

BREAK DETECT
   Yes (Enabled) 64 0 x 40
   No (Disabled) 0

STOP BITS
   1 Stop Bit 0
   2 Stop Bits 32 0 x 20

PARITY
   None 0
   Odd 4
   Even 8

DATA LENGTH
   7 bits 0 or 2
   8 bits 1 or 3

A mode parameter of 0 gets the default value 0 x c3
A NULL bufptr designates that the system commbuffer be used.

Note that portopen fails if the specified port is already open. Since port 0 is opened automatically at startup for standard input and output, it is necessary to issue fclose calls for stdin, stdout, and stderr prior to a portopen call for port 0.

putargs write the block output and save area to Invoke C.

The C program can write to the Invoke C function block output and to the private save area associated with each Invoke C block. The block output has quality and alarm fields which can be written by this means. The getargs function must be first called to obtain a pointer at the parblock structure which putargs uses.
FUNCTIONS

putmsg write to serial ports with gapless transmission

This function works much like the write function for output to the serial ports. If gapless transmission of the message is required, then putmsg should be used. Note that the port must first be associated with a file descriptor acquired from open or portopen.

synopsis

status putmsg(fd, buffer, length)

int fd: file descriptor
char* buffer: pointer at message
int length: length of message
int status: 1 if bad fd
            nbytes written

rfilet copy remote file to local file

This function copies a remote file' (i.e., a file residing in another module) into a local file.

The specifications of the copy operation are passed to the function in a structure of type 'rffspec.' The status of the operation is passed back to the caller in this structure. The structure is defined in the header 'n90.h,' and is described below.

rffspec

Specified by Caller

short loop, loop address of remote module
short pcu; PCU address of remote module (must be 0
for Plant Loop)
short mod, remote module address
short rfid, remote file ID number
short lfid, local file ID number
short ldscr, local file descriptor (if file is opened
by caller) or 1 (if not opened by caller)
short wait, wait mode
    0 immediate return
    1 wait until done
short maxtime, maximum time (in seconds) allowed for
operation
FUNCTIONS

Returned by Function

short stats;

Status:
1  busy
0  normal completion
1  invalid spec
2  can not open/read remote file
3  local memory overflow
4  can not write local file
5  timeout

synopsis

#include 'n90.h'
void rfile (&specs),
struct rfspec specs,

Miscellaneous Comments

1. This function copies whole file only; will not copy parts of a file

2. If the destination file is opened by the caller, it must be opened for read/write and the caller must not access the file while the status is "busy". When the function completes, the "file position" is indeterminate (the caller must seek to the desired position).

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

3. The function reads the entire file into local dynamic memory before writing to the local file. Therefore, if the read operation is not successful, the local file is not modified

4. Multiple copy requests may be active at one time (immediate returns and/or multiple tasks). There is no built-in limit on the number of concurrent requests

5. Errors detected by the function are classified as fatal (e.g., the specified remote file does not exist) or non-fatal (e.g., the remote module is busy). When a fatal error is detected, the function terminates immediately. When a non-fatal error is detected, the operation which caused the error is retried periodically until it succeeds or the elapsed time (measured from the call to the function) exceeds 'maxtime.'
6. Dynamic memory usage: each active request uses about 180 bytes of local dynamic memory. As described above, a complete copy of a file is read into dynamic memory before being written to the destination file. However, even if multiple requests are active, only one request at a time is allowed to make its copy. The memory usage for a copy depends on several factors but typically is about 1.2 times the file size.

7. The module bus reads are performed by the same task which performs the module bus I/O for function blocks such as Analog Input/Block, Digital Input/Block. A file read requires at least 2 cycles of this task (see function code 90, spec 2).

8. The number of module bus messages required to read a file is at least $4 + n/90$ (where $n$ is the file size).

9. The remote module uses one MBF buffer for each active copy operation. If the entire file fits in an MBF buffer, the remote module locks the file only long enough to read the file into its MBF buffer. Otherwise, the remote module locks the file during the entire module to module transfer. It is usually desirable to avoid this situation by making the MBF buffers large enough to hold the largest file to be transferred.

rfiler  read remote file to RAM.

This function reads a remote file (i.e., a file which resides in another module) into a buffer provided by the caller.

The specifications of the read operation are passed to the function in a structure of type "rfspec." The status of the operation and a byte count are passed back to the caller in this structure. The structure is defined in the header "n90.h" and is described below:

**rfspec**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short loop;</td>
<td>loop address of remote module</td>
</tr>
<tr>
<td>short pcu;</td>
<td>PCU address of remote module (must be 0 for Plant Loop)</td>
</tr>
<tr>
<td>short mod;</td>
<td>remote module address</td>
</tr>
<tr>
<td>short rfid;</td>
<td>remote file ID number</td>
</tr>
<tr>
<td>long ofs;</td>
<td>remote file offset (byte number)</td>
</tr>
<tr>
<td>long cnt.</td>
<td>maximum number of bytes to read (must be less than or equal to size of destination buffer)</td>
</tr>
</tbody>
</table>

Specified by Caller
FUNCTIONS

char *ptr, pointer to destination buffer
short wait, wait mode
0 immediate return
1 wait until done
short maxtime: maximum time (in seconds) allowed for
operation

Returned by Function

short sts, status
1 busy
0 normal completion
1 invalid spec
2 cannot open/read remote file
3 local memory overflow
5 timeout

long nb, number of bytes read

 synopsis

#include n90.h

void rfiler(&specs),
struct rfirspec specs,

Miscellaneous Comments

1. By setting 'ofs' and '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 set 'ofs' to zero and 'cnt' equal to or greater than the file size

2. The file data is read from the remote module in block (usually 90 bytes but sometimes less) and put directly in the destination buffer. Therefore, the buffer should not be accessed while the status is 'busy.' If the final status (0 or neg) indicates an error, the buffer contents are indeterminate and should not be used.

3. Multiple read requests may be active at one time (immediate returns and/or multiple tasks). There is no built-in limit on the number of concurrent requests.

4. Errors detected by the function are classified as fatal (e.g., the specified remote file does not exist) or non-fatal (e.g., the remote module is busy). When a fatal error is detected, the function terminates immediately. When a non-fatal error is detected, the operation which caused the error is retried periodically until it succeeds or the elapsed time (measured from the call to the function) exceeds 'maxtime'.
FUNCTIONS

5 Dynamic memory usage: each active request uses about 140 bytes of local dynamic memory.

6 The module bus reads are performed by the same task which performs the module bus I/O for function blocks (e.g., Analog Input/Block, Digital Input/Block). A file read requires at least 2 cycles of this task (see function code 90, spec 2).

7 The number of module bus messages required to read a file is at least 4 +n/90 (where n is number of bytes read).

8 The remote module uses one MBF buffer for each active read operation. If the requested number of bytes (or the entire file) fits in an MBF buffer, the remote module locks the file only long enough to read the file into its MBF buffer. Otherwise, the remote module locks the file during the entire module to module transfer. It is usually desirable to avoid this situation by making the MBF buffers large enough to hold the largest file to be transferred.

r3flt convert NETWORK 90 real 3 format to floating point

The real 3 value supplied is converted to a real 4 format in IEEE format, and placed into the four bytes pointed to by pr4. A real 3 value received from a CIU can be converted to a format which can be manipulated using arithmetic and floating point operations by a C program by using this function. There is no return value.

 synopsis

void r3flt (pr3, pr4);
char *pr3, pointer to r3 to convert
float *pr4; result

sysclk read MFC 1 millisecond resolution clock

The value returned from sysclk is a 31 bit positive integer. This means that the clock rollovers occur approximately every 596 hours. The value is reset to zero when the MFC is reset, and increases monotonically to maximum value. The next clock tick resets the value returned by sysclk to zero, and the cycle repeats. All time interval applications must take clock rollover into account.
FUNCTIONS

synopsis:

value syseclk(),
(long) int value,

	imer value
RAM STRUCTURE

C RAM
All of the configurable, RAM resident structures related to C are contained in this area.

The C user RAM contains the data structures that are directly accessible by the C program. The C program can read from and write to any address in this space. An attempt to write anywhere outside this space will put the module in Error mode.

The C system RAM contains the program's executable code and other data not directly accessible to the C program.

The total size of this space is established during the Format operation and is specified by the format spec:

\[ \text{total RAM allocated to C} \]

The partitioning between user and system space is established during the program download operation and is determined from the memory specs and the constraint that the size of the user space must be a multiple of 64K.

Function Block RAM
The entire RAM resident function block configuration is contained in this space.

The size of this space is established during the Format operation and is inferred from the format specs:

\[ \text{size of C/FB RAM} \times \text{total RAM allocated to C} \]

Idata Section
This area of RAM holds the C program's initialized data. The idata section file is read into this space during Execute mode start up.

The size of this space is established during the download program operation and is specified by the memory spec:

\[ \text{max size of C's idata section} \]

Udata Section
This area of RAM holds the C program's uninitialized data. This space is initialized to zeroes before the C program is started.
APPENDIX A

The size of this space is established during the download program operation and is specified by the memory spec:

\[ \text{max size of C's udata section} \]

**Segment Stacks and Library Data**

This area of RAM holds the "per task" data space (each function block segment (task) which invokes C must provide its own stack space and C library data space).

The size is established during the download program operation. The total size allocated is determined by the memory specs as follows:

\[ \text{the sum of (segment stack size + 2000) for each specified segment entry point} \]

The amount of stack space required for each task depends on several factors including depth of function nesting and amount of automatic data defined by each function. Allocate at least 2 Kbytes for each stack.

**Dynamic Memory Pool**

This area of RAM is available to the C program for data storage. It is managed by the functions `malloc` and `free`.

The size of this space is established during the download program operation and is specified by the memory spec:

\[ \text{min size of dynamic memory pool} \]

**Checkpoint Buffers**

This area of RAM is used for temporary storage of file data while it is transmitted from a primary MFC to its backup. When a request is made to checkpoint a file, the file data is first copied into an available checkpoint buffer and then is transmitted to the backup from the buffer.

The size is established during the download program operation. The total size allocated is determined by the memory specs:

\[ \text{(number of checkpoint buffers * (size of each checkpoint buffer + 30))} \]

**MBF Buffers**

This area of RAM is used for temporary storage of file data while it is being accessed via the module bus. When a request is made via the module bus to open a dynamic file, the file data is first copied into an available MBF buffer and subsequent reads and writes operate on the copy instead of the actual file.
APPENDIX A

The size is established during the download program operation. The total size allocated is determined by the memory specs as follows:

(number of MBF buffers * (size of each MBF buffer + 40))

Code Section
This area of RAM holds the C program’s executable code. The code section file is read into this space during Execute mode startup.

The size of this space is established during the download program operation and is specified by the memory spec.

Max size of C’s code section

C Function Block RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Area</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>F80000</td>
<td>C user RAM</td>
<td>idata section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>udata section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>segment data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dynamic memory pool</td>
</tr>
<tr>
<td>XX0000</td>
<td>C system RAM</td>
<td>checkpoint buffers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mb buffers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused</td>
</tr>
<tr>
<td></td>
<td></td>
<td>code section</td>
</tr>
<tr>
<td>Function</td>
<td>Block RAM</td>
<td>function block space</td>
</tr>
<tr>
<td>FD0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTES:
1 One segment data area per FB segment (size is zero if no segment entry point is defined).
2 C library segment data (part of segment data area) accessed via A6 (A6 points to base) no initialization
3 Stack accessed via A7 (stack pointer) and A5 (frame pointer)
APPENDIX A

NVM STRUCTURE

Module Format Table
This table contains information specifying the format of the
module (e.g., amount of NVM allocated to C, max number of
data files) It is initialized by the Format operation. Its size
is 34 bytes.

File Directory
This directory contains the system information for the current
files. It is made up of fixed sized records with one record for
each existing or potential file. It is initialized by the Format
operation.

The size established by the Format operation is derived from
the format specs.

\[ 36 \times (5 + \text{max number of data files}) \text{ bytes} \]

File Space
This area of NVM contains the C program files and the data
files. Each file is composed of a linked list of fixed sized
blocks (256 bytes). Unused file space is organized as a
special file called the free list. Initialized by the Format
operation.

The size of the file space is established by the Format
operation and is specified by the format spec:

\[ \text{total NVM allocated to C (in bytes)} \]

Function Block Space
This area of NVM contains the function block configuration.
The Format operation or EE initialization can be used to
initialize this space.

The size of this space is established by the Format operation.
The size is not explicitly specified; it is the remaining NVM
space not assigned to the other structures.
**APPENDIX A**

**NVM Structure**

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>68000 to 68222</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>Module Format Table</td>
</tr>
<tr>
<td></td>
<td>Directory</td>
</tr>
<tr>
<td></td>
<td>File Space</td>
</tr>
<tr>
<td></td>
<td>Function Block space</td>
</tr>
<tr>
<td>7BFFFF</td>
<td></td>
</tr>
</tbody>
</table>

**File ID**

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32759 data files</td>
</tr>
<tr>
<td>32760-32764</td>
<td>reserved, unused</td>
</tr>
<tr>
<td>32765</td>
<td>C code file (section 0)</td>
</tr>
<tr>
<td>32766</td>
<td>C data file (section 1)</td>
</tr>
<tr>
<td>32767</td>
<td>C map file</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>C Compiler</td>
<td>There are two compilers Native and Cross. Native produces code for Workstation execution, Cross produces code for MFC execution.</td>
</tr>
<tr>
<td>C file</td>
<td><code>&lt;filename&gt;</code> C the Workstation resident text file that contains a file to be read by the C compiler (C program source file).</td>
</tr>
<tr>
<td>checkpoint buffers</td>
<td>MFC module buffers that provide temporary storage for file data while it is transmitted from primary MFC to backup MFC.</td>
</tr>
<tr>
<td>C map file</td>
<td>MFC resident file containing memory map data of C program. Written during the download function (File ID 32767).</td>
</tr>
<tr>
<td>C RAM</td>
<td>Two parts C user RAM and C system RAM. Continuous block of RAM in the MFC containing all configurable, RAM resident structures of the C environment.</td>
</tr>
<tr>
<td>code section</td>
<td>The executable code of a C program (also known as text section).</td>
</tr>
<tr>
<td>code section file</td>
<td>MFC resident file (file ID 32765) that contains the C program's code section. This file is written to the MFC during the program download operation. The file contents are derived from the LMS and MAP files.</td>
</tr>
<tr>
<td>CSP file</td>
<td>The Workstation resident file containing information about the C environment for the MFC. Information such as format specs, memory specs, object files, etc. is found in this file. A CSP file exists for each target MFC.</td>
</tr>
<tr>
<td>CUP</td>
<td>Acronym for C Utility Program, the Bailey supplied software that is used to specify and manage the C environment of the MFC module.</td>
</tr>
</tbody>
</table>
GLOSSARY

C system RAM
The part of C RAM containing the code section, checkpoint buffers, and MBF buffers.

C user RAM
The part of C RAM containing data structures that are directly accessible to the C program. Included are idata, udata, segment stacks, and C library data, and dynamic memory pool. This is the only part of MFC address area that the C program can write to.

dynamic file
An MFC file that can be modified while the module is in the Execute Mode.

file attributes
Certain file characteristics that are specified when the file is created, for example, write access type.

file ID
Unique MFC file identifier. The following IDs are permanently assigned:

0  free list
32765  code section file
32766  idata section file
32767  udata section file

Data file IDs (1 32759) are assigned by the user when the files are created.

format operation
Operation performed on the MFC to format and initialize its user configurable memory. This operation establishes the major partitions of the configurable RAM and NVM (i.e., the boundaries between C and function blocks), deletes all files and initializes the function block configuration.

The format operation can be performed one of two ways:

1. via the module bus (through CUP)
2. by hardware reset with switch U72 set for the format special operation 0C0CCCCC

format specs
The set of specifications that define the MFC's memory format. The format specs are stored in the module's CSP file.
function block Address in MFC memory containing a function code

function block segment A group of function blocks that are associated with the same Segment Control Block (function code R2); belonging to the same scan cycle. Each function block segment is executed by its own, independent ROS task.

idata section Refers to the initialized data of a C program.

idata section file MFC resident file (file ID 32766) that contains the C program’s initialized data. This file is written to the MFC during the program download operation. The contents of the file are derived from the LMS file.

lcm1/lcm2 LATTICE C compiler for the 68000 processor. This compiler runs on the Workstation. It is implemented in 2 phases: lcm1 and lcm2. Phase 1 reads a C source module (C file) and produces an intermediate quad file (Q file). Phase 2 reads the Q file (then deletes it) and produces an object file (O file).

linker A program that combines separately produced object modules into a single load module. There are two linkers:

DOS link LATTICE load module builder

Both linkers execute on the Workstation. The distinction between them is their target environments. The target of the DOS linker is the Workstation; the load module builder is the MFC 68000 processor.
GLOSSARY

LMS file
<filename>.LMS  A Workstation resident file that contains a C program 'load module'. The load module contains the C program's code section (the executable code) and the initialized data (idata) section. This file and related MAP file is produced by the load module builder. During the program download operation, the C Utility Program reads the LMS and MAP files and transmits the appropriate information to the MFC.

LNK file
<filename>.LNK  A Workstation resident text file that contains the commands for the Load Module Builder (LMB command file). The LNK file is produced by the C Utility Program from information in the csp file.

load module builder
(LMB) Workstation resident linker program that combines a set of object modules (O files) into a single load module (LMS file) that can be downloaded to an MFC.

Example of lmb invoke command:
lmb68k <**.lnk>** map ** lms

where "**" is a filename (same as csp)

MAP file
<filename>.MAP  A Workstation resident text file which contains the memory map of a C program load module. The map indicates addresses and sizes of the load module's control sections and the addresses of the global symbols. The MAP file is produced by the Load Module Builder. During the program download operation, the C Utility Program reads the MAP file to verify the control section locations and to determine the addresses of the segment entry points.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBF buffers</td>
<td>MFC resident buffers that provide temporary storage of file data while the file is being accessed via the module bus. When a request is made via the module bus to open a dynamic file, the file data is first copied into an available MBF buffer and subsequent reads and writes operate on the buffer instead of the actual file.</td>
</tr>
<tr>
<td>memory specs</td>
<td>The set of specifications that determine the structure of an MFC's C RAM. The memory specs for a particular MFC are stored in its csp file.</td>
</tr>
<tr>
<td>NVM</td>
<td>The MFC module's configurable non volatile memory (battery backed RAM).</td>
</tr>
<tr>
<td>O file</td>
<td>&lt;filename&gt; O Workstation resident file containing a C object module (i.e., a file produced by the C compiler).</td>
</tr>
<tr>
<td>segment entry point</td>
<td>The function in the C program that is called by an Invoke C function block. For each function block segment with one or more Invoke C blocks, there is one entry point into the C program. All Invoke C blocks in a segment call the same entry point. An entry point may be shared among 2 or more segments.</td>
</tr>
<tr>
<td>segment stack</td>
<td>An area in RAM used by the C program to store temporary data. There is one segment stack for each function block segment (task) which invokes C.</td>
</tr>
<tr>
<td>udata section</td>
<td>The uninitialized data of a C program.</td>
</tr>
<tr>
<td>write access</td>
<td>MFC file attribute that controls write access to the file. There are 3 types:</td>
</tr>
<tr>
<td></td>
<td>UW unrestricted write. The file can be written by the local C program and via the module bus.</td>
</tr>
<tr>
<td></td>
<td>LW - local write. The file can be written by the local C program but not via the module bus.</td>
</tr>
</tbody>
</table>
RO  read only. The file cannot be written by the local C program or via the module bus (although the file can be written, it can be initialized via the module bus).