

Developer's Guide

for The Major BBS[®]

Version 6.2
January 1994



Galacticomm, Inc. • 4101 SW 47th Avenue • Suite 101 • Fort Lauderdale, FL • 33314
Voice: (305) 583-5990 • U.S. Sales: (800) 328-1128 • Fax: (305) 583-7846
BBS: (305) 583-7808 • Technical Support: (305) 321-2404

Copyright (C) 1989-1994 by Galaticomm, Inc.

All rights reserved. No portion of this document or the accompanying software may be reproduced or stored in any medium without prior written authorization from Galaticomm, Inc., except by a reviewer who wishes to quote brief passages in connection with a review for a newspaper or magazine.

Information in this document is subject to change without notice. This document and any related software are sold "as is". Galaticomm, Inc. makes no representations or warranties with respect to the contents of this document, or to the software described, and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Liability for the information in this document, and for the software described herein, shall be limited to the purchase price of the document or software.

This document discusses computer software and other copyrighted materials that may be restricted by license agreements or other protections. This document shall not be used as a written exception to any license agreement, and shall not be construed by any party as a license, authorization, or waiver of rights by any owner or copyright holder of the software discussed.

The Major BBS is a registered trademark of Galaticomm, Inc. Galaticomm, the Galaticomm logo, Advanced LAN Option, BBSDRAW, CNF, Entertainment Collection, GalactiBoard, GalactiBox, Locks and Keys, Major Gateway/Internet, Menu Tree, The Major Database, User Six-Pack, and X.25 Software Option are trademarks of Galaticomm, Inc. RIPscrip, RIPaint, and RIPterm are trademarks of TeleGrafix Communications, Inc. All other products are trademarks or registered trademarks of their respective companies.

CONTENTS

1. OVERVIEW	1
Requirements	1
Registering Names with Galacticomm	2
Choosing a Developer-ID	3
Installation	4
Installation Software for your own Add-on Option	8
.RLN Release Notes	8
How .MSG files are Updated	8
2. DEVELOPMENT ENVIRONMENT	11
Directory and File Structure	13
C Source Conventions	16
Rebuilding MAJORBBS.EXE and standard .DLL files	17
Creating a New .DLL	19
Rebuilding Your .DLL File	21
Versions	21
3. OPERATING ENVIRONMENT	23
Module Definition Files (.MDF)	23
Initialization Routine (init_xxx())	29
Modules	29
Channel Numbering and Grouping	39
Data Structures and Memory Allocation	41
Volatile Data Area	46
Ways to Split up a Long Task	48
File Handles (fopen())	50
Exception Handling (catastro())	51
Languages	53
Creating CNF Options	55
Compiling CNF Options	64
Using CNF Options	65
4. USER INTERFACE	69
User Output (prf(), prfmsg())	69
Multilingual User Output	70
Defining Text Variables	73
User Input	74
Profanity	75
Echo	75
Command Concatenation	75
User-ID Cross Referencing	79
Default Selection Character	80
User Status and Handling	81
Hanging up on a User	83
Intercepting User-Connect	84
Autosensor Routines	86

5. USER SERVICES	90
Security (Locks & Keys)	90
Registerable Pseudo-Keys	92
Accounting (credits)	93
Global Commands	96
Full Screen Editor	99
Full Screen Data Entry	103
File Transfer	107
Uploads	107
ASCII Downloads	120
Downloads	121
File Transfer Protocol	136
6. OPERATOR INTERFACE	137
Video Output (printf())	137
Keyboard Input (getchc())	144
Cursor	145
7. OPERATOR SERVICES	146
Statistics	146
Audit Trail	148
Channel Status Reporting	148
8. DATABASES	149
Database Functions (xxxbtv())	149
System Variables Database	157
User Account Database	158
User Class Database	159
Generic User Database	160
9. OFFLINE UTILITIES	162
Window output (explode())	162
Window input (edtval(), choose())	164
Large Model Programming	168
Language Editor DLLs	169
.MSG File Reading and Writing	172
10. MORE ROUTINES AND VARIABLES	174
Character and String Routines	174
Real-Time Routines (rtkick(), rtihdlr(), interrupts)	177
Time and Date Routines	178
Numeric Routines	179
Text File Scanning	180
Disk I/O	181
Everything Else	184
11. RELIABILITY	185
Some Philosophy on Debugging	185
Programming Tips for The Major BBS	185
General Protection Faults	186
INDEX	196

1. OVERVIEW

This guide could be useful to you in two ways:

Developing	Programming an Add-on Option for The Major BBS
Customizing	Custom tailoring the functionality of one specific BBS

The main purpose of this manual is to help developers create new products for The Major BBS. But it will also be helpful if you're running a BBS and you want to customize it by making changes or additions to the source code or other aspects.

This guide is written assuming:

- o You have purchased The Major BBS and the Developer's C Source Kit.
- o You have read the System Operations Manual for The Major BBS.
- o You are proficient in using The Major BBS (via modem or other interface).
- o You are proficient in operating The Major BBS (from the console).
- o You are proficient in the C language (we recommend "The C Programming Language," by Kernighan & Ritchie).
- o You are proficient at using DOS on the IBM PC and compatibles.

Knowing assembly language can be helpful too, but it's by no means required to make extensive use of this development environment.

Requirements

To set up your development environment for developing your own Add-on Options for The Major BBS, you will need:

- o 80386, 486, or Pentium IBM style computer with at least 4 Megabytes memory (1 Mb real, 3 Mb extended)
- o 60 Megabytes hard disk minimum, but you may want 100 Megabytes or more
- o DOS version 3.3 or higher
- o Borland C++ compiler version 3.1 (see page 21 for a discussion of versions)

- o Phar Lap 286|DOS-Extender SDK V3.0 or higher
- o The Major BBS version 6.2 or higher
- o The Developer's C Source Kit for The Major BBS, which includes:
 - C source code for The Major BBS
 - supporting subroutine libraries
 - supporting batch, make, and other files
- o The Extended C Source Suite for The Major BBS (optional), which includes:
 - C source code for the GCOMM.LIB subroutine library
 - C source code for numerous offline utilities and supporting programs
- o A text file editor of your choice, such as:
 - KEdit, by the Mansfield Software Group
 - QEdit, by SemWare in Marietta, GA
 - Personal Editor, by Personally Developed Software
 - EDIT (comes with DOS 5.0 or later)
 - EDLIN (comes with all versions of DOS)
- o Btrieve from Novell (optional; required if you need to create your own databases)

Except for the computer, Btrieve, and the text editor of your choice, all of these products are available from Galacticomm.

To develop Flash games for The Major BBS, contact Galacticomm. This does not require source code licenses.

Registering Names with Galacticomm

To avoid conflicts between the names used in Add-on Options for The Major BBS from various third party developers, Galacticomm will register the names used in your finished products. This starts with your Developer-ID (more below). Besides Developer-IDs, here are a few of the kinds of things that could run into naming conflicts:

- o Module names (from .MDF files, see page 23)
- o Text variable names (see page 73)
- o New CNF levels (see page 55)
- o Statistics screen names (see page 146)

Call Galacticomm at (305) 583-5990 and ask for Third-Party Developer Services to register your Developer-ID and other names you are using.

Choosing a Developer-ID

Galacticomm maintains a master list of 3-character Developer-ID's.

All files that you supply on your product diskette (including compressed files and the files inside of compressed files) and that are created during the execution of the BBS should have names that start with your three-letter Developer-ID.

See the exceptions for INSTALL.EXE and DISK1.DID, etc., below.

All files associated with running an Add-on Option for The Major BBS should strictly adhere to this convention to avoid file name collisions.

Source and object file names should ideally also use the Developer-ID prefix to avoid name collision. Even if you use a separate directory for your source files, your object files will reside in the same directory as all other object files from all other developers (\BBSV6\PHOBJ or \BBSV6\LOBJ).

You may choose not to use your Developer-ID prefix on source files. The need for unique source and object file names is not as strict as the need for unique names of files associated with running the BBS and Add-on Options. It would be a disaster for a BBS operator to buy two Add-on Options from different sources and have them not work because of a file name collision. It would be an annoyance for someone who buys your source code to run into name collisions. The latter is going to happen to a far smaller group of people, and those people (BBS developers or BBS customizers) are more likely to be able to recover from it themselves by manually renaming some of the files.

These Developer-ID's are reserved by Galacticomm:

BBS Global or internal purposes
GAL Standard and Add-on Options from Galacticomm

CNF LOC MTH \\
MAJ EMU MDF \ other reserved prefixes
MJR CAT /
BTR GP /

INS Reserved for INSTALL.EXE or INSTALL.BAT on the floppy disks
that we ship, or that you ship
DIS Reserved for DISK n .DID files on floppy disks
GHO Reserved for Galacticomm Host program for Doors

In this manual we'll use "DDD" to represent an example of a Developer-ID:

DDD example Developer-ID

For Sysops, or anyone developing something to run on one BBS only, the following Developer-ID is available:

ZZZ Sysop Developer-ID

Installation

The following procedure has been designed to apply to everyone who purchases C source code from Galacticomm. It was designed for non-programmers, but it assumes you already have a way to edit text files like .BAT batch files.

There are several products discussed here, and several sections may or may not apply to you. Whether you're building a development environment for the basic BBS, or for a BBS with dozens of Add-on Options, including some of your own, please use this as your central reference, and follow the steps very carefully.

To create a development environment for The Major BBS on your computer:

1. Install The Major BBS. Put the first disk in your A: floppy drive and type:

A:INSTALL

We strongly recommend you use the default directory "\BBSV6" for your development environment. (If you use a different directory, you'll have to edit all the .BAT, .LNK, .MAK, .DEF, and .CFG files. See page 13.) The \BBSV6 directory will be created automatically for you if it does not already exist.

You can install from a different floppy drive by typing:

B:INSTALL

or Q:INSTALL, or whatever. This will be true for installing software from any disks from Galacticomm, even though we'll just talk about A: in the rest of these steps.

Before you install the source code and other items, it might be a good idea to try running The Major BBS and getting it on the air. See the System Operations Manual for details.

2. Install the Borland C++ Compiler. Put the INSTALL DISK in your A: floppy drive and type:

A:INSTALL

We recommend that you use the default directory "\BORLANDC". This directory will be created automatically for you if it does not exist already.

After installing the compiler, create a TLINK.CFG file and a TURBOC.CFG file in your \BORLANDC\BIN directory (or replace them if they're there already) with these contents:

TLINK.CFG

-L\BORLANDC\LIB;\RUN286\BC3\LIB;\BBSV6\DLIB

TURBOC.CFG

-I\BORLANDC\INCLUDE;\RUN286\INC;\BBSV6\SRC
-L\BORLANDC\LIB;\RUN286\BC3\LIB;\BBSV6\DLIB

3. Install the Phar Lap 286|DOS-Extender SDK. Put the Software Development Kit disk 1 in the A: floppy drive and type:

```
A:INSTALL
```

Use the default "\RUN286" directory (it will be created for you, as well as its subdirectories). When it comes time to choose what to install, pick these:

```
286|DOS Extender Binaries   YES
MS C Support                 NO
MS Fortran Support          NO
Borland C++ Support         YES
MS C Examples               NO
Borland C++ Examples       NO
```

4. Put the compiler and DOS-Extender BIN subdirectories and the \BBSV6\SRC directory in your path statement. For example, you could have something like this in your AUTOEXEC.BAT file:

```
AUTOEXEC.BAT
:
PATH=C:\DOS;C:\BORLANDC\BIN;C:\RUN286\BIN;C:\BBSV6\SRC
:
```

(If you make changes to AUTOEXEC.BAT, remember to reboot your computer so they take effect.)

5. Install Borland C Huge Model Support. Download the huge model support files from Phar Lap's BBS at (617) 661-1009. Look in the File Library in the "GALACT" LIB and download the file named "PLPAT2.ZIP".

Unzip the files in this ZIP file and run the MKLIB batch file:

```
CD \RUN286\BC3\LIB
```

(download or copy PLPAT2.ZIP into this directory)

```
\BBSV6\PKUNZIP PLPAT2.ZIP
MKLIB H \BORLANDC\LIB
```

If MKLIB gives you a "file not found" message, it's OK.

6. Install The Major BBS Developer's C Source Kit. Put the first disk in your A: floppy drive and type:

```
A:INSTALL
```

If you have the Extended C Source Suite, do steps 7 and 8:

7. Install The Major BBS Extended C Source Suite. Put the first disk in your A: floppy drive and type:

```
A:INSTALL
```

8. For every new \BBSV6\SRC*.MAK file that comes with the Extended C Source Suite, insert the following lines at the end of the master make file \BBSV6\SRC\MAKEBBS.BAT:

```
cd \bbsv6\src
make -fbbsxyz.mak
```

(Instead of "bbsxyz.mak" use the name of each .MAK file that comes with the Extended C Source Suite.)

This step isn't critical, but you may find it handy to be able to run MAKEBBS to reliably compile and link whatever needs compiling and linking.

And if you purchase Btrieve from Novell (for your own databases), do step 9:

9. Install the Btrieve database development package from Novell.

If you have purchased the C source code for any Add-on Options from Galacticomm, do steps 10-12:

10. Install the Add-on Options. Put the first disk in your A: floppy drive and type:

```
A:INSTALL
```

11. Install the Add-on Option C Source code. Put the first disk in your A: floppy drive and type:

```
A:INSTALL
```

12. Add the appropriate lines to the end of \BBSV6\SRC\MAKEBBS.BAT to invoke the .MAK file for the option, such as:

```
cd \bbsv6\src
make -fgalxyz.mak
```

(Instead of "galxyz.mak" use the name of whatever .MAK file comes with the C source code of the Add-on Option.)

This step will allow you to use MAKEBBS to properly follow up on any and all changes you may make to the source files or other files in the Add-on Options.

To develop your own Add-on Option, you can either use our \BBSV6\SRC directory for your source and support files, or you can make your own subdirectory.

Choose one of these (either 13a or 13b):

- 13a. Put your C source code in \BBSV6\SRC.

Add lines onto the end of \BBSV6\SRC\MAKEBBS.BAT that invoke your make files, such as:

```
cd \bbsv6\src
make -fdddxyz.mak
```

(Instead of "dddxyz.mak" use the name of your .MAK file.)

or:

- 13b. Create a subdirectory for your source code with the same name as your Developer-ID. For example, if your Developer-ID is "DDD", create:

```
\BBSV6\DDD      C source code for your Add-on Option
```

You can also add lines onto the end of \BBSV6\SRC\MAKEBBS.BAT that invoke your make files, such as:

```
cd \bbsv6\ddd
make -fdddxyz.mak
```

(Instead of "dddxyz.mak" use the name of your .MAK file.)

Whichever method you choose: \BBSV6\SRC or \BBSV6\DDD for your source code, try to use unique names for your source files. Either way, all object files end up in \BBSV6\PHOBJ: yours, and those of other developers. To minimize the probability of name collisions, it's a good idea to use your Developer-ID as a prefix to all your source file names.

Now, test your development environment:

14. Run \BBSV6\SRC\MAKEBBS.BAT:

```
CD \BBSV6\SRC
MAKEBBS
```

This will compile and link \BBSV6\MAJORBBS.EXE, plus almost all \BBSV6*.DLL files, and some \BBSV6*.EXE offline utilities. The process may take many minutes depending on your computer's processing power. Keep a sharp eye out for warnings or error messages.

15. Bring up the BBS and exercise it (try a file transfer over the modem, or over the LAN, for example).

Congratulations! Your development environment for The Major BBS is up and running!

Installation Software for your own Add-on Option

The INSTALL.EXE program we use has some general purpose capabilities that make it useful for just about any Add-on Option for The Major BBS. As with all Galacticomm code, you can only use INSTALL.EXE in your product under special license arrangement with Galacticomm. Assuming you have reached such an arrangement, here are the technical details.

All files must be combined into .ZIP compressed files on your release floppy diskette(s), with only a few exceptions that we'll discuss here. For one, put on your first diskette a file called DISK1.DID, for example:

```
DISK1.DID  
2 150 7000000
```

Here is what these three numbers represent:

- 2 - total number of diskettes for this Add-on Option
- 150 - total number of all files inside of the .ZIP files on all diskettes (inner .ZIP files are not unzipped)
- 7000000 - approximate total hard disk space that will be required to complete installation

Other disks must contain files named DISK2.DID, DISK3.DID, etc., but the contents of these files don't matter. If your .ZIP files require a certain version of PKUNZIP, then put PKUNZIP.EXE on the last disk in your set (outside of any .ZIP files of course).

To run your own batch file or program after INSTALL.EXE completes, be sure to include at least one .MDF file with an "Install:" directive (see page 24 for details) somewhere in one of your .ZIP files.

Note: The BBS automatically deletes whatever file is named in an .MDF "Install:" directive after it's done running.

.RLN Release Notes

You can include release notes in an .RLN file in one of your .ZIP files. INSTALL will automatically display them to the operator. We use this to display BBSMAIN.RLN to Sysops when they first install the BBS. That file contains very up-to-date information and it makes sure the Sysop gets a chance to read it. Note: be sure your lines are no more than 76 characters long.

How .MSG files are Updated

Any .MSG files included in your .ZIP files are automatically merged with .MSG files that exist already on the Sysop's system. This way their offline CNF option settings are preserved when they update to a new version of your software.

See page 55 about creating .MSG files. Here's an example of how it works: Suppose you supply a DDDSAT.MSG file with an option named HOOPLA{ }. If a DDDSAT.MSG file with a HOOPLA{ } option already exists on the Sysop's BBS, then the contents of that option (what's between the curly braces) in the Sysop's old DDDSAT.MSG displace those from your release disks. In all other

respects (option type, description, help message), the contents of your DDDSAT.MSG file are updated on the Sysop's system.

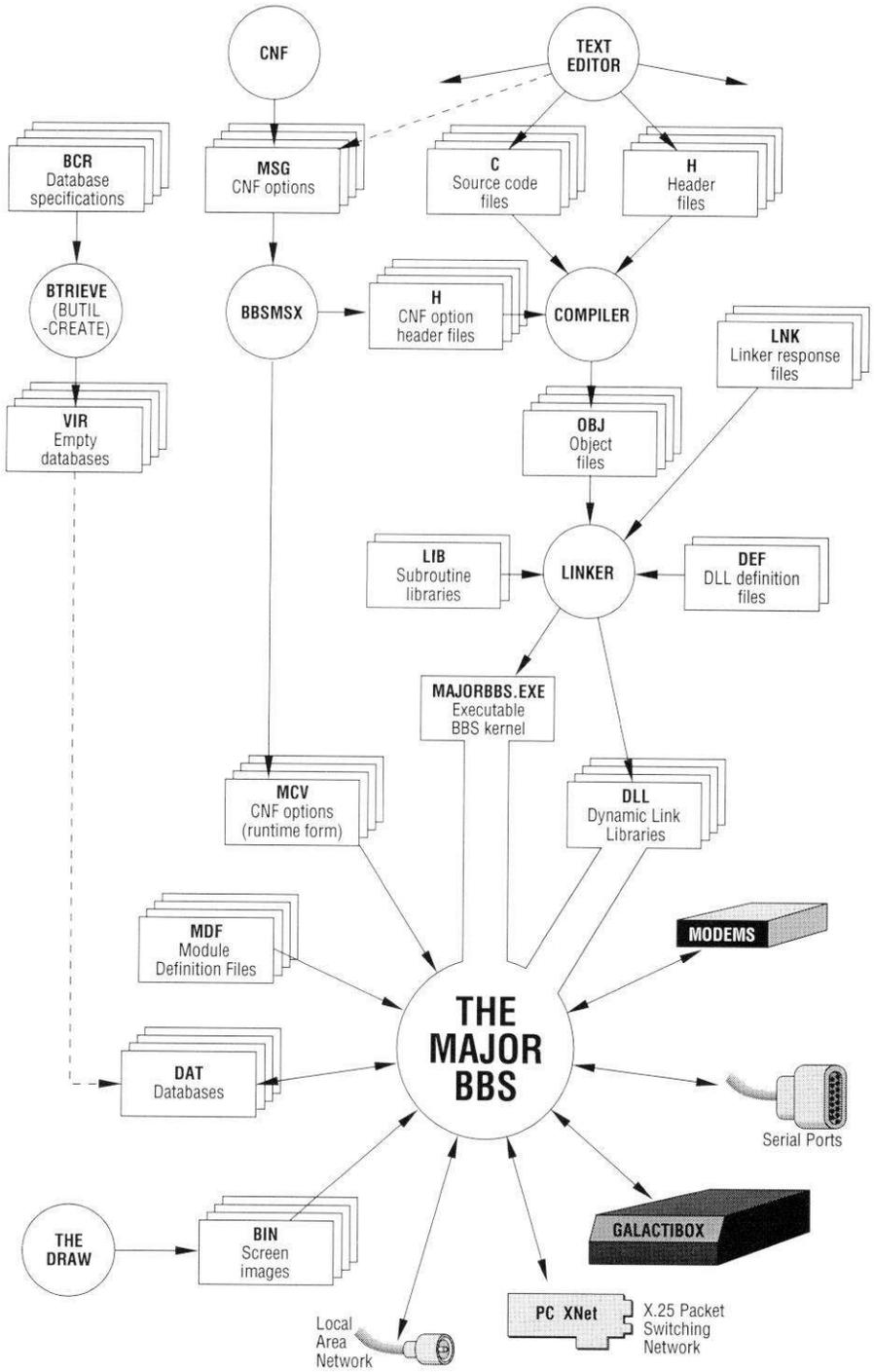
An alternate form of the .DID files on each of your release diskettes adds a language name at the end. For example:

```
DISK1.DID  
2 150 7000000 English/RIP
```

This is used in certain special situations. When a language name appears as the fourth parameter in a .DID file, it tells the INSTALL program to be sure to force all text in that language onto the BBS, and not to mix it with any text the Sysop may already have in that language.

This is especially helpful, for example, with English/RIP text block design and updating. There are many cases where the function of one English/RIP text block depends on the contents of another text block, and a system with a mix of old and new English/RIP versions of text blocks won't work very well.

You can use this feature with any user-language, such as "Spanish/RIP" or "German/RIP". If you use this feature in your Add-on Option, the Sysop will be given the option (called "UPDATE NEW") of mixing his text in the specified user-language with yours, but he'll be strongly encouraged to allow all of your text to take precedence (that option being called "UPDATE ALL").



Development and Runtime Structure of The Major BBS

2. DEVELOPMENT ENVIRONMENT

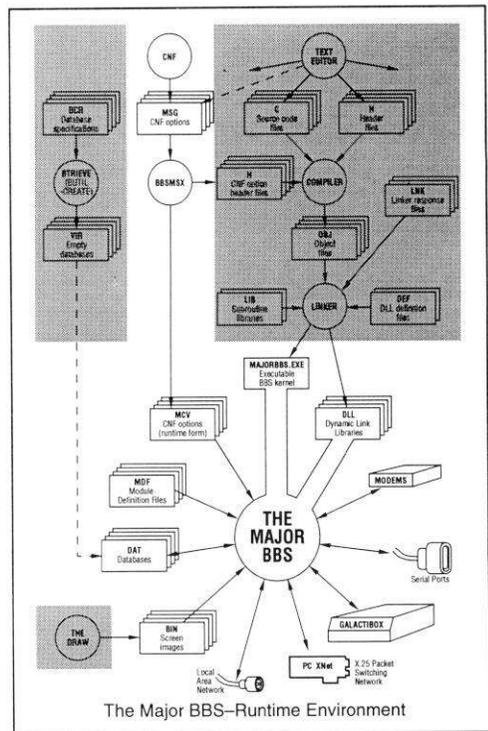
The Runtime Environment

The Major BBS runs with a large supporting cast of disk files. MAJORBBS.EXE is the kernel of the BBS with multi-user support, user and console interfaces, security, accounting, menuing, and online services like file transfer and message editing, as well as numerous utility subroutines.

.DLL (Dynamic Link Library) files contain software for various online services like Electronic Mail and Forums (together in GALMSG.DLL), Teleconferencing, File Library.

.MDF files are Module Definition Files. These are the starting points for all operations of the BBS, specifying: what user services are available when Sysops design their Menu Trees; what text files to use for the BBSUSER.DOC online user manual; what databases, CNF options, text blocks, and utilities to use; and what happens at installation, at the nightly auto-cleanup, or at other timed events.

.DAT Databases maintain information on users, messages, online files and more. .MCV files store prompts, menus, and configuration options. .BIN files store screen image backgrounds or statistical graphs. You can edit .BIN files with a shareware program like TheDraw.



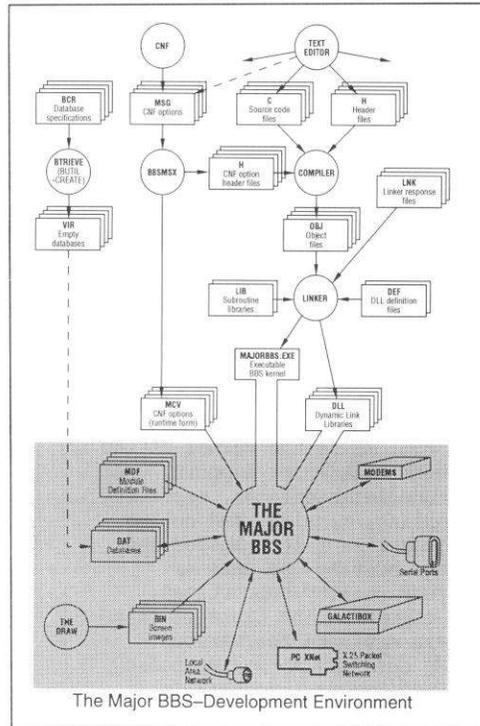
The Development Environment

Behind the scenes of the files that support running The Major BBS are a host of files on the development system. Almost everything starts with a text editor and a text file. Various programs (shown as circles) process text files and turn them into other things. At the heart of the matter, C source code in the .C and .H files gets compiled into object files. The object files get linked with each other and with subroutine libraries to form executable code: either in the MAJORBBS.EXE kernel, or the .DLL files.

Also behind the scenes, .MSG files encode CNF options and text blocks. These are referenced in the source code using codes defined in .H files, and used online by reading from the indexed .MCV files. BBSMSX converts .MSG files into .MCV files.

If you're creating your own databases, you can use Btrieve from Novell to generate empty .VIR database files from the .BCR database specifications. The install process first copies .VIR files to .DAT files to start off the databases (it knows to do this if the .DAT file doesn't exist yet).

See page 162 for the development environment for offline utilities.



Directory and File Structure

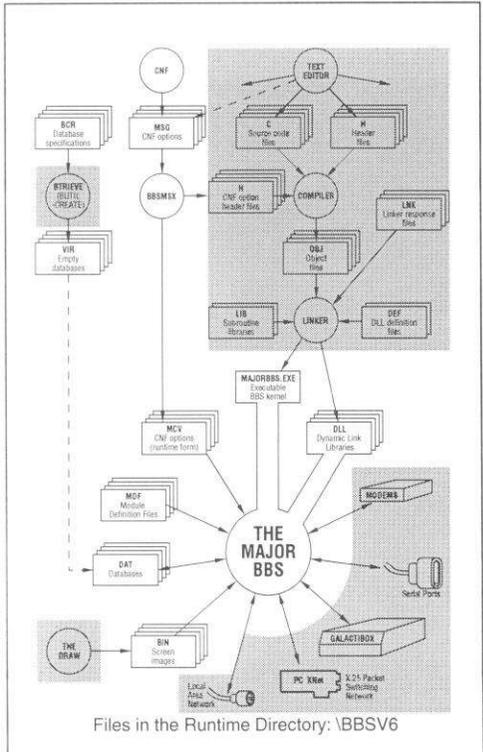
Directory Structure for The Major BBS Development Environment

<pre> \BBSV6 ├── \HELLO ├── \EMAIL ├── \MAIN ├── ... ├── \SRC ├── \PHOBJ ├── \LOBJ ├── \DLIB └── \LIBSRC \BORLANDC ├── \BIN ├── \INCLUDE ├── \LIB ├── ... \RUN286 ├── \BIN ├── \BC3\LIB ├── \SRC ├── \INC ├── ... </pre>	<p>Runtime directory for The Major BBS /Hello Forum message attachments E-mail attachments Files in the MAIN LIB Subdirectories for other Forums and LIBs</p> <p>C source files (.C .H), Linker response files (.LNK) Object files for Phar Lap Huge model (.OBJ) Object files for Large model (.OBJ) Libraries (.LIB) and DLL definition files (.DEF) C source files for PHGCOMM.LIB (extended source pkg)</p> <p>Compiler, linker, librarian, make, assembler (.EXE) Include files (.H) Libraries (.LIB)</p> <p>Phar Lap 286 Loader programs Libraries for Borland C++ version 3 Source code Include files</p>
--	--

We recommend the directory name "\BBSV6" for running The Major BBS, but this is your option. You can easily run The Major BBS from any directory on any drive. However, when developing for The Major BBS, it's much more work to use a directory structure other than one that's based on \BBSV6. (If there are very good reasons why you don't want to use \BBSV6, you'll need to use a text editor to modify all batch files, link files, make files, and compiler configuration files (.BAT, .LNK, .MAK, .DEF, .CFG). Also, before you even install these files from the source diskette, you'll need to modify the INSTALL.BAT file on the source diskette to refer to the directory you're using that's other than "\BBSV6".)

The Runtime Directory

Almost all of the files that The Major BBS needs to run will be located in the \BBSV6 subdirectory. This includes the

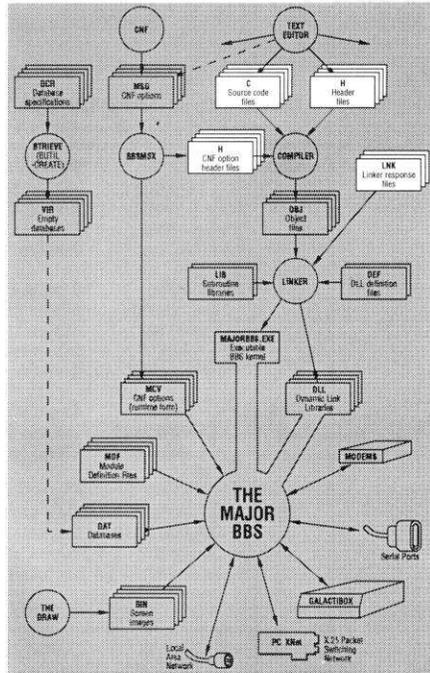


main executable file (MAJORBBS.EXE), all dynamic link libraries (*.DLL), all databases (*.DAT), and all CNF options (*.MCV) plus many more. See page 3 for conventions on file naming with Developer-ID's. See page 15 for the conventions on file extensions. When you run The Major BBS, don't execute MAJORBBS.EXE directly, use BBS.BAT instead.

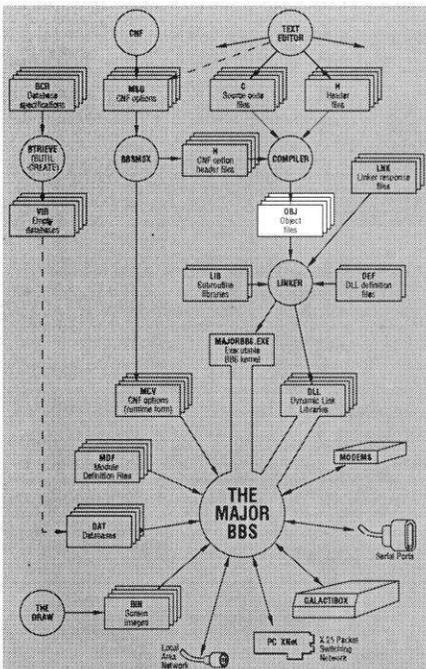
The Development Directories

For The Major BBS and all Add-on Options from Galacticomm, the sub-directory \BBSV6\SRC will hold the source files and \BBSV6\PHOBJ will hold the object files. For all online source code, the compiler huge model will be used, along with Phar Lap 286|DOS-Extender enhancements. Subroutine libraries are in \BBSV6\DLIB.

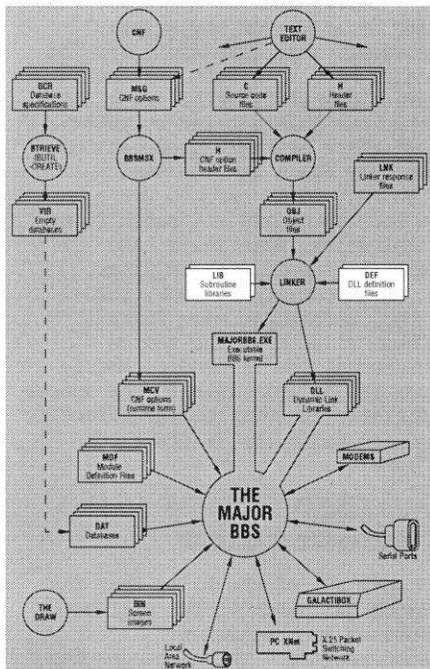
See page 162 about developing offline utilities.



Source Directory: \BBSV6\SRC



Object Directory: \BBSV6\PHOBJ



Library Directory: \BBSV6\DLIB

File Extensions

.ALT	Alternate sorting sequence for databases
.ANS	Text files with ANSI commands
.ASC	Text files without ANSI commands
.BAT	Batch files
.BCR	Btrieve database creation specifications
.BIN	Screen image files
.C	C-language source files
.CFG	Compiler and linker configuration files
.DAT	Btrieve databases
.DEF	Protected mode definition files -- used during linking to specify exported symbols, etc.
.DID	Installation specifications file on floppy disks
.DLL	Dynamic Link Libraries (linker output)
.DMD	Disabled .MDF file
.DOC	Documentation
.EXE	Executable files (linker output)
.FLG	Special purpose "flag" files
.H	C-language header files (some are generated by BBSMSX)
.HLP	Helpful text files
.IBM	Text file with ANSI commands and Extended ASCII characters
.IDX	Internal data file
.INS	Instructions
.LNK	Linker response files -- specifies all the object files that the linker needs to create an .EXE or .DLL file
.LOG	Capture of local or emulated sessions
.MAK	Make files
.MAP	Linker report output
.MCV	CNF options (runtime form)
.MDF	BBS module definition file
.MSG	CNF options (editable form)
.OBJ	Object files (compiler output)
.REF	Internal data file
.RLN	Release notes
.RPT	Reports
.SCN	Screen image files
.TXT	Text files for online viewing (sample Menu Tree file pages)
.VIR	Empty or starting-point databases
.ZIP	Files compressed and combined with PKZIP

See also page 3 on the use of Developer-ID's as file naming prefixes.

A few unique file names:

MAJORBBS.EXE	Main executable program
MJRBBS.CFG	Menu Tree generated list of DLLs, languages, MSG's, and other BBS requirements
GP.OUT	General Protection report file

To avoid conflicts between file names and directory names wherever possible:

File names should have nonblank extensions and directory names should have blank extensions.

C Source Conventions

Here are a few of Galacticcomm's in-house standards on the formatting of C source files:

- o All C source files, after the comment header, should begin by including GCOMM.H:

```
#include "gcomm.h"
```

GCOMM.H is a header file that does several things, including:

- Includes several of the standard Borland C++ header files for defining constants, data types, macros, and function prototypes:

```
stdio.h    dos.h    setjmp.h
stdlib.h   io.h    string.h
ctype.h    math.h   stdarg.h
dir.h      mem.h    time.h
```

- Includes some special-purpose Galacticcomm header files:

```
btvstf.h   Btrieve database functions
dosface.h  DOS time and date, file finding
dskutl.h   More DOS time and file functions
tfscan.h   Text File Scanning functions
lingo.h    Multilingual information
msgdrdr.h  Reading .MSG files (for offline utilities)
```

- Includes Phar Lap's header file phapi.h
- Defines prototypes for the functions in GCOMM.LIB.
- Defines constants for the second parameter of the fopen() function (see page 50).
- Makes sure that the abs(), min(), and max() macros are defined.
- Defines constants for special keystrokes, for example F1, ALT_P, and CTRLHOME (these are possible return values of getch(), see page 144).
- Defines other constants, data types, and macros.

- o Galacticcomm C Source code uses function prototypes. This means, among other things,
 - defining the return value and parameters
 - using "void" when a function doesn't return anything
 - using "void" when a function has no parameters
 - putting a prototype in a corresponding .H file if other files must use the function

NOTE: For functions with a variable argument list that are coded in C, we don't use prototypes for the argument list, just for the return value:

```
char *spr();
int tokopt();
```

With similar functions that are coded in assembly language, we go ahead and use as much prototyping as possible:

```
void prf(char *fmat,...);
void prfmsg(int msg,...);
```

There are shareware utilities like PROTOE that will help you generate prototypes for local-use functions in a .C file.

- o Routines that are needed only within the .C file where they reside should be coded STATIC, as in:

```
STATIC void
localroutine(void)
{
    doessomething();
}
```

- o Routines that need to be called from code in other C source files should be coded like normal:

```
void
globalroutine(void)
{
    doessomething();
}
```

- o Your init__xxx() routines (page 29) should be coded as EXPORT, as in:

```
void EXPORT
init__routine(void)
{
    initsomething();
}
```

Rebuilding MAJORBBS.EXE and standard .DLL files

This section is mainly for customizing The Major BBS, not so much for developing new products for it.

When developing products for The Major BBS, we assume you won't be modifying any of the code that makes up MAJORBBS.EXE -- the BBS kernel. If you do, you're severely limiting your market to BBS operators who have purchased the Developer's C Source Kit for their own customization purposes.

When customizing The Major BBS to suit your needs, you may be modifying the C source files that make up MAJORBBS.EXE or the standard .DLL files.

If you do, just run the MAKEBBS.BAT file to recompile and link:

```
CD \BBSV6\SRC
MAKEBBS
```

This compiles and links what needs to be recompiled and relinked based on what files you have changed. The MAKE program, which comes with the Borland C++ compiler, only looks at the times and dates of the files, not at the actual changes you made. For example, if you change MAJORBBS.C, that file will be recompiled and relinked to form MAJORBBS.EXE. That makes sense. However, if you merely change an itty-bitty comment in GCOMM.H, then every single file in the standard package will be recompiled and relinked. MAKE sometimes does a little more work than it has to, but it's the safest way to be sure that you're running with the results of your latest source code changes.

You can always do the compiling and linking yourself.

To process a <filename>.MSG file into .MCV and .H files:

```
CD \BBSV6
BBSMSX <filename> -OSRC
```

To compile a <filename>.C source file that contributes to MAJORBBS.EXE:

```
CD \BBSV6\SRC
CTPH <filename>
```

(It's not a good idea to do "CTPH *" because different source files need to be compiled with different CTXXX.BAT files.)

To compile a <filename>.C source file that contributes to a .DLL file:

```
CD \BBSV6\SRC
CTDLL <filename>
```

It's very important to compile using the correct batch file (CTPH or CTDLL).

To relink MAJORBBS.EXE:

```
CD \BBSV6\SRC
LTPH
```

To relink any <filename>.DLL file:

```
CD \BBSV6\SRC
LTDLL <filename>
```

For developers: you'll be able to use many of these same steps for compiling and linking your own .DLL file for your own Add-on Option.

Creating a New .DLL File

Here are most of the administrative aspects of making a .DLL. The technical and algorithm considerations will be discussed in later sections.

1. Name your initialization routine starting with "init__" (that's two underscores), such as "init__colormod()". See page 29 for details.
2. The rest of the code in your C source file(s) will probably flow indirectly from what you do in your init__xxx() routine. For example, if you register a module, the text line input entry point will need to be coded (page 29). If you register a text variable, that routine will probably be in your C source files too.
3. Create a .LNK file for your .DLL. You can use the file \BBSV6\SRC\GALP&Q.LNK as an example:

```
\run286\bc3\lib\c0phdll +
\bbsv6\phobj\galp&q
\bbsv6\galp&q
nut
phapi galimp msgimp gsblimp /Twd /s /n
\bbsv6\dlib\nodef
```

This .DLL includes the code for the Polls and Questionnaires Add-on Option. Only one .OBJ file from the \BBSV6\PHOBJ directory is named: GALP&Q.OBJ. This .LNK file specifies the NODEF.DEF file, the one you're most likely to use (the .DEF extension is implicit)

One example of a condition that would require you to design your own .DEF file is if your DLL had routines that other DLLs needed to call for some reason. In that case: you'd make a .DEF file for the DLL where the routines were *defined*; you'd use Borland's IMPLIB program to create a .LIB import library from the .DEF file; and you'd link the .LIB file in with the DLL where the routines are *referenced*.

Read about TLINK in the Borland C++ Tools and Utilities Guide if you want to know more about .LNK files (linker response files) and .DEF files (module definition files, in Borland's terminology). See page 22 for how we use MAJORBBS.DEF to make GALIMP.LIB.

Your DLL will be able to use routines in MAJORBBS.EXE by means of our import library \BBSV6\DLIB\GALIMP.LIB. Our import libraries use ordinal references to help with upward compatibility. Our intention is that your DLL files will continue to work with future versions of The Major BBS.

If you use any routines from the Borland compiler library that we don't use in MAJORBBS.EXE and export in \BBSV6\DLIB\MAJORBBS.DEF, you will get undefined symbol errors. We use an awful lot of Borland library routines in The Major BBS, but naturally we don't use every single one. There are a couple of possibilities you may want to consider. It may be possible to simply include BCH286.LIB in the list of libraries in your .LNK file, for example:

```
phapi galimp msgimp gsblimp bch286 /Twd /s /n
```

But this doesn't always work. Many of the Borland library routines have interdependencies with other library routines or internal data structures. For example, if you use a memory allocation routine, or any kind of file I/O routine, conflicts could arise with the memory or I/O routines used in MAJORBBS.EXE. A truly independent routine will not have this problem, but the only way to know this for sure is if you have the compiler library source code.

The safest approach may be to find an alternative to using the routine. Perhaps you could write your own version of the routine (giving it a different name to avoid conflicts). Or redesign the calling routine.

4. The Major BBS makes no use of floating point numbers. Many of our Add-on Options don't either, but a few do. If your DLL uses floating point math at all, make these changes to your .LNK file:

- A. Add a second line that links in FPDMY.OBJ:

```
\run286\bc3\lib\fpdmy      +
```

- B. Add a reference to the MATHH.LIB file just after that of PHAPI.LIB on the next-to-the-last line:

```
phapi mathh galimp msgimp gsblimp /Twd /s /n
```

- C. Change the last line to refer to \BBSV6\DLIB\MATHDEF.DEF instead of \BBSV6\DLIB\NODEF.DEF. Compare the contents of those two files to see what the critical differences are.

Your software uses floating point math if: you use "%f" in any control string (for sprintf(), spr(), remember that prf(), prfmsg() and printf() don't support "%f"); if you declare any floating point variables or constants; or if you use any floating point math operators or functions.

5. Make a .MAK make file with instructions for compiling and linking your .DLL file (and .MSG file indexing with BBSMSX, as required). Basically, you're telling MAKE about all the dependencies between files. See the .MAK files that we use (\BBSV6\SRC*.MAK) as a starting point, and see Borland's documentation on the MAKE utility.
6. You may want to add your .MAK file to the end of \BBSV6\SRC\MAKEBBS.BAT for convenient, centralized recompiling:

```
cd \bbsv6\src      or      cd \bbsv6\ddd
make -fdddxyz.mak  make -fdddxyz.mak
```

where "dddxyz.mak" represents the name of your make file. (See step 13 on page 6 about whether to use \BBSV6\SRC or \BBSV6\DDD for your source directory.)

7. To try out your individual make file, just type the following, where <program name>.MAK is the name of your MAKE file:

```
MKU <program name>
```

8. Name the .DLL file in the "DLLs:" line of your .MDF file (more later on page 23).

Rebuilding Your .DLL File

If you have made your own <program name>.MAK file, you can run that whenever you need to incorporate changes to your source code.

```
MKU <program name>
```

If you've called out your .MAK file in MAKEBBS.BAT, then run that.

```
MAKEBBS
```

On the other hand, here's how to do it a piece at a time:

To process a <filename>.MSG file into .MCV and .H files:

```
CD \BBSV6  
BBSMSX <filename> -OSRC
```

To compile a <filename>.C source file that contributes to your .DLL file:

```
CD \BBSV6\SRC          or      CD \BBSV6\DDD   (as appropriate)  
CTDLL <filename>      CTDLL <filename>
```

(It's not a good idea to do "CTDLL *" because different source files need to be compiled with different CTXXX.BAT files.)

To relink your <filename>.DLL file:

```
CD \BBSV6\SRC          or      CD \BBSV6\DDD   (as appropriate)  
LTDLL <filename>      LTDLL <filename>
```

Versions

The development environment for The Major BBS depends on a number of software products, most importantly:

- o Borland C++ Compiler
- o Phar Lap 286|DOS-Extender SDK

These products are available from many sources, including Galacticom.

These software packages are updated regularly (once or twice a year) and this causes a number of problems for Galacticom, developers, and customizers of The Major BBS. We try to keep up with the latest versions, but what's available in the stores and mail-order houses, and what's compatible with what, isn't always within our control.

If you purchase these products on your own, you need to make sure that the versions are compatible with each other and with Galacticom source code. When possible, call us and verify version compatibility *before* you buy these products. We also stock them to give you a minimum-hassle alternative.

Updating Software

When updating software from Galacticom or other parties, be sure that your new combination of software is a compatible set.

Updating software from Galacticom

1. Make a back-up copy of your entire hard disk.
2. If you've changed source code that the new update overrides, save your code somewhere handy.
3. Insert the first floppy diskette into your A: drive.
4. Type "A:INSTALL". (You can use B: and B:INSTALL if you need to.)
5. Resolve changes in your source code with changes that the update provides, possibly by "merging" the two together. The "MATCH" utility might aid in this effort. It's available in the UTILITY Library on the Galacticom Demo System, at (305) 573-7808.
6. Run \BBSV6\SRC\MAKEBBS.BAT to recompile everything.

Updating your compiler or the Phar Lap DOS-Extender

1. Make a back-up copy of your entire hard disk.
2. Remove your compiler from your hard drive completely.
3. Remove Phar Lap from your hard drive completely.
4. Install the compiler, and Phar Lap, from scratch, following steps 2-4 on page 4.
5. Delete \BBSV6\MAJORBBS.EXE and all \BBSV6*.DLL files that you can recreate (probably all .DLL files except GALGSBL.DLL and BBSBTU.DLL).
6. Delete all object files in \BBSV6\PHOBJ*.OBJ and \BBSV6\LOBJ*.OBJ.
7. Run \BBSV6\SRC\MAKEBBS.BAT to recompile everything.

How we use MAJORBBS.DEF to make GALIMP.LIB

MAJORBBS.DEF contains a long list of symbols for variables and functions defined in MAJORBBS.EXE. It gets used in two ways to bridge the gap between the code in the DLLs and the code in MAJORBBS.EXE. (1) MAJORBBS.DEF is used when creating MAJORBBS.EXE to identify those symbols as "exported" so they are available to DLLs. (2) It's also used to make GALIMP.LIB, the import library that's linked with all DLL code, so that code can use those symbols.

To use MAJORBBS.DEF to create GALIMP.LIB, the following line must first be added to the beginning of MAJORBBS.DEF:

```
LIBRARY MAJORBBS
```

Then we use Borland's IMPLIB to create GALIMP.LIB

```
IMPLIB GALIMP.LIB MAJORBBS.DEF
```

Then we remove the LIBRARY command from MAJORBBS.DEF, restoring it to the form that's useful when linking MAJORBBS.EXE (via LTBBS.LNK)

3. OPERATING ENVIRONMENT

Module Definition Files (.MDF)

Module Definition Files are key to how The Major BBS recognizes many aspects of your Add-on Option:

- o What .DLL files to load, and what module(s) the Sysop can use in Menu Tree module pages
- o What .MSG files to include, providing CNF options
- o Module dependency on other modules
- o Module replacement of other modules
- o Special installation processing
- o Online users manual, to integrate into BBSUSER.DOC
- o Btrieve database requirements
- o Special processing at auto-cleanup, or at timed events
- o Add-on utility (option 8 from the introductory menu)
- o Language information
- o Custom editors for CNF text blocks and Menu Tree custom menus

Here is the .MDF file for the Registry of Users:

```
; GALREG.MDF
Module Name: Registry of Users
Developer: Galacticomm
Requires:
Replaces:
Install:
Online user manual: GALREG.DOC
DLLs: GALREG
MSGs: GALREGIS
Btrieve page size: 1024
Btrieve files: 1
Cleanup:
Event-1:
Event-2:
Event-3:
Event-4:
Add-On Utility:
```

Here are some details about what to put in your .MDF file.

Comment Header

This is the name of the .MDF file.

Module Name

This description of the module appears when designing module pages in the Menu Tree. It will also appear on the miscellaneous statistics screen. (Use gmdnam() to read in this description -- see page 30.) The name may be up to 24 bytes long.

Developer

Your name or company name.

Requires

If your module will require that any other modules be "active", name them here. You might use this if that module exports symbols that you use. When several modules all require each other, use a circular definition. (For example GALFOR.MDF requires GALTLC.MDF which in turn requires GALEML.MDF which itself requires GALFOR.MDF again.) You can list up to five .MDF files.

Replaces

If you have a module that replaces another, name the other module here. For example, Galacticcomm's Entertainment Collection has an Entertainment Teleconference that replaces the teleconference that comes with the standard version of The Major BBS. (Note: the "replaces" and "requires" logic work well together, such that if module A requires module B, and module C replaces module B, then module A's requirement is satisfied by module C being present.)

NOTE: be sure not to "replace" a module that has exported symbols, even if your module exports the same symbols.

Install

If any special installation procedures are required, call out the .EXE, .COM or .BAT file here (just the root of the file name, not the extension, like you'd type in a DOS command). This file will be run only when the module is first installed on a Sysop's computer, and then automatically deleted.

The presence of this file acts as a "flag" to find out whether installation is needed or not.

Online user manual

This should name a text file you write to orient new users to the online services your module provides. This text file will be automatically combined with others to form BBSUSER.DOC. That file is (by default) attached to the welcoming Electronic Mail message sent to all new users. It is also available online in the Information Center. The convention is to use the same name as the .MDF file with a .DOC extension.

DLLs

Name the Dynamic Link Library (or Libraries, up to five of them) for your module. See page 19.

MSGs

Name the .MSG files that contain CNF options (for the Sysop to manage in Hardware Setup, Security and Accounting, Configuration options, and Text Block Editing). More about CNF options starting on page 55.

Btrieve page size

If your Add-on Option uses any Btrieve databases of its own, you must name the maximum page size in bytes. This is also specified in the .BCR files for creating the databases. See the Btrieve documentation for more on page sizes.

Btrieve files

This is the number of .DAT Btrieve database files that your Add-on Option will have open at a time. (We keep all .DAT files open for the entire time the BBS is up.)

Cleanup

Name an .EXE or .BAT file to run when the BBS shuts down for auto-cleanup. You can have multiple "Cleanup:" lines with multiple DOS commands.

Event-1 through Event-4

Name an .EXE or .BAT file to run when the BBS shuts down for any of the timed events.

Add-On Utility

If you want an offline utility to appear in the menu from option 8 of the introductory menu, specify the .EXE, .COM or .BAT file and description of the option. For example:

Add-On Utility: DDDANLYZ (analyze color reciprocity)

The file DDDANLYZ.BAT (or DDDANLYZ.EXE, etc.) will be run if the operator picks that option.

For aesthetics, we recommend that you fit the name within an 8-character left-justified field, capitalize the entire name, follow it with one space, and append a lower-case description in parenthesis. The description can be up to 40 characters long.

UNCONDITIONAL

If this word appears on a line by itself, anywhere in the .MDF file, then your module is loaded unconditionally (whether something in the operator's Menu Tree calls for it or not).

INTERNAL

If this word appears on a line by itself anywhere in the .MDF file, then your module won't appear in the list of choices for module pages. If your module is an INTERNAL module, you'll almost certainly want to make it also an UNCONDITIONAL module.

Language .MDF Files

Now, here is an example of an .MDF file for the "German/RIP" Alternate Language Add-on Option:

```
; GERM RIP.MDF (language file created by BBSLANG.EXE)

Module Name: German/RIP language

Developer: Sysop

Internal
Unconditional

Language: German/RIP
Language Description: Die deutsche Version von RIPscrip graphics
Language File Extension: .RIP
Language Editor: RIPaint.DLL %s
Language Yes/No: JA/NEIN
```

This has a few additional constructs in it.

Language

The language name has a 1-8 character spoken language followed by a slash (/) and a 1-6 character terminal protocol. That's a total of up to 15 characters.

Language Description

This description can be up to 40 characters long. It shows up when picking a language out of a list, as in the CNF <F3> CHOOSE LANG softkey, or online when users pick a language.

Language File Extension

A file extension may be needed to store text for this language on disk. This comes up when customizing menus under Menu Tree or editing CNF options with certain custom editors.

For custom menus under the "/ANSI" languages, three file extensions apply:

- .IBM ANSI colors and cursor control with IBM extended ASCII
- .ANS ANSI colors and cursor control with standard ASCII
- .ASC Standard ASCII

When it comes time to use one of these files, here's the decision process that occurs in opnans() in MENUING.C:

```
If the user's terminal has ANSI capability
  If the user's terminal is an IBM PC
    Use the .IBM file (or .ANS or .ASC as required)
  otherwise
    Use the .ANS file (or .IBM or .ASC as required)
otherwise
  Use the .ASC file (or .ANS or .IBM as required)
```

For other editors, you'll probably want to specify a single file extension, such as .RIP for RIPaint.

When it comes time to custom design the way a menu looks, **Menu Tree** will present a list of file names with all of the extensions for all user-languages that are defined, as in:

```
Edit INFOMENU.IBM using BBSDRAW
Edit INFOMENU.ANS using BBSDRAW
Edit INFOMENU.ASC using BBSDRAW
Edit INFOMENU.RIP using RIPAINIT
Edit INFOMENU.ZAP using ZAPDRAW
```

For CNF options, a temporary file is created for external editors when they specify a DOS command line. The Language File Extension will be used on that file.

Language Editor

There are two cases where an editor is needed: CNF text blocks and Menu Tree customized menus. BBSDRAW is the standard in-memory editor and is specified like this:

```
Language Editor: BBSDRAW %s
```

Let's say you want to use an editor named "ZOGEDIT" that expects the file name as it's parameter and needs the "/A" switch for ANSI capabilities, you'd specify:

```
Language Editor: ZOGEDIT %s /A
```

Then when it came time to edit an option or a menu, the %s would be replaced with a file name, and the above command executed as a DOS command. We use a system() call on this command line, so DOS will look for ZOGEDIT.BAT, ZOGEDIT.COM, or ZOGEDIT.EXE throughout the path.

The first word of the language editor command line should be the name of the editor. An optional extension could have special meanings:

<name>	to specify a DOS command line
<name>.EXE	to specify a DOS .EXE file that can be "spawned" as a daughter process (the daughter EXE file must have a very large capacity for file handles -- this is not usually the best approach)
<name>.DLL	to specify an editor in a DLL file (see page 169 about custom editors in DLLs that register themselves)

In all cases the <name> part is advertised as the name of the editor, both in CNF and in Menu Tree.

Any "%s" in the language editor command line gets replaced with the name of the file to be edited.

Language Yes/No

This line of the language .MDF file tells you what words to use for "yes" and "no" in the language. For example:

Language Yes/No: Affirmative/Negative

In this case the letters 'A' or 'N' will be expected in response to a yes-or-no question. Clearly the first letter of each word must be different.

There may be conflicts when certain prompts expect Y=yes, N=no or other special characters. First, you should avoid words that start with either '?' or 'X', as those characters have universal meaning on the BBS. Also, a yes or no word that started with 'R' would conflict with the highly visible exit-logoff-relog situation where users can choose (Y)es=logoff, (N)o=stay online, or (R)elog on.

The Language Yes/No directive also affects the operation of cncyesno(). If a user enters an 'A', cncyesno() will return 'Y'. That way, your code can always check cncyesno() for 'Y' or 'N' return values. See page 76.

Users can also type in the entire yes or no string. cncyesno() will take it all, and still return 'Y' or 'N'.

You can see that BBSMAI.MDF has the standard MDF information combined with language information for the "English/ANSI" language.

Initialization Routine (init_XXX())

The Major BBS will recognize any routine in your .DLL whose name starts with "init_" (that's two underscores) as an initialization routine, and call that routine when the BBS comes up. In your initialization routine, be sure to:

- o Declare it as EXPORT (page 17).
- o Register your module with register_module() (page 30).
- o Open the CNF options file (.MCV in it's runtime form) using opnmsg() (page 65). You can read in options using routines like numopt() and ynopt().
- o Open any Btrieve databases you're using with opnbvtv() (page 150).
- o Declare how much if any of the Volatile Data Area that you'll need to service users, using dclvda() (page 46).
- o Allocate memory if you need it. (Try to use the Volatile Data Area if at all possible.)
- o Register global commands with globalcmd() (page 96).
- o Prepare enough VDA memory for any Full Screen Data Entry sessions using fsdroom() (page 103) and dclvda().
- o Register text variables with register_textvar() (page 73).
- o Kick off any rtkick() routines (page 177).

Modules

A module is the main mechanism for making your code run on The Major BBS while it's online. There are other ways to run online code, like text variables or global commands, but modules are by far the most powerful.

By registering a module, you complete one of the links necessary to make your service available to users online. The other link is completed by the Sysop, when he designs his Menu Tree. Remember that the Sysop ultimately decides who uses each service and how the menus leading to it are structured.

Here are the items to remember when making a module for The Major BBS:

- o Put your initialization code in a routine whose name starts with "init_" (see above).
- o In that initialization code, register your module using the register_module() function and a unique module structure.
- o Compile and link your module code into a .DLL file (page 19).
- o Identify the .DLL file in your .MDF file (page 23).

```

statecode=register_module(ptrmodule);   Register a module
int statecode;                          Value of usrptr->state
                                         whenever user will be "in"
                                         this module

struct module *ptrmodule;                Pointer to your module
                                         structure

```

You might use the statecode in some circumstance to determine if the user is "in" your module (by comparing `usrptr->state == statecode`, for example). Each time you call `register_module()` you must pass a pointer to a unique module structure.

Here is the data structure for each module, consisting of a description and nine entry points (this comes from MAJORBBS.H):

```

extern
struct module {
    char descrp[MNMSIZ];                /* module interface block          */
    int (*lounrou)();                   /* description for main menu        */
    int (*sttrou)();                   /* user logon supplemental routine */
    void (*stsrrou)();                 /* input routine if selected       */
    int (*injrou)();                  /* status-input routine if selected */
    int (*lofrou)();                  /* "injoth" routine for this module */
    void (*huprou)();                 /* user logoff supplemental routine */
    void (*mcurrou)();                /* hangup (lost carrier) routine   */
    void (*dlarou)();                 /* midnight cleanup routine        */
    void (*finrou)();                 /* delete-account routine          */
    void (*finrou)();                 /* finish-up (sys shutdown) routine */
} **module;

```

Here are some sample pieces of code for initializing a simple module:

```

int colorinp(void);
void clscol(void);

struct module colormod=(
    "", /* module interface block          */
    NULL, /* name used to refer to this module */
    colorinp, /* user logon supplemental routine */
    dfsth, /* input routine if selected       */
    NULL, /* status-input routine if selected */
    NULL, /* "injoth" routine for this module */
    NULL, /* user logoff supplemental routine */
    NULL, /* hangup (lost carrier) routine   */
    NULL, /* midnight cleanup routine        */
    NULL, /* delete-account routine          */
    clscol /* finish-up (sys shutdown) routine */
);
static
int colstt; /* ANSI color diagnostics, state no. */

void EXPORT
init_colormod() /* the module initialization routine */
(
    stzcpy(colormod.descrp, gmdnam("DDDCOLOR.MDF"), MNMSIZ);
    colstt=register_module(&colormod);
)

```

As this example shows, you should leave the description blank (a zero-length string) and read it in from your .MDF file during initialization. This example reads in the module description from the DDDCOLOR.MDF file, such as:

Module Name: ANSI color diagnostics

This way, your module name is only in one spot: the .MDF file. (If the description in the .descrp field disagrees with the one in the .MDF file, then the module won't be accessible: Menu Tree will call out one name, but the other will be registered online.)

Any of the entry points can be NULL if you don't need them except: `ststrou()` (make it "dfsth" if you don't need special status handling, like in the above example); and `sttrou()` (make it point to a function that returns zero if your module has no menu-selectable interactive services). The above example sets up to call the `colorinp()` for text input and `clscol()` when the BBS shuts down.

Variables available to many of the entry points

The following global variables are set up by the executive before calling any of the entry points `lonrou()`, `sttrou()`, `ststrou()`, `lofrou()`, or `huprou()`:

```
int usrnum;           the user number for the communications channel.

struct user *usrptr;  points to that channel's "user" struct (see
                     MAJORBBS.H). For example, usrptr->baud is his
                     baud rate, usrptr->substt is his substate, etc.

struct extusr *extptr; points to extendable in-memory data for the
                     channel. (For upward compatibility, new in-memory
                     fields will be added here, and not to the user[]
                     array.)

struct usracc *usaptr; points to that channel's "usracc" struct (see
                     page 158). For example, usaptr->userid is his
                     User-ID, usaptr->usrpho is his phone number.
                     This information is stored in a database.
```

Each user's session on the BBS is a procession through a series of states. The states distinguish conditions like "waiting for him to type in his User-ID", "waiting for him to type in his password", or "waiting for him to decide whether to thread forward or backward from this Forum message".

The `usrptr` structure contains this state information, represented by three variables (see MAJORBBS.H). These represent the "context" of the user:

```
usrptr->class      Channel condition:

                  VACANT = 0   channel is not in use
                  ONLINE = 1  call has been answered (getting User-ID
                               and password)
                  BBSPRV = 2  acts similar to ACTUSR, except credit
                               deduction and time limits don't apply
                               (a service might take advantage of this)
                  SUPIPG = 3  sign-up in progress (brand new user)
                  SUPLON = 4  supplemental log-on activity in progress
                               (using the lonrou() entry point)
                  SUPLOF = 5  supplemental log-off activity in progress
                               (using the lofrou() entry point)
                  ACTUSR = 6  logged on (using the sttrou() entry point
```

(Don't confuse `usrptr->class` with the user classes that Sysops define from the Remote Sysop ACCOUNT menu CLASS command -- the concepts are not related.)

```
usrptr->state      Module number in use, corresponding to the return
                  value of register_module(). (This makes sense only
                  when usrptr->class is one of the last three above
                  values.)
```

`usrptr->substt` Sub-state number within the module selected, if any. This is usually some number indicating the question last asked of the user. It's zero at the beginning of a series of calls to `sttrou()` (when entering a module page) or `lonrou()` (when logging on) or `lofrou()` (when logging off). If those entry points return 1, then they should also set `usrptr->substt` to a nonzero value, so that they can recognize the context when they are called again with a line of input from the user. More about this on page 33.

The following global variables are available for any entry point where user input is expected. This includes `sttrou()` for lines of text when the user is "in" the module, and `lonrou()` and `lofrou()` when you've set up some supplemental interaction during log-on or log-off (more on that below).

<code>int margc;</code>	the number of separate arguments (words) in the user's input line (see page 74)
<code>char *margv[];</code>	table of pointers to the "words" of the user's input line
<code>char *margn[];</code>	table of pointers to the ends of the input words
<code>int inplen;</code>	total length of the input line in bytes
<code>int pfnlvl;</code>	profanity level of the input (0 to 3, mild to severe)

The Volatile Data Area is maintained during any protracted interactive session, as with `sttrou()`, `lonrou()`, and `lofrou()`. So you can use the VDA for storing information to track that session. You can also use the VDA in your `huprou()` entry point, with restrictions (see page 47).

<code>char *vdaptr;</code>	points to the Volatile Data Area. This is memory allocated for a channel, and used by the routines of a module -- but used only while the module is selected for that channel.
----------------------------	--

lonrou() - log on input service routine

You can use this entry point to give a user some kind of notice when he is logging on, such as "Your stock rose 4 points last night". You can also use `lonrou()` for a protracted interactive session with the user -- a series of questions and answers, or prompts and commands that he goes through right after logging on, such as "would you like to purchase more shares now?", "Ok, how many?", etc. This all happens before the user has a chance to make a selection from the top menu.

The difference (one log-on message versus a series of log-on prompts and responses) lies in the `lonrou()` return value.

<code>lonrou()</code> returns 0	you're done with logging this user on
<code>lonrou()</code> returns 1	you're expecting the user to respond to a prompt that you just sent to him

To send one log-on message, just make `lonrou()` always return 0. To go through a series of prompts and responses, return 1 whenever you're expecting more input from the user. Then `lonrou()` will be called again when he types the next line of input.

When your lonrou() returns 0, the user will resume his log-on process. (Perhaps there are other modules with lonrou()'s of their own. Otherwise he's ready to get the top menu.)

You can tell the first lonrou() from subsequent lonrou() calls by the user's substate:

```
usrptr->substt == 0 on the first call to lonrou(). Your lonrou() routine should change it to something nonzero, and return 1.
```

```
usrptr->substt == nonzero on subsequent calls. Keep changing this substate between lonrou() calls. When done, return 0.
```

While you're lonrou() routine is "in effect" (until it returns 0), any status codes on the channel trigger a call to your ststrou() entry point.

Note: to intercept the moment when a user *first* connects to the BBS, use the handle-connect vector (*hdlcon()). See page 84.

For special handling of new users who have just signed up, you could maintain your own separate database of User-ID-tagged information and, during lonrou(), look for the case when usaptr->userid isn't in your database yet. (And then insert it of course.)

sttrou() - module input service routine

sttrou() is the most heavily used entry point for most modules. It is first called when a user selects your module (selects a module page that refers to your module from a parent menu). After that, sttrou() is called each time a user enters a CR-terminated input line while "in" the module. He gets "out" of the module (returning to the parent menu) when the sttrou() routine returns 0.

You can keep track of the user's context with the usrptr->substt variable. This is always zero when the user first enters a module. You can set it to a different value for each prompt you send him, so that when you get his reply, you can interpret it in the proper context.

TIP: We use the CNF option number codes for double duty: identifying the prompt text block, and remembering the context (substate) of a reply.

Identifying the prompt text block: As you'll see on page 65, text blocks are identified (1) by their .MCV file (specified by setmbk()), and (2) by an integer sequence number defined in the .H file. That integer can be used in calls to prfmsg() to output the text block, for example:

```
setmbk(colmb);
prfmsg(PROMPT);
```

(colmb is returned by opnmsg(). PROMPT is from a BBSMSX-generated .H file.)

Remembering the context (substate) of a reply: We often use the integer sequence number defined in the .H file to remember the last prompt sent to the user:

```
usrptr->substt=PROMPT;
```

That way when he replies to the prompt, his reply can be handled in the proper context:

```
switch (usrptr->substt) {
:
:
case PROMPT:
    handleit(margc,margv[0]);
    break;
:
:
}
```

We recommend that whenever the user enters the single letter 'X', that he exits out of whatever he's doing and returns to a previous menu. If your module has it's own local menu (many do), then he returns to that menu, unless he's already there, in which case he returns to the parent menu page. We also recommend that whenever the user just hits <CR> all by itself (so that margc == 0), that you re-transmit the last prompt. See page 36 about handling asynchronous messages (such as user-to-user paging, Sysop-to-user messages, etc.) in the injrou() entry point or by checking the INJOIP flag.

Entering and Exiting a Module

Special things happen when a user first enters a module. The input string that's parsed into words in margv[] (see page 74) is a combination of three things:

- o The select character that got the user into this module (that's the single-character menu selection that the user typed, as defined in Menu Tree Design for the parent menu page)
- o The command string for the corresponding module page, if any (the Sysop specified this in offline Menu Tree Design)
- o What the user typed after the select character, if anything

Also the substate (usrptr->substt) always starts at zero. The state (usrptr->state) is the value returned to you by register_module() when you registered your module.

The sttrou() entry point for your module keeps getting called with each new input line until your sttrou() routine returns a zero. That's your way to signify (to the BBS executive) that the user is exiting your module back to the parent menu.

See page 78 for more about whether to exit to a module's local menu or to the parent menu page.

stsrou() - status handler routine

The stsrou() entry point is invoked when the user is "in" the module (has selected a module page that's based on your module from some menu, or is in the module's log-on or log-off supplemental entry point), and the channel detects a status condition. Here are some of those status conditions (for more details, see the GSBL documentation):

Some values of the global variable "status"

- 5 Output to user has completed -- this status is intercepted by the executive when it results from an injoth() (an asynchronous message). In some cases however, an injoth() or other system operation will let a status 5 slip through to your stsrou() entry point. If you use status 5's yourself, turn them on with btuoec(usrnum,1) and then send your output. When you get the status 5, be sure to check context (such as a flag or substate code) before processing it (spurious status 5's should be ignored or passed to dfsth()). After processing, turn off status 5's with btuoec(usrnum,0). If you don't turn them off, then you'll get a status 5 after every prompt from then on.

- 2,12,22 The GSBL routine btucmd() has been passed a command and that command has completed normally

- 240 Used by convention in The Major BBS for "cycle-mediated" tasks. This value is represented by the constant CYCLE in MAJORBBS.H. You can generate status 240's artificially with btuinj(usrnum,CYCLE). See page 48.

- 251 Data input overflow (usually harmless)
- 252 Echo buffer overflow
- 253 Data output overflow
- 254 Status input overflow (rather serious)
- 255 Command output overflow (rare)

Call dfsth() (the default status handler) for status conditions your module is not specifically expecting.

injrou() - reprompting routine

There are many cases when the BBS needs to immediately send a user a brief message. An asynchronous message is one that interrupts the user's normal banter of prompts and commands. After the message is displayed, the user needs to see his current prompt over again.

The injoth() function (see page 72) is used to send asynchronous messages to a user's terminal, such as "SYSTEM GOING DOWN", "FRED SMITH IS PAGING YOU", or "YOUR FAX WAS SUCESSFULLY SENT". There are two ways your module could handle this.

Asynchronous message handling method 1: If the injrou field of your module structure is NULL, the executive just simulates a <CR> from the user's terminal to get the user's prompt back. If you need to use the condition of an empty input line for some purpose other than for reprompting (such as for default answers) then you can detect the use of injoth() with a test similar to this (this could be an excerpt of your sttrou() routine):

```
if ((usrptr->flags&INJOIP) == 0) {
    handlecommand();
}
else {
    reprompt();
}
```

Asynchronous message handling method 2: the injrou() entry point, if one exists in your module, is called when an asynchronous message needs to get through to the user. The message is already formatted in the prdbuf buffer and can be transmitted with btuxmn(). You should also resend the latest prompt. For example, here's an excerpt of a possible injrou() routine:

```
btuxmn(othusn,prdbuf);
btuoec(othusn,1);
user[othusn].flags|=INJOIP;
return(1);
```

In fact, this is exactly the code in dftinj() that get's executed when there is no injrou() entry point (except that dftinj() has no return value -- don't use it as your injrou() routine). There's little point to having an injrou() routine that's exactly the above, but you could make little modifications to it. The btuxmn() is used in place of an outprf(othusn), specifically so that a user <Ctrl-O> abort of text output doesn't clobber the message. You may have some other way of displaying the message (e.g. transmitting an ANSI sequence to pop up a window or something). Setting btuoec(othusn,1), and setting the INJOIP flag prepares for a future call to the sttrou() entry point with the INJOIP flag set, asking for a reprompt.

Your injrou() entry point must not modify the prdbuf buffer contents. Those contents may be needed for output to other channels.

So to reprompt after displaying the asynchronous message, use btuoec() and INJOIP, as shown above.

Note the use of othusn instead of usrnun. Whatever instigated this asynchronous message, it probably didn't happen due to a process on the recipient's channel. It may be in response to a "/p" global page command from another users channel, or to a send-message softkey command from the Sysop's console. So the familiar usrptr is not defined at this point.

```
int othusn;           User number of the channel that is to receive
                      the asynchronous message.
```

Remember that in your `injrou()` routine:

- o Don't use `usrnum`, use `othusn`.
- o Don't use `usrptr->substt`, use `user[othusn].substt`.
- o Don't use `usaptr->userid`, use `uacoff(othusn)->userid`.
- o Don't use `extptr->lingo`, use `extoff(othusn)->lingo`.

Your `injrou()` routine needs to return a value indicating whether the user got the message or not:

- `return 0` if the user could not be interrupted at the moment
- `return 1` if the message was sent to the user's terminal

This, quite logically is also the return value of `injoth()` (see page 72).

`lofrou()` - log off input service routine

You can use this entry point to give a user some kind of notice when he logs off, such as "Thank you for your purchase order of 12 items". You can also use `lofrou()` for a protracted interactive session with the user -- a series of questions and answers, or prompts and commands that he goes through just before logging off, such as "Would you like your order shipped to you within six hours by Lazer Express for an extra \$29?", "Then we'll need your complete 9-digit ZIP+4 code:", etc. This all happens before the final "Are you sure you want to log off (Y/N)?" prompt.

The difference (one log-off message versus a series of log-off prompts and responses) lies in the `lofrou()` return value.

- `lofrou()` returns 0 you're done with this user, he can log off
- `lofrou()` returns 1 you're expecting the user to respond to a prompt that you just sent to him
- `lofrou()` returns -1 user does not want to log off after all, return him to his most recent menu

To send one log-off message, just make `lofrou()` always return 0. To go through a series of prompts and responses, return 1 whenever you're expecting more input from the user. Then `lofrou()` will be called again when he types the next line of input.

When your `lofrou()` returns 0, the user will resume his log-off process. (Perhaps there are other modules with `lofrou()`'s of their own. Otherwise he's ready to confirm that he really wants to get disconnected.)

You can tell the first `lofrou()` from subsequent `lofrou()` calls by the user's substate:

- `usrptr->substt == 0` on the first call to `lofrou()`. (Change `usrptr->substt` to something else, and return 1.)
- `usrptr->substt == nonzero` on subsequent calls to `lofrou()`, until `lofrou()` returns 0.

huprou() - user disconnect routine

Every module's huprou() entry point is invoked whenever a fully-logged-on user loses carrier. You can use this to de-allocate any resources you might have allocated for the user (be careful with the Volatile Data Area, see page 47). If NULL, no action is taken.

mcurou() - auto-cleanup routine

The mcurou() entry point of each module is invoked once per day, (default time: 3:00 AM). EMAIL.C uses this opportunity to scan the Electronic Mail database for stale messages (over 3 weeks old by default). This entry point may be NULL if the module has no need for auto cleanup processing. You can also use the "Auto Utility:" line of the .MDF file to specify offline processing during the auto-cleanup. See page 25.

dlarou() - delete user account routine

This entry point is called for all modules when a user account is deleted. A pointer to the User-ID of the account to be deleted is passed as an explicit parameter to this routine.

The dlarou() entry point exists so that if any special, module-specific actions are necessary when an account is deleted, they will get done. For example, the Registry module maintains a separate database, keyed by User-ID, containing all of the information the user had entered in response to the Registry questionnaire. When a user's account on the BBS is deleted, the corresponding Registry entry should be deleted from the Registry database too, so as not to waste disk space and create account-confusion problems. Therefore, the Registry's dlarou() entry point routine deletes the record for the user, if it exists, from the Registry database.

finrou() - system shutdown routine

Finally, the finrou() entry point is invoked as the system is shutting down and returning to DOS -- this is the place to flush buffers, close files, and so on. In general, you will be undoing whatever was fired up by the corresponding init_xxx() routine that registered your module. This entry point will be called when a "catastro" fatal error occurs (see page 51), in an attempt to save whatever can be saved before returning to DOS. This routine should be designed to safely execute in this kind of hostile situation. For example, say your init_xxx() had some code that did this:

```
if ((localfp=fopen("SOMEFILE.TXT",FOPWA)) == NULL) {
    catastro("Cannot create SOMEFILE.TXT!");
}
```

Then your `finrou()` entry point could do this:

```
if (localfp != NULL) {
    fclose(localfp);
    localfp=NULL;
}
```

That way, the `fclose()` is guaranteed to execute no more than once, and then only if the file had been opened in the first place.

Channel Numbering and Grouping

The Major BBS can handle up to 256 communication channels simultaneously. Channels are numbered in hexadecimal from 0 to FF.

Channel numbers are assigned to actual hardware devices in Hardware Setup, with the `STARTx` options (starting channel number for each group 1-16). See the System Operations Manual for The Major BBS. Channel numbers need not be assigned sequentially. The Sysop can have a GalactiBox on channels 10 to 1F, a PC Xnet card on channels E0 to FF, and a COM3 modem on channel 03.

Channel numbers control where an online user is indicated on the Summary and Online User Information screens. Channel numbers are also recorded in most audit trail messages.

Channel numbers are used to identify channels for the Sysop.

The total quantity of channels that are defined adds up to an important value, called "nterms". Many data structures in The Major BBS are multiplied by this value.

```
int nterms;          total number of channels defined
```

Channel 00 is reserved for local Sysop emulation, and counts as one of the defined channels that add up to `nterms`.

User Numbers

Independent of channel numbering, there is an internal index called a user number. User numbers are assigned sequentially to each channel that is defined. So user numbers always run from 0 to nterms-1. The global variable "usrnum" is set to the user number of the user currently being serviced.

User numbers are used internally to identify each channel.

`usrnum` has the following additional values for special occasions:

User number -1	Channel FFFF (hex)	Operator Console operations
User number -2	Channel FFFE (hex)	Auto-Cleanup operations
User number -3	Channel FFFD (hex)	Timed shutdown event

User numbers are used to index the arrays that are dimensioned by the value nterms. Almost all of the low-level hardware interface routines in the Galacticomm Software Breakthrough Library deal directly with a specific communications channel, and this channel is specified by this user number, not by the channel number that the Sysop assigns.

If you want to change the value of the usrnum variable, even temporarily, it's a good idea to use curusr() to do it. See page 73.

The array channel[] in MAJORBBS.C translates from user number to channel number:

```
channel[<user number>] == <channel number>
```

The function usridx(), also in MAJORBBS.C, does the reverse translation (or returns -1 for unassigned channel numbers):

```
usridx(<channel number>) == <user number>
```

User number nterms-1 is channel number 00, and is reserved for local Sysop emulation.

Group Numbers

The Major BBS can have up to 16 channel groups, nominally and internally numbered 1 to 16. (Use the constant NGROUPS for the number of groups).

```
grpnum[<user number>] == <group number>
```

You can use this to determine the channel type of a group using the grtype[] array:

```
grtype[<group number>] == <group type code>
```

Group Type Codes (from MAJORBBS.H)

```
#define GTMODEM 1 /* group type code: Modem channels */
#define GTMLOCK 2 /* group type code: Locked modem channels */
#define GTSERIAL 3 /* group type code: Serial channels */
#define GTX25 4 /* group type code: X.25 channels */
#define GTLAN 5 /* group type code: LAN channels */
#define GTNONE 0 /* group type code: No channels defined */
```

Here is an example of testing a channel's type:

```
if (grtype[grpnum[usrnum]] == GTLAN) {
    ... LAN channel type ...
}
else if (usrptr->flags&ISX25) {
    ... X.25 channel type ...
}
else {
    ... other channel type ...
}
```

Note that there are two ways to check for an X.25 channel. The flag check (when done by itself) takes less code.

Data Structures and Memory Allocation

The Major BBS has numerous ways of handling data, based upon how long the data must last (its lifetime) and upon how often, or in what manner, the data must be accessed.

DATA STRUCTURES AVAILABLE TO ALL MODULES

<u>Structure</u>	<u>Lifetime</u>	<u>Access</u>	<u>See page</u>
CNF options	Unlimited	Disk (direct)	55
Online User status and session information (usrptr)	User session	Memory	81
Online user detail (usaptr)	Unlimited	Memory	158
Offline user detail (BBSUSR.DAT)	Unlimited	Disk (indexed)	158
System variables (sv, sv2)	Unlimited	Memory	157

DATA STRUCTURES FOR USE BY A SPECIFIC MODULE

<u>Structure</u>	<u>Lifetime</u>	<u>Access</u>	<u>See page</u>
Volatile Data Area	Module active	Memory	46
alcmem() memory	BBS session	Memory	42
File opened using fopen()	Unlimited	Disk (sequential)	50
File opened using opnbtv()	Unlimited	Disk (indexed)	149

EXAMPLES OF DATA USED BY A SPECIFIC MODULE

<u>Structure</u>	<u>Lifetime</u>	<u>Access</u>
Electronic Mail message	21 days (default)	Disk (indexed)
Uploaded attachment to an E-mail message	14 days (default)	Disk (sequential)
Online user's Forum quick-scan configuration	Unlimited	Memory
User log-off record	BBS session	Memory

<u>Lifetime</u>	<u>Explanation</u>
o Unlimited	For the life of your hard disk (or until someone explicitly changes the information)
o BBS Session	While the BBS is "on-the-air"
o User Session	While a user is online
o Module active	Between the time a user selects a module from the Menu Tree, and exits that module

<u>Access</u>	<u>Explanation</u>
o Memory	Access is fastest
o Disk (direct)	Access requires reading only a sector or two
o Disk (sequential)	Access as a serial stream of bytes
o Disk (indexed)	Access records by their content (using Btrieve)

You must take careful consideration of the scope of a variable before you use it. For example, some mistakes to watch out for:

- o Referencing BBSUSR.DAT for account detail on a user who is online. If your application must deal with the accounting detail of a user, be sure to use uacoff() (page 82), for online users, or the "BBSUSR.DAT" database record for offline users (these have the same structure, see page 158). You can see how the module addcrd() in ACCOUNT.C makes this distinction when it adds credits to a user's account.
- o Unconditional use of the Volatile Data Area in the huprou() entry point. If your module requests temporary use of the Volatile Data Area, your module can freely use the area in its sttrou() (line input) and stsrou() (status) entry points. But your module should not use the Volatile Data Area in the huprou() entry point unless the usrptr->state code is equal to the module's handle (return value of register_module()), i.e. unless the user hung up when he was "in" your module. See page 46 for more details.
- o If you are developing a global command handler (page 96), be careful not to use the vdaptr memory area. This might conflict with the use of vdaptr by whatever module the user is working in, such as Electronic Mail. Use the vdatmp buffer only for one-shot ad hoc purposes (see page 47).
- o Heeding all of these cautions, use the Volatile Data Area whenever you can, rather than alcmem()'ing your own memory region. This will keep the BBS's use of memory from getting out of hand.

```

region=alcmem(nbytes);           Dynamically allocate some memory
char *region;                   pointer to the region
unsigned nbytes;                size of the region, in bytes
                                (up to 65530)

```

This routine differs from the standard C malloc() allocation function, in that alcmem() NEVER returns the value NULL. Any memory allocation errors are handled by the shut-down routine memcata() (see page 52).

Since memory allocation is relatively time consuming, we recommend that you avoid using alcmem() for short-term solutions. The Volatile Data Area is better for data that is only needed while a user is in a specific module (see below).

Memory allocated using alcmem() is automatically deallocated when The Major BBS shuts down and returns to DOS. To deallocate yourself, you may use the standard Borland library routine free().

To move an already-allocated block of memory into bigger (or smaller) living quarters:

```
newspace=alcrsz(oldspace,oldsize,newsize);           Reallocate space to a different size
char *newspace;                                     new space
char *oldspace;                                     old space
unsigned oldsize;                                    old size
unsigned newsize;                                    new size
```

The new space will have the same contents as the old space, up to the size of the smaller space. If `oldspace` is `NULL`, or `oldsize` is zero, then `alcrsz()` will act exactly like `alcmem(newsize)`. Otherwise, the `oldspace` parameter should be the return value of an earlier call to `alcmem()` (or an equivalent routine, such as `alcrsz()` itself). Like `alcmem()`, `alcrsz()` will never return `NULL` (it will catastro() if it runs into trouble).

To allocate new space for an existing NUL-terminated string:

```
newspace=alcdup(string);                             Allocate new space for a string
char *newspace;                                     new space
char *string;                                       old space
```

You might do this if the old space is volatile and about to be used for some other purpose. The `alcdup()` routine is similar to the Borland library function `strdup()`, except that `alcdup()` will never return `NULL`.

```
zregion=alczer(nbytes);                              Allocate new memory and zero it out
char *zregion;                                       address of new memory
unsigned nbytes;                                     size in bytes
```

This routine is just like `alcmem()` except that the new memory is filled with zero bytes.

If you're allocating an array of blocks, one per channel (that is, "nterms" of them), then you should only use `alcmem()` or `alczer()` if each block is smaller than 256 bytes. If the block is 256 bytes or larger, you should use `alcblok()` or `alctile()`.

To allocate more than 64K worth of memory at a time, you'll need a way to break it down into smaller parts. There are two schemes for doing this that differ in how the memory is allocated, but are almost identical functionally. `alctile()` gives you a different selector for each of these smaller parts, and `alcblok()` crams as many parts into each selector as possible.

If you're allocating an `nterms` array (an array of structures, one per online user), and that structure is 256 bytes or larger, then the array could be $256 \times 256 = 65,536$ bytes or larger, and you need to use one of these schemes. The prime recommended method for allocating a region that is $N \times M$ bytes long is to use `alcblok()`:

```
bigregion=alcblok(qty,sizblock);                     Allocate a very large memory
                                                    region, qty by sizblock bytes
void *bigregion;                                     return value, for ptrblok() only
unsigned qty;                                        number of "blocks"
unsigned sizblock;                                   size of each "block"
```

The `alcblok()` routine never returns `NULL` (it calls `catastro()` in case of insufficient available memory). If `qty x sizblock` is less than about 64K, `alcblok()` only allocates one region. Otherwise it will allocate multiple regions of up to 64K each.

When you want to use one of the individual blocks, you have to pass the "bigregion" return value to `ptrblok()`:

<code>block=ptrblok(bigregion,unum);</code>	Dereference an <code>alcblok()</code> 'd region
<code>void *block;</code>	pointer to an individual block
<code>void *bigregion;</code>	return value from original <code>alcblok()</code>
<code>unsigned unum;</code>	index, 0 to <code>qty-1</code>

The `alcblok()` return value can only be passed to `ptrblok()` and not dereferenced in any other way. You should store it in a variable declared to be type "void *". The `ptrblok()` return value on the other hand can be assigned or cast to the native type of your blocks (whatever it is you're allocating that has "sizblock" bytes). Typically you would cast it to a variable of type (struct *something* *).

This is roughly how we allocate memory in `MAJORBBS.C` for the Volatile Data Area, and in `FILEXFER.C` for the file transfer session control blocks.

The "unum" parameter is an index between 0 and `qty-1` ("qty" was passed to the original `alcblok()`). `ptrblok()` returns a pointer to the block of memory `sizblock` bytes long corresponding to this index. Each value from 0 to `qty-1` will give you a different block. You shouldn't count on any other aspect of the `ptrblok()` return values. For example, different values of `unum` might or might not produce pointers with different selectors.

In general you can count on `ptrblok()` never returning `NULL`. The only exception might be if you abusively call it with a `NULL` `bigregion`, or an out-of-range `unum`.

The other method for large memory allocation is `alctile()`. There is a corresponding `ptrtile()` routine, which is the only legal way for dereferencing `alctile()`'s return value, just like the `alcblok()/ptrblok()` cousins. In fact the calling parameters are identical too:

<code>bigregion=alctile(qty,sizblock);</code>	Allocate a very large memory
<code>void *bigregion;</code>	region, <code>qty</code> by <code>sizblock</code> bytes
<code>unsigned qty;</code>	return value, for <code>ptrtile()</code> only
<code>unsigned sizblock;</code>	number of "tiles"
	size of each "tile"
<code>block=ptrtile(bigregion,unum);</code>	Dereference an <code>alctile()</code> region
<code>void *block;</code>	pointer to a tile (offset always 0)
<code>void *bigregion;</code>	return value from original <code>alctile()</code>
<code>unsigned unum;</code>	index, 0 to <code>qty-1</code>

What's happening under Phar Lap is that the region is "tiled" into a series of regions that are each smaller than 64K. Each region gets a different selector, and its base offset is guaranteed to be zero.

You might need this special feature of `alctile()/ptrtile()`, but it is usually much better to use `alcblok()/ptrblok()` if at all possible, because of the latter routine's economy with selectors. Every computer has a rock-solid limit of 8192 selectors, no matter how much memory it has. (That limit is imposed by the number of possible 16-bit values with the three low order bits set to 111.) So selector economy is a very desirable thing.

By the way, there is no way to free the memory allocated by `alcblok()` or `alctile()` before the program terminates (at which time the memory is automatically freed, of course). It's assumed that you'll be keeping these very large regions of memory in use for the duration of the program.

We use the following routines for general purpose handling of memory regions:

<code>movmem(source,destination,nbytes)</code>	Move a block of memory
<code>char *source;</code>	source block
<code>char *destination;</code>	where to put it
<code>unsigned nbytes;</code>	number of bytes, 1 to 65535
<code>setmem(destination,nbytes,value)</code>	Set a block of memory to a value
<code>char *destination;</code>	pointer to the block
<code>unsigned nbytes;</code>	number of bytes, 1 to 65535
<code>char value;</code>	1-byte value or character
<code>repmem(destination,pattern,nbyt)</code>	Replicate a pattern in memory
<code>void *destination;</code>	where to put it
<code>char *pattern;</code>	NUL-terminated string
<code>int nbyt;</code>	total number of bytes at dest

The `repmem()` function will replicate $\lceil \text{nbyt}/\text{strlen}(\text{pattern}) \rceil$ copies of the pattern at the destination. (The `'\0'` terminator of pattern is not replicated.) If that quotient is not an integer, the last copy of the pattern will be truncated, but exactly `nbyt` bytes will be written. No NUL is ever written to destination.

<code>chimove(source,destination,nbytes)</code>	Reentrant version of <code>movmem()</code>
<code>char *source;</code>	source block
<code>char *destination;</code>	where to put it
<code>unsigned nbytes;</code>	number of bytes, 1 to 65535

The `chimove()` function can be called by interrupt routines as well as mainline routines without conflict.

<code>memavl=sizmem();</code>	Find out how much memory is available
<code>long memavl;</code>	number of bytes

Volatile Data Area

```
dclvda(nbytes);           Declare size of the Volatile Data Area
int nbytes;              size in bytes
```

This function should only be called by your `init_XXX()` routines (see page 29). The function declares the maximum size that the module will require of the Volatile Data Area. Each user online will be given a separate region of this size. When the user selects a module page from a menu option, the corresponding module may use that region until the user exits back to the parent menu again. For example, if the Electronic Mail module requires 1000 bytes and the Registry module requires 500 bytes of the Volatile Data Area, they should both declare these amounts in their `init_XXX()` routines. 1000 bytes will be allocated (the larger of the two).

```
char *vdaptr;           Points to the Volatile Data Area
```

This global variable points to a memory region that is allocated for each user who is online, and is used by the module that is in effect at the time. The variable `vdaptr` is set to point to the appropriate region upon each call to these entry points for the module:

```
sttrou()  character line input after user selects the module page
lonrou()  log-on message / line input during log-on
lofrou()  log-off message / line input during log-off
stsrrou() status input
```

Continuing the above example for `dclvda()`, each time that a user in E-mail types in a line, the `sttrou()` entry point for E-mail is invoked (see page 33) and the global variable `vdaptr` points to that user's Volatile Data Area. The E-mail software is free to store whatever it likes there for the duration of the user's stay in E-mail.

```
int vdasiz;            The actual size of the Volatile
                        Data Area.
```

Of course, `vdasiz` is only valid when all the voting is done -- that is, at any point other than your `init_XXX()` routine.

The entry point:

```
huprou() hang up
```

may also use the Volatile Data Area under the condition that:

```
usrptr->state == <module number>
```

where <module number> is the return value of register module() (page 30) of the module whose huprou() entry point has been called. In other words, huprou() may work with the Volatile Data Area if the user hung up while that module was active (while the corresponding menu option was selected). Remember that whenever a user logs off or hangs up, The Major BBS calls the huprou() entry point of every module, not just that of the module he was using. So if your huprou() entry point detects that the user who is logging off was inside of your module (using the above test), then huprou() may take appropriate steps to clean up any unfinished business in the Volatile Data Area. Otherwise, it must leave the VDA alone.

```
char *vdatmp;                               Points to the ad hoc Volatile Data Area
```

This additional area is available after the initialization entry points have been called for all modules and the Volatile Data Areas (page 46) have been allocated for each channel. "vdatmp" is to be used for brief ad hoc purposes. You can't depend on the contents of the buffer it points to being preserved through any cycle of the BBS. You can only use vdatmp within a single call to any of the other entry points, or within a single rtkick() invocation, or within other routines for short-term purposes (but not within your init_XXX() routine).

For one example of vdatmp usage, see the implementation of the global "/r" registry lookup command in REGISTRY.C (function gloreg()).

```
char *vdaoff(unum);                          Compute volatile data pointer for  
int unum;                                    some other user
```

This routine is used whenever you need to access the volatile data area of some user other than the one that you are directly servicing (the one referred to by the global variable "usrnum"). You might use this to check before a user deletes an item, to make sure that no other users are using it at the same time. Just remember that any module other than your module can make any use of the VDA that it pleases. You'll probably only want to use vdoff() on users who are also in your module.

Ways to Split up a Long Task

Since The Major BBS is a multi-user system, it cannot work on any one task for too long at a time. If it did, then some users would experience an annoying delay in the response time of the BBS. (By the way, this delay would not show up between character transmissions to the user through the modems -- those are interrupt driven. Echoes of user keystrokes are also interrupt driven. Rather, this kind of delay might show up in the time between a user typing in a line and receiving his next prompt.) A certain amount of delay cannot be avoided, particularly with disk I/O.

If you have a time-consuming task to perform, and if you can break that task down into chewable computation bites, then you can improve BBS response time in two ways:

Cycle Mediating	Simulating CYCLE status codes
Polling Routine	begin_polling() and stop_polling()

Cycle Mediating

The trick is this: perform a little bit of the task and then generate a status 240 condition. Channel status conditions are managed internally by the GSBL (Galacticomm Software Breakthrough Library). Then when that status 240 is reported back to you, do a little more work on the task, generate another status condition, and so on. This allows The Major BBS to service all other channels that are online, plus perform other housekeeping chores, while it's also working for the user in your module. (You will see in your GSBL manual that the status code we use for this "cycle mediating" purpose, status 240, is reserved for application program use.) In the source file MAJORBBS.H, the constant "CYCLE" is defined as 240. You can see how this scheme is used in the status handler for Electronic Mail and for Forums -- function emsth() in ESGUTL.C.

As a simple example, suppose a module, when selected by a user, simply displayed four lines on the user's screen and then, after the user hit the return key, returned to the parent menu, as follows:

```
line 1
line 2
line 3
line 4
```

```
Hit RETURN (wait until user hits return key)
back to menu...
```

(In practice, you would never split up such a small task.)

The task is split up into four sub-tasks using the "cycle-mediated" method. The following functions would be used for the sttrou() and stsrout() entry points:

```
STATIC int
stttxm(void)
{
    switch (usrptr->substt) {
        case 0:
            prf("line 1\n");
            outprf(usrnum);
            btuinj(usrnum,CYCLE);
            usrptr->substt=1;
            return(1);
        case 4:
            prf("back to menu...\n");
            outprf(usrnum);
            return(0);
    }
    return(1);
}

STATIC void
stsexm(void)
{
    if (status == CYCLE) {
        switch (usrptr->substt) {
            case 1:
                prf("line 2\n");
                usrptr->substt=2;
                btuinj(usrnum,CYCLE);
                break;
            case 2:
                prf("line 3\n");
                usrptr->substt=3;
                btuinj(usrnum,CYCLE);
                break;
            case 3:
                prf("line 4\n\nHit RETURN ");
                usrptr->substt=4;
                break;
            default:
                dfsth();
                return;
        }
        outprf(usrnum);
    }
    else {
        dfsth();
    }
}
}
```

Notes:

- o `usrptr->substt` is used to keep track of the progress of each user that selects this module from a menu. It is always set to zero when a user first selects the module.
- o `prf()` is like `printf()`, except that the converted text goes into a buffer. `outprf()` transmits the contents of that buffer to a specific user.
- o `stsexm()` calls `dfsth()` (the default status handler, in `MAJORBBS.C`) when `stsexm()` encounters a status code that it is not expressly designed to deal with.

Polling Routine

The other way to break a long task down into parts is by registering a polling routine. Each channel can have a polling routine that is called regularly. The actual polling rate depends on system loading, but it can be very rapid.

<code>begin_polling(unum,rouptr);</code>	Turn on polling for this channel
<code>int unum;</code>	User number for the channel
<code>void (*rouptr)(void);</code>	Polling routine (no parameters, no return value)
<code>stop_polling(unum);</code>	Turn off polling for this channel
<code>int unum;</code>	User number for the channel

To start, register the polling routine with `begin_polling()`. The `stop_polling()` function is often called by the polling routine itself, when it decides polling is over.

File Handles (fopen())

The Major BBS supports up to 256 users simultaneously. So it needs to have numerous files open simultaneously. Unfortunately, DOS "EXE" programs, and most compilers support only 20 file handles. To get around that limitation, we've included code in the `PHGCOMM.LIB` library that increases the file handling capacity. It's important that linker response files list `PHGCOMM.LIB` before the patched Borland library `BCH286.LIB` (as is done in `LTBBS.LNK`) for this to work.

This allows up to 254 total files to be open simultaneously, using either the standard `"fopen()"` or `"open()"` routines (we use `fopen()`).

The Major BBS as shipped from the factory was compiled and linked with these modified routines installed, so the `MAJORBBS.EXE` file can handle more file handles. If you're developing your own Add-on Option, your `.DLL` code will use the `fopen()` that's in `MAJORBBS.EXE`.

The Second Parameter of fopen()

Use the following constants for the second parameter of fopen(), depending on how you will use the file:

FOPRA	Read in ASCII mode
FOPRB	Read in Binary mode
FOPWA	Write in ASCII mode
FOPWB	Write in Binary mode
FOPAA	Append in ASCII mode
FOPAB	Append in Binary mode

Exception Handling (catastro())

catastro(ctlstg,p1,p2,...,pn)	"catastrophic" error, exit to DOS
char *ctlstg;	control string for error message
TYPE p1,p2,...,pn;	parameters for error message (maximum 8 bytes of parameters)

This module is called under numerous failure mode conditions. You should remember that catastro() failure conditions are severe cases, such as missing databases, DOS errors, or illegal formatting in the CNF options (see page 52 about insufficient memory errors).

So, when to use catastro()? Most often, it's to give a bumbling Sysop a soft place to fall. There are many cases when not to. If it's a likely Sysop mistake, then the Sysop procedures need reworking. If it's a programming mistake, then you may need more safeguards. If it's the result of something bizarre that a non-Sysop user has done, you absolutely *must* keep the system up and not penalize innocent bystander users.

Typical catastro() events are things that should "never happen" under normal conditions. But when Murphy's Law prevails and they do happen, it should be orderly. It's good to use a catastro() when the alternative would be a chaotic hard-to-trace result that nobody in their right mind would want.

Say you're using two databases and you just know that if you pull a certain name from the A database, that the same name will appear in the B database one or more times. Well, the Sysop could trip you up by failing to properly restore both database files from a backup in tandem. The result should not be that the program destroys both databases.

When choosing the wording of your catastro() message, try to keep in mind honest mistakes the Sysop could make and use plain english to lead him toward a solution. A Sysop is more likely to be able to handle "Cannot find file XXXX.ZOO" than "fopen() is NULL on XXXX.ZOO". Here's the point: word Sysop-causable errors for Sysops and word errors that only a programming error could cause for programmers.

Often, rather than calling catastro(), you can just allow something unpleasant but isolated happen, like the user who triggered the unhappy event could get an empty list with no explanation (but not trash -- sending trash to the screen might have unpredictable consequences). For example, you should apply extra caution when processing strings that they don't overflow the destination buffer -- use stzcpy(), or brutally chop off the source string if you have to.

Now, to make every function that deals with a pointer check for NULL is pretty silly, so a balance is needed. For example, whenever you use fopen() to open a file, always check for NULL, so that if a Sysop messes up his installation or runs out of disk space he get's something predictable and not a wild memory write followed by a computer lock-up. Nothing's worse than an intermittent lock-up.

A non-Sysop user should never be able to trigger a catastro() -- your customers' BBSes would vulnerable to hackers.

Here is an example of how you would use catastro() to handle the case of a missing file:

```
if ((fp=fopen("NEEDTHIS.FIL",FOPRA)) == NULL) {
    catastro("Cannot find the file \"NEEDTHIS.FIL!\");
}
```

Note that the function for opening Btrieve databases, opnbvtv() (page 150), has a built-in catastro() to handle the file-not-found condition (BTRIEVE OPEN ERROR 12).

The parameters are identical to those of the standard printf(), but no more than 16 bytes of parameters (that's not including the control string) can be passed. For example, each of the following would exhaust the parameter list p1,p2,...,pn, but they would work:

- 4 pointers to character strings
- 8 integers
- 8 characters (remember, a character parameter takes up 2 bytes)

Note: no long integer or floating point values can be used as parameters (i.e. your control string cannot contain "%ld" or "%f" directives). If you need to make such conversions, see about the l2as() and spr() functions starting around page 175.

All catastro() messages are written to the text file CATASTRO.TXT with a time and date stamp, assuming, of course, the system is still capable of writing to disk rationally.

Insufficient Memory (memcata())

All errors that result from a quantitative lack of memory should not call catastro(), they should call memcata():

```
memcata();                                Generate a catastro() with a polite
                                           message about insufficient memory:
                                           "There is not enough memory to
                                           continue. Please either reduce
                                           your memory requirements or install
                                           more memory, and try again."
```

(The routines alcmem() and alczer() have their own internal calls to memcata(). They never return NULL.)

Languages

The Major BBS can support multiple spoken languages ("English", "French", "German"), multiple dialects ("Expert", "Tutorial"), and multiple terminal protocols ("ANSI", "RIP") for multiple users simultaneously. Language names consist of a 1-8 character spoken language, a slash, and a 1-6 character terminal protocol. That's a total of up to 15 characters. Some examples:

English/ANSI	Spanish/RIP	Expert/ANSI
English/RIP	German/ANSI	Staff/ANSI
Spanish/ANSI	German/RIP	Tutorial/RIP

The multilingual feature primarily allows different versions of user output to be defined for different languages. The BBS won't translate user input (e.g. menu selections and commands). For example, if a user has to type 'R' for read or 'W' for write, he'll have to do the same thing in all languages. The best way to handle this is in the way the prompts are worded, for example:

```
RDOWRT {(R)ead or (W)rite? },{ (L)eer or (E)scribir?}      ..is wrong..
RDOWRT {(R)ead or (W)rite? },{R=Leer, W=Escribir?}        ..is right..
```

One exception: YES and NO responses can be translated. Different languages can mean that the BBS expects different strings for "yes" and "no". This affects the operation of the `cnctyesno()` routine, and some other special cases. You can use `lingyn()` for those special cases: it translates a user's single-character response into 'Y' or 'N' depending on their language (see page 77).

When the BBS comes up it builds a list of the user-languages that are defined on the BBS and sets a few global variables:

<code>nlingo</code>	number of languages defined, always at least 1
<code>clingo</code>	language index, 0 to <code>nlingo-1</code> , for the current user
<code>extptr->lingo</code>	usually the same as <code>clingo</code>
<code>extoff(n)->lingo</code>	language index of user number <code>n</code> (where <code>n</code> is 0 to <code>nterms-1</code>)
<code>languages[clingo]->name</code>	name of the current user's language
<code>languages[clingo]->desc</code>	description of the current user's language

See `LINGO.H` for more fields in the `languages[]` array of language information structures.

The main function of `clingo` occurs when reading in the type "T" (text block) CNF options from disk. There can be a different version of each type "T" option for each language, and the value of `clingo` determines which version to read in. We'll get into this more on page 65.

To look up the index of a language by its name:

```
ilingo=lngfnd(lngnam);   look up a language by it's name
char *lngnam;           name of language, 1 to 15 characters long
int ilingo;             language index, 0 to nlingo-1, or -1=unknown
```

To show users a list of all languages for them to pick:

```
prf("\rWhich language/protocol would you prefer to use on this BBS?");
lnglist(1);
lngfoot(1);
```

You'd be better to use prfmsg() (page 65) than prf() of course, but using prf() is a better way to show you what's going on in this example. The "1" parameter to lnglist() and lngfoot() means offer all languages as options for the user to pick. Use a "0" instead to only offer those languages with the top voting confidence factors (more about that on page 88). Here's how you might put the user's choice into effect:

```
int ilingo;
:
:
if ((ilingo=cnclng()) != -1) {
    clingo=extptr->lingo=ilingo;
}
```

Either a number or a language name will satisfy cnclng(). After this, all future prfmsg() output on this channel will be in the new language.

Maximum Number of Languages

We claim that The Major BBS can support up to 50 simultaneous languages, but the practical limit is probably higher. The tightest constraint comes from the needs of a certain structure in each .MCV file. You can compute that limit like this:

```
language limit = 32767 / number of options in the .MSG file
```

There's actually a different language limit for each individual .MSG file. For example, BBSMAJOR.MSG has about 300 options in it, so it should be able to support over 100 languages. That means that each text block in BBSMAJOR.MSG could have 100 different versions. But if one Sysop's BBSMAJOR.MSG had 100 languages, then problems could occur if a future release of BBSMAJOR.MSG had more than 327 options. Hence the official limit of 50 languages.

If a Sysop exceeds the limit on the number of languages, then BBSMSX would report:

```
Too many options (starting at "XXXXXX")
or too many languages in XXXXXXXX.MSG.
```

Creating CNF Options

CNF options affect many aspects of the operation of The Major BBS. See the System Operations Manual. CNF options are stored in .MSG files, converted to .MCV files, and read in, as needed, using a very quick direct indexed scheme. CNF options help in these ways:

- o The Sysop who isn't a programmer can change numerous values, names, options, prompts, and messages that affect the operation of the BBS.
- o A large volume of text is stored on disk, saving memory.

As a developer, you can specify your own CNF options in .MSG files. These are converted into a special form for use by The Major BBS at runtime -- the .MCV files. These sections will help you create new CNF options, and use them in The Major BBS.

The Major BBS Configuration Facility, CNF, requires special formatting information about each CNF option in the .MSG files. When the system operator uses CNF to change the value of a CNF option, then this information is used to make his job easier.

CNF type "text" options can be specified in different languages. The first line of an .MSG file defines the languages that may appear throughout the file, in this format:

```
LANGUAGE {<language 0>},{<language 1>},{<language 2>} ...
```

For example:

```
LANGUAGE {English/ANSI},{Spanish/ANSI},{French/ANSI}
```

Language 0 is always English/ANSI. Omitting the LANGUAGE{} pseudo-option is equivalent to including the line:

```
LANGUAGE {English/ANSI}
```

CNF options can be specified at different levels:

```
LEVEL1  Hardware Setup options
LEVEL3  Security and Accounting options
LEVEL4  Configuration options
LEVEL6  Editable Text Blocks
```

The levels are numbered to correspond with the numeric selections from the introductory menu.

Other special-purpose levels:

```
LEVEL8   Full Screen Editor help messages

LEVEL30  \
LEVEL31  \ Reserved for configuring the 16 databases of
:        / The Major Database
LEVEL45  /

LEVEL96  Reserved for configuration options of the Major Gateway/
         Internet Add-on Option

LEVEL97  Reserved for options in the Entertainment Teleconference that
         cannot be edited by CNF, including "action" specifications

LEVEL98  Reserved for text that Sysops are not expected to want to view
         or modify using CNF

LEVEL99  Reserved for Full Screen Data Entry templates (which are not
         editable by CNF)
```

The .MSG files have the following format:

```
LANGUAGE {<language 0>},{<language 1>},{<language 2>} ...

LEVEL1 {}
<option specifier>
<option specifier>
:
LEVEL3 {}
<option specifier>
<option specifier>
:
LEVEL4 {}
<option specifier>
<option specifier>
:
LEVEL6 {}
<option specifier>
<option specifier>
:
```

Each section at any level may contain from zero up to any number of option specifiers. The "LEVELn {}" may be omitted for any section that contains no option specifiers.

Each <option specifier> has the following format:

```
<help paragraph> <option name> <version list> <hinge> <coding>
```

```
<help paragraph>  Up to 12 lines of text describing the CNF option.
                  This message appears on the CNF screen when the
                  operator is in HELP mode. You should only use
                  columns 2 through 60 of these 12 lines to give the
                  paragraph the proper appearance on the CNF screen.
                  (For best appearance, if you use less than 12 lines,
                  add a blank line before the line with the option
                  name. If you use all 12 lines, use no blank line.)
```

The <help paragraph> may be omitted. In that case, leave two blank lines in its place.

<option name> One to eight characters (capital letters or numbers). This same symbol will be used in the "C" language source code to refer to this CNF option. This is done with the .H file that BBSMSX generates.

<version list> The text of the option, perhaps with versions in multiple languages (for type "T" options only). There is always a version for language 0. All other languages may or may not have versions. Of course, there can't be more versions than there are languages, as defined by the LANGUAGE{} line at the start of the file.

In a 4-language file, here are the possibilities for encoding the 4 different versions:

```
<version 0>
<version 0>,<version 1>
<version 0>,<version 1>,<version 2>
<version 0>,,<version 2>
<version 0>,<version 1>,<version 2>,<version 3>
<version 0>,,<version 2>,<version 3>
<version 0>,<version 1>,,<version 3>
<version 0>,,,<version 3>
```

Notice that empty and missing are not the same thing. An empty option has nothing between the curly braces, but a missing option has no curly braces. See about language subsets in the System Operations Manual.

Only type "T" options can have multiple versions in multiple languages. Other types of options always have exactly one version.

<version n> This is a string of characters that are available to The Major BBS at runtime. This, and the option name is the only information that BBSMSX takes from the .MSG file to create the runtime .MCV file.

In the .MSG files:

```
"}" is represented as "~}"
"~" is represented as "~~"
```

The number of characters in each version is limited by the offline Configuration option OUTBSZ, which may be set to 4096, 8192 or 16384.

<contents> The <version 0> text for options of all types except type "T" is the <contents> of the option: what's between the curly braces.

<hinge> The hinge is an optional field that implies that a particular option "hinges" on another option. This mechanism is used to avoid contradictory combinations of options from appearing on the

CNF screen. You can use it to hide one option based upon the value of a preceding option. See page 62 for more details.

<coding> This information is used by the CNF utility to control the format and limitations on the option contents.

There are examples of CNF options on page 63. Also look in the .MSG files.

Option Coding Syntax

C	Character, ' ' through '~'
B	Binary ("YES" or "NO")
E <v1> <v2> ... <vn>	Enumerated (multiple choice)
N <min> <max>	Decimal numeric (%d)
L <min> <max>	Large decimal numeric (%ld)
H <min> <max>	Hexadecimal numeric (%x)
S <length> <descript>	String of characters (%s)
T <description>	Text (up to OUTBSZ-1 characters)

Type C: Character Configuration Options

The format of the <contents> for this type of option is:

<description> <character>

Where <character> is a single character, as for a menu selection, and <description> is a short description, for example:

This is the activation code letter for calibrating
uplink #3

UPSEL3 {Select character for uplink 3: G} C

Type B: Binary Configuration Options

The format of the <contents> for this type of option is:

<description> YES
or
<description> NO

Where <description> is a short description, for example:

Answer YES to this question if you want
new users to be able to play in the
games. Answer NO to allow them to
watch, but not play.

NEWGAM {Allow new users to play games? NO} B

The YES or NO choices will show up as softkey selections under CNF.

Type E: Enumerated Configuration Options

The format of the <contents> for this type of option is:

<description> <choice>

Where <choice> is a one-word selection among a small set of possible answers. <description> is a short description. The set of possible answers is enumerated in the <coding>, for example:

How rough do you want users to be
able to play?

EASY -- nobody loses too much
NORMAL -- can lose your shirt
ROUGH -- users can cheat
BRAWL -- cheaters can be shot

PLALVL {Play difficulty: ROUGH} E EASY NORMAL ROUGH BRAWL

These four enumerated <choice>'s will show up as softkey selections under CNF.

Type N: Numeric Configuration Options

The format of the <contents> for this type of option is:

<description> <number>

Where <number> is a 16-bit integer between -32768 and 32767. A smaller set of limits may be specified in the <coding>, for example:

How many seconds should we
wait for a user's bet before
skipping him for the round?

PLWAIT {Wait for how many seconds? 30} N 5 3600

In this case, 5 and 3600 are the "permanent" inclusive limits on the value of the <number>. The operator, using CNF, can change the value of this option to something other than 30, but not to something outside of the range 5 to 3600. If you don't want any particular limits on a option, then you may specify "N -32768 32767"

Type L: Large Numeric Configuration Options

The format of the <contents> for this type of option is:

<description> <number>

just like for type N options, except that this <number> will be stored as a 32-bit integer. Limits are specified in the <coding>, for example:

How much should we allow a user
to bet during one round?

MAXBET {Maximum bet: 1000000} L 0 100000000

In this case, the value of the option is one million. The operator, using CNF, will not be able to make it larger than a hundred million. If you wish to have no particular limit on the option, you may code "L -2147483648 2147483647".

Type H: Hexadecimal Numeric Configuration Options

The format of the <contents> for this type of option is:

<description> <hexadecimal number>

The <hexadecimal number> is unsigned, and may be between 0 and FFFF. Smaller limitations may be encoded in the <coding>, for example:

What channel would you like to reserve
for your satellite uplink?

SATCHN {Channel for satellite uplink: 3F} H 0 3F

Type S: String Configuration Options

The <contents> for this type of option are the value of the string. The maximum length and description of the string are encoded in the <coding>, for example:

This string is the sign-on message for initiating
uplink using the 227.85-228.05 MHz "APLINK" band,
including your FCC registration number

UPSIGN {U905 Westar 7::88A,5932-051} S 30 Uplink sign-on command

This would appear on the CNF screen something like this:

UPSIGN Uplink sign-on command U905 Westar 7::88A,5932-051

If you use 0 as the length of the string, the maximum length will end up being used, as limited by the width of the CNF screen.

Note: A longer <description> means a shorter <contents> length.

Type T: Text Configuration Options

The <version n> text for this type of CNF option may consist of up to OUTBSZ-1 characters.

BBSDRAW, the default editor for all "/ANSI" languages, can edit an image of up to 25 lines of 79 characters each. If you use all 25 lines, then the last line cannot end with a line terminator (i.e. no more than 24 line terminators may be in the <version n>). The <coding> field specifies a short description for the option, for example:

```
UPCOMP {
Uplink established, at %s on %s

*** BEGINNING UPLINK TRANSMISSION ***
} T Uplink established notification
```

Type T options are the most numerous. Almost all user prompts and messages are among the Editable Text Blocks, and are type T options.

If Sysops change the sequence of %-symbols in a type-T option, CNF will warn them about the consequences. Even so, The Major BBS tends to be tolerant of "%s" symbols that show up where they don't belong. In case of emergency, the BBS will try to convert the %s symbols into one of these strings:

```
<null pointer>      The pointer is NULL (all 4 bytes are zero)
<invalid pointer>  The pointer does not contain a valid selector,
                   or the offset is too big for the selector
```

This may not always work, and it is possible that a misplaced %s will cause messy characters to show up on the user's terminal, or worse, the BBS could crash with a GP (general protection fault) when prfmsg() tries to use the pointer.

Hinge Specification

This feature keeps CNF from showing one option based upon the value of a preceding option. For example:

```
NEWGAM {Allow new users to play? NO} B
CHGGAM {Charge new users how much to play? 1000} N 0 32767
```

This combination of option settings does not make sense. How can you charge new users for playing if you never allow them to play? If these options were coded like this:

```
NEWGAM {Allow new users to play? NO} B
CHGGAM {Charge new users how much to play? 1000} (NEWGAM=YES) N 0 32767
```

then the second option would not even appear on the CNF screen, at least not as long as the value of the NEWGAM option was NO. Change NEWGAM to YES and CHGGAM appears.

CAUTION: The hinge feature has no effect on the contents of the .MCV file, and thus no effect on the execution of The Major BBS. Your programming on The Major BBS must specially handle a situation such as the above to be sure that new users aren't charged for a game that they aren't allowed to play, or anything similar, where BBS operation would be out of sync with the CNF option settings.

You can also use the hinge specification to test for a set of values, for example:

```
(SATLINK=KBAND,QBAND,ZBAND)
```

This hinge will activate an option when the SATLINK option is *either* KBAND, QBAND, or ZBAND. On the other hand:

```
(GEOSYNC#90,105,120)
```

will activate an option when the GEOLINK option is *neither* 90, 105, nor 120.

You probably will not want to hinge on the value of a 'T' option. Any option that does so will always be inactive.

Examples

Here's an example of an option specifier:

```
If you want users to be able to change their date of birth,
answer this question with a YES. If only you, as the
Sysop, want to have the option of changing a user's date
of birth after he or she signs up, answer this question
with a NO. You can always change the date of birth from
the User Account Detail screen
```

```
CHGBDY (Allow users to change their date of birth? NO) (ASKBDY=YES) B
```

This option is named CHGBDY. It has six lines to its help message. (These appear while CNF is in help mode, per the <F1> key). This is a B-type option, which is a YES or NO option. It's current value is NO. It is hinged on the option named ASKBDY. CHGBDY appears only if ASKBDY is set to YES.

Here's another example of an option specifier in an .MSG file:

```
GREET {Hello},{Hola},{Bonjour} T Greetings for a user
```

This GREET{} option has no help text. It has three versions for languages 0, 1 and 2. Here are more examples, with other languages missing:

	language 0	language 1	language 2
	-----	-----	-----
GREET {Hello}	Hello	Hello	Hello
GREET {Hello},{Hola}	Hello	Hola	Hello
GREET {Hello},{Hola},{Bonjour}	Hello	Hola	Bonjour
GREET {Hello},,{Bonjour}	Hello	Hello	Bonjour

The first and last examples omit the Spanish (language 1) versions of GREET{}, so Spanish language callers will revert to the English version "Hello" (which is language 0). The last two examples omit the French (language 2) version of GREET{}.

A friendly reminder:

```
GREET ,{Hola},{Bonjour}
```

```
*** Not allowed! It's never ***
*** legal to omit language 0. ***
```

Compiling CNF Options

The Major BBS makes sure it has all the .MCV files it needs to run by running BBSMSX with no arguments (this happens in BBS.BAT). That checks all .MSG files and makes .MCV files out of them if their time and date disagree. (After BBSMSX makes an .MCV file, its time and date are identical to that of the corresponding .MSG file.) That's fine, but if you insert or delete CNF options, you need a new .H file in the source directory. That should be taken care of with your .MAK file, or by specific steps in your development process (page 21).

If you're ever in doubt, here's how to run BBSMSX in a development environment:

```
CD \BBSV6
BBSMSX <filename> -OSRC
```

where <filename>.MSG is the name of your editable .MSG file. This puts the .MCV file into \BBSV6 and the .H file in \BBSV6\SRC where it can be used to compile the software that uses the options. (Of course, if you're putting your source code in a separate directory, you'll need "-ODDD" or something.)

BBSMSX has these alternative command syntaxes:

```
BBSMSX [-O<source directory prefix>]
BBSMSX @<list file> [-O<source directory prefix>]
BBSMSX <root filename> [-O<source directory prefix>]
BBSMSX <MSG file path> <MCV file path> <H file path>
```

The last syntax gives you complete control over what the files are named and where they go.

Using CNF Options

Files with .MCV extension contain the values of the CNF option for use at runtime. The Major BBS reads from these files at runtime to get the values of the options. Your source code can refer to the options by the <option name> that you used in your .MSG file, as specified on page 57. To do this, your source file will need to include the header file in your source file using the C language "#include" directive.

The symbols defined in this header file are often used for more than just referring to CNF options -- they also keep track of user substate. See page 33 for more on this.

<code>prfmsg(msgnum,p1,p2,...,pn);</code>	like <code>prf</code> , but the control string comes from an .MCV file
<code>int msgnum;</code>	message number within current .MCV file
<code>p1,p2,...,pn;</code>	just like <code>printf()</code> 's parameters (except no "longs" or "floats")

This function is just like `prf()` (page 69), in that the formatted text output goes into the `prfbuf`. However, with `prfmsg()`, the control string comes from a CNF text block. Be sure to call `setmbk()` to identify the appropriate .MCV file that the text block should come from before calling `prfmsg()` (more on `setmbk()` below). The global variable `clingo` defines the language that `prfmsg()` will read (page 53).

`prfmsg()` is used far more often than `prf()` for two reasons: (1) memory is saved by storing the text on disk, and (2) the Sysop can change the control string using CNF in offline Text Block Editing. Like `prf()`, there is no limit to the number of parameters (`p1,p2,...,pn`).

It's fine to use `prfmsg()` in cases where you're formatting text for the current user. When you're formatting text for another online user however, you need to consider what language that user has selected. Remember `clingo` is the language of the current user, and all `prfmsg()` calls depend on `clingo`. See page 70 for more about `prfmsg()`'s multilingual cousin, `prfmlt()`.

The library `PHGCOMM.LIB` (and the source file `MSGUTL.C`, which is available with the Extended C Source Suite) has several utility routines for reading and processing CNF options from these .MCV files.

<code>inimg(maxsiz)</code>	initialize the message buffer
<code>unsigned maxsiz;</code>	maximum number of bytes in any option

You'll only have to use `inimg()` if you're writing an offline utility that reads .MCV files. Set `maxsiz` to 16384 if you need to be sure you're compatible with any .MCV file used on The Major BBS.

<code>mbkptr=opnmsg(mcvfil);</code>	open a new MCV file
<code>FILE *mbkptr;</code>	MCV file identifier
<code>char *mcvfil;</code>	filespec of MCV file ("xxxx.MCV")

This routine opens a file of CNF options for reading. An array of pointers is read in at this time so that when it comes time to read the actual value of an option from disk, access time is minimal.

MCV File Identifiers

The return value of `opnmsg()` is a pointer to type `FILE`. The value identifies a specific `.MCV` file -- a file containing `CNF` options. You should only need to use this value when you call the routines `setmbk()` and `clsmg()`. The same type of value is also stored in the global variable `curmbk` (see below).

```
setmbk(mbkptr);           set "current" MCV file block ptr
FILE *mbkptr;            MCV file identifier (from opnmsg())
```

This important routine identifies the `.MCV` file to be used in subsequent calls to `getmsg()` or to `prfmsg()` (see above). When a file is opened, an implicit `setmbk()` takes place. See the above note on `.MCV` File Identifiers.

A common programming mistake is to forget to use `setmbk()` at the beginning of a series of `prfmsg()`'s. This can lead to a program that appears to work when you test it with one user, but fails with multiple users. The symptoms are usually quite obvious: messages are total nonsense, or you get a fatal error like "RAWMSG: MSG NO. <nn> OUT OF RANGE IN <filename>".

```
rstmbk();                restore previous MCV file block ptr
                          from before last setmbk() call
```

A typical usage of `rstmbk()`:

```
setmbk(fbkmb);
prfmsg(AUXBEEP);
rstmbk();
```

Calls to `setmbk()` and `rstmbk()` can be nested up to 10 levels deep.

```
extern FILE *curmbk;     get the current MCV file identifier
```

This global variable contains the current `MCV` file identifier (see above) that was last set by `opnmsg()` or `setmbk()`.

There is an alias for `curmbk`, called `lclmbk`, that allows you to get at the internal `.MCV` structure. It's the identical variable as `curmbk`, but recast to `(struct msgblk *)`. For example, `lclmbk->filnam` is the name of the `.MCV` file. See `\BBSV6\SRC\MSGUTL.H` for details. To use `lclmbk`, include `MSGUTL.H` in your C source file.

```
bufadr=getmsg(msgnum);   read value of CNF option
char *bufadr;           address of buffer with retrieved text
int msgnum;             message number (use option name from
                        the .H file)
```

This routine retrieves a `CNF` option into a buffer, and returns a pointer to the buffer. The same buffer is always used for option contents (and hence the same pointer is always returned by `getmsg()`), so you must finish using these contents before you execute another `getmsg()`, `prfmsg()`, `rawmsg()`, or `getasc()` call.

The "msgnum" parameter is the sequential number of the option within the `.MCV` file. In your source code, you can use the name of the option here. Your source file should include the appropriate header file using the `"#include"` directive. The `.H` header file was generated from the `.MSG` file by the `BBSMSX` utility.

getmsg() is called indirectly by prfmsg() (see page 65) for the most common usage of CNF options: user prompts and messages. These CNF options are type T. getmsg() does translate embedded text variables (page 73).

bufadr=getasc(msgnum);	read value of CNF option
char *bufadr;	address of buffer with retrieved text
int msgnum;	message number (use option name)

This variation on getmsg() returns text blocks with ASCII compatible line terminators (both CR and LF are on every line -- getmsg() uses an internal line terminator format where CR is a hard return, and LF is a soft return). getasc() does not interpret text variables (page 73).

bufadr=rawmsg(msgnum);	read value of CNF option
char *bufadr;	address of buffer with retrieved text
int msgnum;	message number (use option name)

This variation of getmsg() reads in the raw text from the .MCV file. Text variables, if any, are not translated, and the internal line termination scheme is used (CR = hard return or paragraph boundary, and LF = soft return).

clsmg(mbkptr);	close an MCV file
FILE *mbkptr;	file identifier (from opnmsg())

This routine closes a CNF option file and deallocates the special structures allocated by opnmsg().

The following routines are used for reading in the values of CNF options other than of type T (text). To save time, these routines are usually called during initialization, and their values are stored in memory. This means that The Major BBS need not do a disk read every time it needs the value of the CNF option.

val=numopt(msgnum,floor,ceil)	get numeric option from MCV file
int val;	value of option
int msgnum;	message number (use option name)
int floor,ceil;	Inclusive limits on the value

This function gets the value of a type N CNF option. If the value read from the file does not conform to the inclusive limits specified by "floor" and "ceil", then The Major BBS reports a "catastro" error message.

lval=lngopt(msgnum,floor,ceil)	get large numeric option from MCV
long lval;	value of option
int msgnum;	message number (use option name)
long floor,ceil;	Inclusive limits on the value

This function gets the value of a type L CNF option. If the value read from the file does not conform to the inclusive limits specified by "floor" and "ceil", then The Major BBS reports a "catastro" error message.

```

hval=hexopt(msgnum,floor,ceil)  get hex option from MCV file
unsigned hval;                  value of option
int msgnum;                     message number (use option name)
unsigned floor,ceil;            Inclusive limits on the value

```

This function gets the value of a type H CNF option. If the value read from the file does not conform to the inclusive limits specified by "floor" and "ceil", then The Major BBS reports a "catastro" error message.

```

flag=ynopt(msgnum)             get yes/no option from MCV file
int flag;                      1 if var started with "Y", 0 if not
int msgnum;                    message number (use option name)

```

This function reads in a YES or NO CNF option (type B).

```

ch=chropt(msgnum)             get single-character from MCV file
char ch;                       the character
int msgnum;                    message number (use option name)

```

This function reads in a type C CNF option.

```

string=stgopt(msgnum)         get a string from MCV file
char *string;                 pointer to newly allocated string
int msgnum;                   message number (use option name)

```

This function puts the contents of a type S CNF option into a newly allocated string that is just big enough to hold it. You could use free() if you ever needed to deallocate the string.

```

index=tokopt(msgnum,token1,token2,...,NULL)  multiple choice option
int index;                                    1=token1, 2=token2, 0=none
int msgnum;
char *token1;
char *token2;
:
```

This function checks a type E CNF option for one of several possible values. If the last word in the option specified by msgnum matches token1, then tokopt() returns 1, if token2, it returns 2, and so on. If the word matches none in the token list, tokopt() returns 0.

Don't forget to terminate the token list with a NULL parameter.

Changing Configuration Variables

If you understand the various roles of the .MSG, .MCV, and .H files you will see that changing the contents of an option without changing the order of the options has no effect on the .H file. This means that you do not need to recompile The Major BBS every time you change a CNF option. The CNF utility never changes option order. If you change the order of CNF options, either by adding, deleting, or just rearranging them, you must remember to regenerate the .H file (CNF does not do this -- use your .MAK file or the BBSMSX utility, page 64), and recompile all the source code that #include's this header file.

4. USER INTERFACE

User Output (prf(), prfmsg())

<pre>prf(ctlstg,p1,p2,...,pn); char *ctlstg; p1,p2,...,pn;</pre>	<pre>prfbuf-directed printf-lookalike printf-like control string just like printf()'s parameters (note: no "longs" or "floats")</pre>
--	---

This function has the same syntax as printf(). However, the formatted output of prf() goes into a global buffer pointed to by prfbuf. An internal variable "prfptr" keeps track of where prf() should write into prfbuf: prf() starts writing text at prfptr, terminates the text with a NUL ('\0'), and leaves prfptr pointing to the NUL when done. This means that the output of several prf()'s in sequence are concatenated together. outprf() is commonly used to transmit the results of one or more prf()'s to a specific user.

As with printf(), there is no limit to the number of parameters (p1,p2,..., pn) than you may pass to prf(). They should correspond one-for-one with the "%" directives in the control string, and you must be careful not to overflow the prfbuf. (Use PFBSIZ for the size of prfbuf, but it's not a constant. PFBSIZ is computed to be the same as the offline Configuration option OUTBSZ and is set by iniprf() at initialization time.)

See page 141 for the coding of ANSI directives that you can transmit to user screens. For example, you could use:

```
prf("\33[37;44;0mFiberlink 92 to Munich is condition \33[32;1mGREEN!");
```

This sends a message that starts out white on blue and ends up flashing bright green on blue.

See also page 65 about prfmsg(), the variation of prf() that reads text from an .MCV file. (prfmsg() is used far more often in the BBS code.)

<pre>outprf(unum); int unum;</pre>	<pre>send prfbuf to a channel & clear user number</pre>
--	---

This function transmits the contents of the prfbuf buffer to a specific user. When unum is anything other than usnum, you should probably use outmlt() (page 70). The "prfptr" variable mentioned above is reset to the beginning of prfbuf. This means that several outprf()'s can be used to transmit the same text to different users, as long as no prf()'s intervene. But the next prf() will start at the beginning of prfbuf again.

clrprf(); clear the prf buffer indep of outprf

This function resets the "prfptr" mentioned above to point to the beginning of prfbuf and stores a '\0' there.

char *prfbuf; output buffer of prf() and prfmsg()

This is the variable mentioned above that points to the buffer where user output is formatted. The contents of that buffer are transmitted by outprf() using the GSBL routine btuxmt().

char *prfptr; pointer to the current position
in prfbuf

This pointer is updated by prf() and prfmsg() to point to the end of the formatted string in prfbuf. Both clrprf() and outprf() reset prfptr to the beginning of prfbuf.

Multilingual User Output

To review, if you want to format text for the current user, you use the prfmsg() and outprf() routines, remembering to call setmbk():

```
setmbk(appmb);
prfmsg(HOWAYA,usaptr->userid);
outprf(usrnum);
```

This code prepares to read from a specific .MCV file, reads the HOWAYA message from it and formats it with the current user's User-ID, and then sends the formatted message to the current user. If there are multiple versions of the HOWAYA text block for multiple languages, then the version corresponding to the current user's language will be read (or the most appropriate alternate -- see about *language subsets* in the System Operations Manual).

Things get a little tricky when you need to send a message to another user who is also online. Let's say your module had some scheme for pairing users, and when both partners logged on you wanted to notify them. To tell the first partner that the second partner had logged on you could code this:

```
if (onsys(partner(usaptr->userid))) {
    prfmsg(PNRHERE,usaptr->userid);
    outprf(othusn);
}
```

PNRHERE says something like, "Your partner, %s, just logged on." The problem is that the other user will get the message in the language of the current user. To avoid this:

```
if (onsys(partner(usaptr->userid))) {
    prfmlt(PNRHERE,usaptr->userid);
    outmlt(othusn);
}
```

There are four routines that have multilingual "cousins":

<u>Monolingual</u>	<u>Multilingual</u>
prfmsg()	prfmlt()
prf()	pmlt()
clrprf()	clrmlt()
outprf()	outmlt()

Each routine has the same parameters as its cousin. The critical routine is `outmlt()`. It transmits formatted information to one user. That information must have been formatted by `prfmlt()` or `pmlt()`, and not `prfmsg()` or `prf()`. By the same token, `outprf()` should not be outputting information formatted by `prfmlt()` or `pmlt()`. (If you combine `prfmsg()` and `outmlt()` then English/ANSI users will get text in the native language of the `usrnum` user, and all other users will get *nothing* at all. If you combine `prfmlt()` and `outprf()`, then all users will get the English/ANSI version.)

When clearing the formatted information, it would be nice to use `clrprf()` or `clrmlt()` as appropriate, but you can always use `clrmlt()` if you're in doubt (it does everything `clrprf()` does and more). `clrmlt()` is already called before every `sttrou()`, `ststrou()`, `lonrou()`, or `lofrou()` entry point, and also before every polling routine (page 50).

The monolingual routines are more efficient than the multilingual routines, so you should always use monolingual if you know you are outputting to the current user only. In most of Galacticomm's software the vast majority of text blocks go to the current user.

Whenever output goes to a user other than `usrnum`, the most convenient thing to do is to use the multilingual suite of routines. In the above case, when `prfmlt()` formats `PNRHERE`, it first checks what languages are represented online (including that of the current user) and then for each one, formats a version of `PNRHERE` for that language. There's actually a separate `prfbuf`-type buffer (page 70) allocated for each language. Here's how to get each buffer's address and pointer:

<code>ptrtile(prfbuffers,ilingo)</code>	the address of the <code>prfbuf</code> for language <code>ilingo</code>
<code>prfpointers[ilingo]</code>	the address within the <code>ilingo</code> 'th <code>prfbuf</code> where we're currently formatting text.

The language 0 version goes in the first of the `prfbuffers`, which is `prfbuf` itself. If anyone is online with language 1 selected, then the language 1 version goes in the `prfbuffers` buffer number 1, and so forth, from 0 to `nlingo-1`. When formatting is done, then `outmlt(othusn)` sends the appropriate version of the text to user number `othusn`.

There is some work wasted here, in formatting text for languages that will never be sent to a user, but if you code multiple `outmlt()`'s, all those languages will come in handy. To save that unnecessary processing here's another way:

```
if (onsys(partner(usaptr->userid)) {
    clingo=extoff(othusn)->lingo;
    prfmsg(PNRHERE,usaptr->userid);
    outprf(othusn);
    clingo=extptr->lingo;
}
```

Here we've just changed the global variable `clingo` to the other user's language temporarily. If you're formatting text for only one other user, you can always set `clingo` like this.

But both of these methods (`prfmlt/outmlt` and changing `clingo`) have an important drawback -- they don't reprompt the other user. The other user was probably sitting at some prompt and it would be polite to show him that prompt again, after your interrupting message.

Here's the best way to send a message to another user who is online and may be using any service at all on the BBS:

```
got=injoth()          inject a message to another user
                      (implicit inputs:
                        othusn ... channel # to inject to
                        prfbuf ... message to be injected)
int got;              i=user got it  0=user was busy
```

This routine is used to transmit an asynchronous message to a user. By asynchronous, we mean a message that does not follow from the question-answer-question-answer banter that normally goes on between each user and The Major BBS. This message is an interruption. `injoth()` is used, for example, for:

- o the Teleconference "page" feature or the global `/p` command
- o the Sysop send-message function
- o notifying online users that they have received Electronic Mail
- o notifying users that credits have been posted to their account

The message will not be injected if the recipient's `NOINJO` flag (in `user[othusn]->flags`) is set, as it is when he is downloading, in Sysop-chat mode, or is otherwise unavailable. The value returned by `injoth()` indicates whether or not this happened: `1`=user got the message; `0`=user did not get the message.

Now here's what we could do to tell both partners that the other is online:

```
if (onsys(partner(usaptr->userid))) {
    prfmlt(PNRHERE,usaptr->userid);
    if (injoth()) {
        prfmsg(PNRTOO,othuap->userid);
        outprf(usrnum);
    }
}
```

`PNRTOO` says something like, "Your partner, %s, is already online." The `injoth()` routine is compatible with both monolingual and multilingual formatting methods. It does the equivalent of an `outprf()` or `outmlt()` as appropriate to the `othusn` user.

You should probably always use `prfmlt()` or `pmlt()` to format the text for `injoth()`, not `prfmsg()` or `prf()`.

In the above example, `prfmlt()` generates the text for `injoth()`, but if `prfmsg()` had been used it would inject the text in the `clingo` language only.

By the way, notice how we can get away with `prfmsg()/outprf()` to the current user after using `prfmlt()/injoth()` on the other user? This does not violate the rules of mixing monolingual and multilingual user output routines.

Changing usrnum

One last point, if you ever change the value of `usrnum`, it's important to call `curusr()`. Suppose you're temporarily changing the user number to "userno" for some reason. Do it this way:

<u>Right way</u>	<u>Wrong way</u>
<code>int unsave;</code>	<code>int unsave;</code>
<code>:</code>	<code>:</code>
<code>:</code>	<code>:</code>
<code>unsave=usrnum;</code>	<code>unsave=usrnum;</code>
<code>curusr(userno);</code>	<code>usrnum=userno;</code>
<code>:</code>	<code>:</code>
<code>curusr(unsave);</code>	<code>usrnum=unsave;</code>

The `curusr()` routine sets up many global variables in tandem with the new user number, like `usrptr`, `usaptr`, and `extptr`. It also sets `clingo` to the new user's language index.

<code>curusr(newunum);</code>	Change to a different user number
<code>int newunum;</code>	new user number, 0 to <code>nterms-1</code>

Defining Text Variables

See the System Operations Manual for The Major BBS, in the `BBSDRAW` chapter about using text variables. Here we'll tell you how to program your own text variables. From a programming standpoint, a text variable is simply a function that has a name and returns a string of arbitrary length. (The length and justification issues arise when using the variable -- see `BBSDRAW`.)

1. Code a routine that returns a pointer to a string. (You're responsible for storing the string somewhere where it will be available for immediate use. Just about any buffer except an "automatic" (stack) array will do.) Example:

```
char *
tvar_nikei(void)
{
    return(l2as(nikeiaverage()));
}
```

2. Register the routine, along with the text variable's name, using the `register_textvar()` routine, as in:

```
register_textvar("NIKEI",tvar_nikei);
```

You can do this in your `init_XXX()` initialization routine. Now you can use the text variable "NIKEI" when creating menus or text blocks.

Be careful about the context of using a text variable. Either you must code the routine so that it will produce valid results no matter when it's called, or you must be sure that when the Sysop uses the text variable in a particular text block or menu that the routine will work. See the context limitations on using some of the standard text variables in the System Operations Manual.

User Input

On The Major BBS, each time a user types a string of characters and hits <Enter>, a status 3 condition occurs on his channel. Whatever module is in effect for that channel processes the input through the sttrou() entry point for the module (see page 33).

The variables in this section are implicit inputs to the sttrou(), lonrou(), and lofrou() entry point routines for a module.

int margc;	number of words in user input line
char *margv[];	array of pointers to the words in user's input line (there are margc of these pointers)
char *margn[];	array of pointers to the ends of the words (to the terminating NUL's)

These variables are initialized by the function parsin():

parsin();	parse input line (insert '\0' after each word, compute margc and margv[])
-----------	---

The parsin() routine is always called before control is passed to your module through the sttrou() entry point. The user's input line is "parsed" into individual words, with the intervening spaces removed and '\0' terminators placed on each word. The global variable margc is the number of words, and margv[] is an array of pointers to those words. Each word contains no spaces and is terminated by NUL ('\0'). margc and margv[] work very much like the C language argc and argv[] work for command line parameters passed to the main() routine.

char input[];	user input line
int inplen;	total length of the input line in bytes
rstrin();	restore parsed input line (undo effects of parsin())

The rstrin() function restores the user's input to its original form (the NUL's are removed and the spaces restored), undoing the effects of parsin(). After calling rstrin(), you use the global variable input[] to refer to the user's entire input line.

For example, if a user types in the line "RAIN IN SPAIN" followed by <Enter>, then the sttrou() entry point of the current module is invoked with:

margc	is 3
margv[0]	points to "RAIN"
margv[1]	points to "IN"
margv[2]	points to "SPAIN"

If you call rstrin(), then:

input[]	contains "RAIN IN SPAIN"
---------	--------------------------

Profanity

int pfnlvl; "profanity level" of the input (0 means no profanity, 1 means mild, 3 means very profane)

This global variable is based on the user input line in input[]. It is saturated at (it's never more than) the value of the offline Configuration option PFCEIL.

Echo

echon(); Turn echo on for this channel

To turn echo off for a channel, use:

btuech(usrnum,0) Turn echo off for this channel

Then use echon() to turn it on again. Don't use btuech(usrnum,1) to turn echo on. To echo "secret characters", such as "****" during password entry, use this routine:

echsec(c,width); Echo secretly
char c; character to echo with every keystroke
int width; maximum number of characters expected

Then call echon() to make things normal again. The convention is to use "secchr" as the first parameter to echsec(). This is the setting of the offline Configuration option SECCHR, which defaults to "*".

Command Concatenation

This feature has two purposes on The Major BBS. (1) It allows the Sysop to define detailed subcommands within your online service. This comes up during Menu Tree design when the Sysop types in command strings for module pages that give users access your module. Look up module page design in the System Operations Manual for some examples of these strings from the standard modules of The Major BBS.

(2) Command concatenation allows an experienced user to type several commands at once. For example "ERT." from a menu that offers "E" for E-mail means: "E-mail / Read messages / To me / starting at the earliest message number".

From a programming perspective, the idea is to loop through the characters and parameters of the user's command. The global variable "nxtcmd" in CNCUTL.C keeps track of what has already been interpreted from the user's command -- it points to the rest of the command.

bgncnc(); begin command concatenation

After calling bgncnc(), the command is unparsed (has spaces again, not separate words), and prepared for interpretation using the command concatenation utilities.

```
done=endcnc();           are we done with the user's command?
int done;                1=yes, done 0=no, there's more
```

After calling endcnc(), the rest of the command is put back into input[] and re-parsed (margc and margv[] are recomputed), just as if the user had typed in the rest of the command starting from this point. If anything is left from the command, this function returns false.

```
ch=morecnc();           is there any more command?
char ch;                next character ('\0' if none)
```

The morecnc() routine tells you if there are any more characters left in the command. It first skips any leading blanks and returns the next nonblank character. The character that is returned is NOT skipped. If you want to use this character, then call cncchr().

The remaining utilities read a single parameter (character, number, etc.) from the user's command string.

```
ch=cncchr();           expect a character from the user
char ch;               the next character ('\0' if none)
                       (converted to upper case)
```

```
n=cncint();           expect an integer from the user
int n;                the integer (0 if none)
```

```
ln=cnclon();          expect a long integer from the user
long ln;              the long integer (0L if none)
```

```
n=cnchex();           expect a hexadecimal number
int n;                the number (0 if none)
```

```
ptr=cncnum();         expect a decimal number
char *ptr;            with optional '-' followed by
                       decimal digits (no conversion takes
                       place -- returns the ASCII string)
```

```
wrd=cncwrld();        expect a space-delimited word
char *wrd;            truncated if over 29 characters
```

```
uid=cncuid();         expect a User-ID or Forum name
char *uid;            the User Id or Forum name
```

```
signam=cncsig();      expect Forum name, with or without "/"
char *signam;         prefix. Always returns name with
                       the "/" prefix.
```

```
yesno=cncyesno();     expect yes or no from the user
int yesno;            'Y'=yes, 'N'=no
```

This routine translates the user's keystrokes from their selected language into 'Y' and 'N'. Suppose this line were in the French language .MDF file:

```
Language Yes/No: OUI/NON
```

Then `cncyesno()` would work like this:

<u>user inputs:</u>	<u>cncyesno() returns</u>
o	Y
oui	Y
n	N
non	N
QUE?	Q
Y	Y

The `cncyesno()` routine returns the next character from the command (and removes it from `nxtcmd`). This is also what `cnchr()` does. One difference: the translation described above. Another difference: if the user enters the entire word for yes or for no, then all of those characters are removed from `nxtcmd` too. But `cncyesno()` still only returns 'Y' or 'N' in those cases.

For cases when yes/no decisions are not made through `cncyesno()`, you could use `lingyn()`:

<code>yesno=lingyn(firstc);</code>	translate user's yes/no into 'Y'/'N'
<code>char yesno;</code>	'Y' if yes, 'N' if no, otherwise
	<code>toupper(firstc)</code>
<code>char firstc;</code>	first character of user's response,
	should be the first character of the
	yes or no words in that user's
	language.
<code>ilingo=cnclng();</code>	expect a language name or language
<code>int ilingo;</code>	pick from numbered list (1 to <code>nlingo</code>)
	returns language index, 0 to <code>nlingo-1</code> ,
	or <code>-1</code> =invalid name or number
<code>cncall();</code>	expect a variable-length word sequence
	(consume all remaining input)

Example of Command Concatenation

User session:

```
<...menu...> Q
QUIZ!
What is the first letter of the alphabet? A
How many fingers do you see? 0
END OF QUIZ! You won!
<...menu...> QA0
END OF QUIZ! You won!
<...menu...>
```


Now here's how you should use `condex()`: When your code would normally return the user to your module's internal menu (but not normally to the parent Menu Tree menu), then you can call `condex()`, to conditionally exit to the Menu Tree menu at that point. (By the way, `condex()` tests the `CONCEX` flag and does nothing if it is not set.) For example, you could code:

```
if (usrptr->flags&CONCEX) {
    prfmsg(X2MAIN);
    condex();
}
```

Now, the result (if any) of `condex()` is identical to the result of exiting from your module's `sttrou()` entry point while returning zero. The big difference is that you can call `condex()` anywhere, perhaps deep from some routine in your code, and the exit is taken immediately -- you will never "return" from `condex()`, if it takes any action at all. This feature is implemented using the `setjmp()` / `longjmp()` feature in the compiler library. See `ESGUTL.C` for a coding example.

User-ID Cross Referencing

When writing an Electronic Mail message, users can type in part of a User-ID and the BBS will present them with all User-IDs that resemble it. The user can type in a more exact User-ID, or just pick one of the alternatives by number.

To use this feature in your own program when you need the user to type in a User-ID, use the `hdluid()` routine:

```
rc=hdluid(string);          Find User-IDs that resemble a string
int rc;                    see below
char *string;
```

Return Codes

- UIDFND** User-ID found, by exact match (case is unimportant), or picked by number. You should get the User-ID from `uidxf.userid`, not from the string you passed to `hdluid()`. That string, even if it is an exact match, probably doesn't have the right case. And it could always be a number if the user ended up picking the User-ID from a list.
- UIDPMT** More than one possible match, or no matches at all. You need to reprompt a short prompt asking for a User-ID. You should be able to use the same prompt you did just before you first called `hdluid()`. Then pass the string received from the user to `hdluid()` again.
- If there were multiple possibilities, they've just been listed out. It should be obvious to the user in that case that he can just type in a number.
- UIDCAL** Continue calling `hdluid()`, no prompting is necessary. The user has just specified an incomplete User-ID, there's only one possible match, and now we're asking the user to confirm yes-or-no.

Before you pass the string to `hdluid()` (which is usually the return value of `cncall()`) you should check it for special values like 'X' for exit or '?' for help. You may be accommodating other possible entries. Then as a last resort, try `hdluid()`.

If you get the return value `UIDPMT` or `UIDCAL`, then `hdluid()` expects to get called again. If that doesn't happen for some reason (the user typed 'X' to exit and you intercepted it), then be sure to call `clrxf()`:

```
clrxf();                                Abandon User-ID cross-referencing
```

The text output of `hdluid()` is in the `prfbuf` -- your calling program must do an `outprf()` eventually.

Default Selection Character

You can allow Sysops to configure the default response to your prompts by (1) putting the default character at the end of the prompt, (2) using `getdft()` just before you output the prompt, and (3) using `chkdft()` when you get the reply.

Here's an example of a text block with the default answer at the end:

```
ASKVOW {Pick a vowel: A} T Prompt asking for a vowel
```

This is a little misleading to Sysops in that we aren't going to send the "A" when we send the prompt. You could also do this:

```
ASKVOW {Pick a vowel (hit RETURN for "A"): A} T Prompt asking for a vowel
```

Here, if Sysops wanted to change the default to "E", they would need to change two things:

```
ASKVOW {Pick a vowel (hit RETURN for "E"): E} T Prompt asking for a vowel
```

What you want your code to do is to use that final character before the "}" curly brace to fill in for a user who doesn't pick any character and just hits <Enter>. Here are the tools:

```
dftchr=getdft();                          Get the default character & remove it  
                                           from the output buffer
```

```
char dftchr;
```

```
chkdft(dftchr);                            Put the default character in the input  
                                           buffer, if user just hit <Enter>
```

You call `getdft()` after you have `prfmsg()`'d the prompt and you're about to use `outprf(usrnum)` to send it to the user's terminal. `getdft()` strips the character out of the `prfbuf` buffer (so it never gets to the user's terminal) and returns it for you to hold onto. (You can also use the "final cursor position" feature of `BBSDRAW` and `getdft()` will work properly.)

The tricky part is that you need to save this default character between cycles somehow. You get the character from `getdft()` when you send the prompt, but you need to use it when the user gets around to typing in a reply.

Then after the reply comes in, `chkdft()` checks to see if the user hit just `<Enter>` and if so, makes the input variables look as if he had typed the character. Then your code can go about its business and parse the input.

User Status and Handling

<code>ison=uinsys(usrid);</code>	determine if a user is online
<code>int ison;</code>	true if user <i>anywhere</i> online
<code>char *usrid;</code>	User-ID to be tested for
<code>int uisusn;</code>	global variable, set to user number
	when <code>uinsys()</code> returns 1
<code>ison=onsys(usrid);</code>	determine if a user is online
<code>int ison;</code>	true if user online & logged on
<code>char *usrid;</code>	User-ID to be tested for

The differences between `uinsys()` and `onsys()` are:

1. `onsys()` only returns true if the user has already logged on. `uinsys()` also catches that space of time between typing in User-ID and password when we think the user is about to log on.
2. `uinsys()` sets the global variable `uisusn`. `onsys()` sets `othusn`, `othusp`, and `othuap` (see below).

<code>isin=instat(usrid,qstate);</code>	see if a user is using a specific module
<code>int isin;</code>	true if user is in the module
<code>char *usrid;</code>	User-ID to be tested for
<code>int qstate;</code>	state (module number returned by <code>register_module()</code>)

If either `instat()` or `onsys()` return true, then the following global variables are also set:

<code>int othusn;</code>	the user number of the other user
<code>struct user *othusp;</code>	pointer to structure for that user in the <code>user[]</code> array (see <code>MAJORBBS.H</code>)
<code>struct extusr *othexp;</code>	pointer to extendable in-memory structure for that user (see <code>extoff()</code> on page 82)
<code>struct usracc *othuap;</code>	pointer to structure for that user in the "usracc" structure (see <code>uacoff()</code> on page 82)

These variables are analogous to `usrnum`, `usrptr`, and `usaptr`, see page 33. To get the other users language index, use `extoff(othusn)->lingo`.

All of the above routines will return false, by the way, for a user with Sysop privileges when he has selected `"/invis"` to become invisible. If you

need to penetrate the Sysop invisibility veil for some reason you could use the following routines instead:

```
use  onbbs(usrid,1)  instead of  uinsys(usrid)  (anywhere online)
use  onsysn(usrid,1) instead of  onsys(usrid)   (logged on)
```

This might be necessary if you were trying to decide whether to modify a user's account record in memory or on disk for example. There are several examples of this in ACCOUNT.C and ACCSCN.C.

To reference the user account information of someone who is online, don't use the `usracc[]` array directly. Since that array might be larger than 64K, you must use `uacoff()`:

```
uaptr=uacoff(unum);           Get online user account info
struct usracc *uaptr;        pointer to in-memory acct info
int unum;                    user number
```

And similarly with the extended in-memory array, use `extoff()`:

```
exptr=extoff(unum);          Get more online user info
struct extusr *exptr;        pointer to extendable in-memory info
int unum;                    user number
```

```
int ripdfd;                  1=at least one /RIP language is
                             defined, or 0=none
```

```
int ripidx;                  Index of the first /RIP language,
                             0 to nlingo-1, or nlingo if there
                             are no /RIP languages
```

```
hasrip=isripu();             Is this a /RIP user?
int hasrip;                  1=yes, 0=no
```

```
hasrip=isripo(unum);        Is that a /RIP user?
int hasrip;                  1=yes, 0=no
int unum;                    user number, 0 to nterms-1
```

Be careful not to use `isripu()` unless you know the "clingo" variable is available. For example, in an interrupt routine such as `hpkrou()` in MAJORBBS.C, only `isripo()` should be used.

Hanging up on a User

If you've decided, for whatever reason, to boot a user off of the BBS, call `byenow()`:

```
byenow(msgnum,p1,p2,...,pn);    say good-bye to a user and disconnect
                               (implicit input:
                               usrnum ... channel to hang up)
int msgnum;                    message number in current .MCV file
                               (don't forget setmbk(), page 66)
TYPE p1,p2,...,pn;            parameters if any (max 12 bytes)
```

This routine will make reasonably sure that your good-bye message gets transmitted to his screen, and then his session will be terminated. You may still get status codes after calling `byenow()`, but you can check the `usrptr->flags&BYEBYE` flag to detect that situation. You will definitely get a call to your `huprou()` entry point.

If you need to do this for a user other than the one you're servicing (other than `usrnum`, that is), then you need to temporarily save `usrnum` and restore it, as in:

```
usnsave=usrnum;
usrnum=othusn;
byenow(LASERCEPT);
usrnum=usnsave;
```

This would do the dirty work for the `othusn` user. Note that `usrptr` and `usaptr` are not involved at this stage at all.

Intercepting User-Connect

You can intercept the moment that a user first connects to the BBS using the (*hdlcon()) handle-connect vector.

```
void (*hdlcon());           Handle-connect vector
```

When any channel, modem, serial, X.25 or LAN, is done establishing connection with the user's terminal, then the function pointed to by this vector gets called. Here are the final events on the different types of channels that occur before the (*hdlcon()) vector gets called:

Modem channel	"CONNECT" received
Serial channel	Any <CR>-terminated string received
X.25 channel	X.3 programming complete (X.29 string sent)
IPX Direct channel	Any <CR>-terminated string received
IPX Virtual channel	Any packet received
SPX channel	Connection established

The (*hdlcon()) vector starts out pointing to the gtansi() routine which is an internal (static) function in MAJORBBS.C. Use (*hdlcon()) just like the parasitic way in which you would use an interrupt vector: save its value (the pointer to some old function), put a pointer to your own function in its place, and then when your own function gets called, make sure to call that function whose pointer you saved (unless you think of something better to do). Here's a simple example:

```
void (*hcsave());          /* save location for old handle-connect vector */

void
brblast(void);             /* blast low-baud rate users on high channels */
{
    if (usrnum >= 32 && usrptr->baud < 9600) {
        setmbk(dddmbk);
        byenow(OTHERBAUD);
        rstmbk();
    }
    else {
        (*hcsave)();
    }
}

void
install_brblast(void);     /* install baud-rate blaster */
{
    hcsave=hdlcon;
    hdlcon=brblast;
}
}
```

The brblast() routine hangs up on slow-modem callers on channels with user number 32 and higher. The install_brblast() routine should be called from your init_xxx() routine (exactly once, of course). It saves the current pointer in the handle-connect vector, and puts a pointer to brblast() in its place.

Now when anyone connects to the BBS, brblast() is called. If their user number is 32 or greater and their baud rate is less than 9600, it sends some goodbye message (politely saving and restoring the current .MCV file handle) and prepares to hang up on the user. Otherwise, the user gets online like normal.

The connect handler that you install by this method is limited in what it can do. Keep in mind that other modules might be intercepting the vector too, and you really shouldn't be depending on one to execute before the other. And if you want a user to log on, you should relinquish complete control and let the normal connect sequence proceed (in other words, call the function whose pointer you saved).

If you want to take over connect-time processing for *multiple status conditions*, then you'll need to set up your own class, state and substate. (Remember the three "context" variables described on page 31?)

```
:
usrptr->class=BBSRV;
usrptr->state=mystate;
usrptr->substt=CONSTEP1;
:
```

If you do this, then your `sttrou()` and `ststrou()` vectors will get called for future events (such as the user entering a line of text or a status condition on that channel). BBSRV is a special class that allows your module entry points to get called without deducting credits or limiting inactivity, etc. -- you're on your own. The "mystate" variable is the handle for your module (the return value of `register_module()`). The `substt` value `CONSTEP1` is a local constant so you can distinguish this type of event in your entry point routines.

When done, you can return control to the powers-that-be and continue with the standard connect process by restoring `usrptr->class` and calling the saved vector value:

```
:
usrptr->class=VACANT;
(*hcsave)();
:
```

Notice that it's important to `resroc` whatever a handle-connect routine does, you should be able to rely on it to either set the `usrptr->state` code or call `byenow()`. (`gtansi()` does assume you're still in the `VACANT` class in some cases.)

There are other moments in the connection process with vectors you can intercept:

```
void (*hdlrng)();           Handle-RING-string vector
```

This routine can be used for auxiliary handling of the "RING" that comes in on modem channels. The routine would get called on every "RING" before the "CONNECT" message. The "RING" and any text that might follow it are available by consulting `margc` and `margv[]`.

```
void (*hdlrng)();           Handle-non-RING-string vector
```

This vector is called when a string other than "RING" is received on a modem channel that is awaiting an incoming call. The string is available by consulting `margc` and `margv[]`. You might use this to handle strings other than "RING" in some special manner.

int (*hdlcnc)(); Handle-CONNECT-string vector

This vector gets called on a modem channel when the first non-RING string is received. The string is available by consulting margc and margv[]. Return values are:

- 1 Ignore the string
- 0 Reset the channel and get ready to receive another incoming call
- 1 Connection complete, the (*hdlcon()) vector will get called after a short pause

You might intercept (*hdlcnc()) if you wanted to handle the parameters of the "CONNECT" string in some special way, or if you were expecting legitimate messages other than "RING" or "CONNECT" to come in.

int (*hdlc25)(); Handle-X.25-connection vector

This vector gets called at the beginning of every X.25 call. You could intercept it to process the parameters of the incoming X.25 call:

margv[0]	"RING"
margv[1]	Caller's network address
margv[2]	"CALLING"
margv[3]	Callee's network address (that of your BBS)
margv[4]	(optionally) User data field (NUL-terminated string)

margv[4] will only be available if margc >= 5 and if you have set the x25udt flag in advance (see the GSBL Reference Guide).

Looking through MAJORBBS.C, you may find that some of these vectors default to pointing to a routine that does nothing at all. If you change the value of one of these vectors, it is still good programming practice to call the routine it originally pointed to from inside of your new routine.

Autosensor Routines

Add-on Options may hook into the autosensing phase of the BBS session -- the very start when we probe the user's terminal for signs of intelligent life. If there are any autosensors active, the "Auto-sensing..." message appears and then all autosensing routines are run in parallel on that channel.

Add-on Options can register autosensing routines to test for particular features in each user's terminal and vote on which languages or protocols he should use. See the ansisns() routine in MAJORBBS.C for an example.

For another example, say that to automatically detect compatibility with the ZEBRATerm terminal software you want to send out a "Z" immediately upon connection, and then wait for a "!" reply. If the BBS gets the reply within 1 second, it knows it's talking to ZEBRATerm. But if it gets nothing for 1 second, it assumes not.

Here's how such a ZEBRATerm autosensor might be coded:

```
int
zebratest(
unsigned snccon,
char *incbuf,
int nbytes)
(
    if (snccon == 0) (
        btuxmt(usrrnum,"Z");
    )
    else (
        while (nbytes--)(
            if (*incbuf++ == '!')(
                zebra(usrrnum)=1;
                setbyprot("/ZEBRA",2);
                return(1);
            )
        )
        if (snccon >= 16)(
            zebra(usrrnum)=0;
            return(1);
        )
    )
    return(0);
)
```

To register your autosensor routine, you need to call regautsns() in your initialization code, for example:

```
:
regautsns(zebratest);
:
```

The code for regautsns() and related routines can be found in AUTSNS.C.

When a user first connects to a BBS all autosensor routines are called repeatedly. Each autosensor eventually returns 1 to indicate it is done, and from that point is no longer called for that session. When all autosensors are done, or 10 seconds elapse, whichever comes first, the autosensing phase is over.

Each autosensor routine will be called with the same three parameters as in the above zebratest() example:

unsigned snccon	count of 1/16 seconds since connection was established with this channel -- the first call is always zero and each call after that is steadily increasing (and nonzero)
char *incbuf	buffer of incoming binary bytes on this channel -- this same data is shared by all autosensor routines
int nbytes	number of bytes in incbuf

As you can see in the example, usrrnum is an implicit input to the autosensor, as well as usrptr and usaptr and other global session variables.

Your autosensor routine will need to return an "int" indicating whether the autosensing is done for that channel.

autosensor return value	1=done autosensing 0=still working on autosensing
-------------------------	--

You can count on the fact that the sncccon parameter will be zero the very first time your autosensor is called for a session, and always nonzero after that for the same session. So use the (sncccon == 0) case to initialize things if you need to. In every call after that, sncccon will be at least one, even if it's 0.000001 second later. In the above example, we transmit the "Z" right off the bat and return 0 (0 because we know we're not done autosensing yet). Otherwise we check for incoming data.

This is a subtle but important point: autosensor routines should check for incoming data before they check for a timeout. This way, if the BBS happens to get tied up with the other channels for an unusually long period, then when it finally does get around to servicing the autosensing channel, it properly handles any data received in the interim. Suppose in the above example that within such a delay of a second or more, a "!" reply did come in. Then zebraTest() gets called with both a timeout condition and data available. Clearly the data available condition should take precedence (as it does in this example).

After checking for incoming data, we take a glance at the watch. If our one second's up, we give up and assume that we're not connected to ZEBRATerm. Remember that other autosensor routines might be at work here so if we don't understand the incoming data we have to ignore it.

This autosensor's ultimate job, when it detects ZEBRATerm, will be to set the zebra[] array element for each user to indicate 1=ZEBRATerm, 0=not. It will also "vote" for the languages that end in "/ZEBRA" with a confidence factor of 3. We'll take a closer look at what this voting business is all about in the next section.

All autosensing is subjected to a 10-second master timeout, so if any autosensor takes more than 10 seconds for any reason, the autosensing period will end anyway. You can change this master timeout if you like by changing the value of the global auswait variable:

```
unsigned auswait;           master autosensing timeout, in 1/16
                             of a second (e.g. 160 = 10 seconds)
```

The healthiest way to change this is probably to be sure you only increase it, as in:

```
auswait=max(auswait,240);
```

Just setting auswait=240 (15 seconds) might cause a problem for another autosensor that needed the master timeout to be at least 320 (20 seconds).

Voting Confidence Factors

All languages start out with a confidence factor of 1. Any autosensor can change the confidence factor of any language to any number between 0 and 100. The voting confidence factors for all languages and users is stored in this 2D array:

```
char *poslng;               pointer to a 2D array of voting confidence
                             factors (varies fastest by language, then
                             by user number, has nlingo*nterms total
                             number of elements)
```

To set the voting confidence factor for the current user (usrnum) for the language ilingo to 5, you would code:

```
poslng[usrnum*nlingo+ilingo]=5;
```

You could vote on all languages with the same terminal protocol suffix with:

```
setbyprot(suffix,value);    set voting confidence factor by protocol
char *suffix;              language name suffix
char value;                voting confidence factor
```

This routine will find all languages with names that end in *suffix* and set their voting confidence factor to *value*. See the example call to `setbyprot()` in the `zebratest()` example, above.

At the end of the autosensing period, the language with the highest confidence factor automatically becomes the language for that channel. If there's a tie, as is often the case, then what happens next depends on the LANGOP offline Configuration option:

```
LANGOP=ASK - Display a numbered list of the languages that have the
             highest confidence factor, and ask the user which one he
             wants to use.
```

```
LANGOP=AUTO - Just go ahead and select one of the languages with the
              highest confidence factor (it picks the language with
              the lowest numbered index, but the Sysop can't count on
              which language that is, unless it's English/ANSI)
```

The voting confidence factors are available throughout a user's session. To determine the top factor, call

```
numcand=cntcand();          determine the maximum voting confidence
                             factor, among all the languages, and
                             how many languages have it
```

Here are the implicit return values of `cntcand()`:

```
int maxcand;                maximum voting confidence factor among the
                             languages

int numcand;                1=one language is clearly the winner
                             >1=number of languages tied for first place
                             (numcand is a global variable and it's also
                             the return value of cntcand())

int fstcand;                the winner, or the first language (lowest
                             index) that is tied for first place
```

5. USER SERVICES

Security (Locks & Keys)

As explained in the System Operations Manual, security on The Major BBS is primarily controlled by locks and keys. Apply locks to features and issue keys to users. When a user requests to use a certain feature, find out if he has a key with the same name as the lock on that feature. A feature can have one or zero locks. A user can have a set of keys by virtue of the class he's in (the class keyring), or by individual ownership.

Note that the question "Can Fred open the SAMPLE lock?" is identical to the question "Does Fred have the SAMPLE key?"

Here is the most versatile routine for testing whether an online user has a specific key or not:

```
ok=gen_haskey(lock,unum,uptr);  Does this user have the key to this
                                lock?
int ok;                          1=yes, let him in  0=no, deny access
char *lock;                       Name of lock on feature / key required
int unum;                          User number of online user
struct user *uptr;                 User structure pointer of online user
```

What follows are some handy variations and alternatives to gen_haskey() that you'll probably use more often:

```
ok=hasmkey(msgnum);              Does the user have the key specified
                                in this offline Security and
                                Accounting option?
int ok;                          1=yes, let him in  0=no, deny access
int msgnum;                       Number of the CNF option
```

This routine checks whether the current user has a key specified by the Sysop in an offline Security and Accounting option. See page 55 for creating CNF options. You could make an offline Security and Accounting option that looks something like this key:

```
SAMPKY   Key required to log on to reserved channels ..... NORMAL
```

Now the Sysop can change this option so that another key is required. All you have to do is:

```
if (hasmkey(SAMPKY)) {
    welcomemyfriend();
}
else {
    sorrynecigar();
}
```

By convention, all CNF options pertaining to Security are stored in level 3 -- Security and Accounting. If you specify any locks of your own in offline Security and Accounting options, we recommend that you try to use one of the four pre-defined lock names for the default values of your option when you can:

DEMO	Everybody gets this key, it's the only one new sign-ups get
NORMAL	Approved users
SUPER	Supervisors or trusted assistants
SYSOP	Top-level access to the BBS

ok=haskey(lock);	Does the user have this key?
int ok;	1=yes, let him in 0=no, deny access
char *lock;	Name of lock on feature / key required

This routine checks if the current user has the specified key. You might store the string in memory with stgopt() and haskey() (instead of reading it each time you need it with hasmkey()) for quicker response.

ok=othkey(lock);	Does the other user have this key?
int ok;	1=yes, let him in 0=no, deny access
char *lock;	Name of lock on feature / key required

This routine checks if the user specified by othusr (user number) and othusp (pointer to user data structure) has a certain key. You can call this routine right after you call instat(), onsys(), or onsysn() (see page 81).

ok=uidkey(uid,lock);	Does the (offline) user have this key?
int ok;	1=yes, let him in 0=no, deny access
char *uid;	User-ID
char *lock;	Name of lock on feature / key required

This routine checks on the access capabilities of a user who is not online at the time.

ok=uhskey(uid,lock);	Does the user have this key?
int ok;	1=yes, let him in 0=no, deny access
char *uid;	User-ID
char *lock;	Name of lock on feature / key required

This routine is universal -- it will tell you if the user has this key, and it will work whether the user is online or not.

Registerable Pseudo-Keys

You can create your own pseudo-keys for users. Say you want to give users access to some feature based upon something. The standard method of locks and keys allows Sysops to make up key names and issue keys either directly to individual users, or to classes of users via the class keyring. But some situations require more flexibility. For example, the "_PORT#xx" pseudo-key implicitly gives each user a special key based upon the BBS channel number they are using for their current session. Here's the corresponding pseudo-key routine from MAJORBBS.C:

```
STATIC int
prtpsk(unum,lock)          /* validate the _PORT# pseudo-key */
int unum;                 /* user number, 0 to nterms-1 */
char *lock;               /* lock name that the key is for */
{
    int chn;

    sscanf(lock,"_PORT#%x",&chn);
    return(channel[unum] == chn);
}
```

and here's how it gets registered:

```
register_pseudok(prefix,rouptr);    Register a pseudo-key routine
char *prefix;                       prefix of the pseudo-key
int (*rouptr)(unum,lock);           pointer to handler routine
int unum;                             user number being checked
char *lock;                           full name of the key required
```

For example:

```
register_pseudok("_PORT#",prtpsk);
```

The registration call says in effect "If anyone asks about users having a key that starts with '_PORT#', then let me make the determination". Now suppose there's some code somewhere like this:

```
haskey("_PORT#2C");
```

Then prtpsk() swings into action and determines whether this user happens to be on BBS channel 2C hexadecimal or not.

See MAJORBBS.C for the other pseudo-key routines for channel group number, spoken language, and terminal protocol.

By convention, and for Sysop sanity, all pseudo-keys start with the "_" underscore character, but nothing enforces this.

Accounting (credits)

User connect time can be controlled or measured with the system commodity called "credits". Credits typically refer to seconds of privileged connect time: If an "approved" user is online for an hour he consumes 3600 credits. A new user doesn't consume credits and can't access many of the features of the system.

There are many other ways credits are used. Certain actions "cost" the user a fixed amount of credits. And credit consumption can vary depending on what service the user is in.

Charging Users

To charge a user credits, you can make use of the `dedcrd()` and `tstcrd()` routines in `ACCOUNT.C`:

Credit testing and charging routines	Actually deducts credits?	Subtracts credits if user is "Exempt" from credit charges?	Will automatically "borrow" credits if user can go into debt?	User
* <code>dedcrd()</code>	YES	NO	YES	current
<code>rdedcrd()</code>	YES	YES	NO	current
<code>odedcrd()</code>	YES	optional	optional	any online
<code>ndedcrd()</code>	YES	optional	optional	any offline
<code>ldedcrd()</code>	YES	optional	optional	any
* <code>gdedcrd()</code>	YES	optional	optional	any
* <code>tstcrd()</code>	NO	NO	YES	current
<code>rtstcrd()</code>	NO	YES	NO	current
<code>otstcrd()</code>	NO	optional	optional	any online
<code>ntstcrd()</code>	NO	optional	optional	any offline
<code>ltstcrd()</code>	NO	optional	optional	any
* <code>gtstcrd()</code>	NO	optional	optional	any

* = routines you're most likely to want to use

All of the `dedcrd()` routines return 1 if sufficient credits were available, or 0 if there weren't enough. The last parameter, "asmuch", decides what to do if there weren't enough: 1=take whatever the user has, reducing his balance to the minimum, or 0=don't take anything.

The `tstcrd()` routines act just like the corresponding `dedcrd()` routines, except that no credits are actually deducted. Use the `tstcrd()` routines if you need to specially handle the case of insufficient credits before any are deducted (for example by exiting a service or issuing a warning).

```

enuf=dedcrd(amount,asmuch);      Deduct credits from current user's acct
int enuf;                        1=had enough, 0=didn't
long amount;                     number of credits to deduct
int asmuch;                      if not enough: 1=take all, 0=none

enuf=rdedcrd(amount,asmuch);     Deduct real credits from online acct
int enuf;                        1=had enough, 0=didn't
long amount;                     number of credits to deduct
int asmuch;                      if not enough: 1=take all, 0=none

enuf=odedcrd(unum,amount,real,asmuch);
int enuf;                        Deduct credits from an online account
int unum;                        1=had enough, 0=didn't
                                user number
long amount;                     number of credits to deduct
int real;                        1=don't put into debt
int asmuch;                      if not enough: 1=take all, 0=none

enuf=ndedcrd(userid,amount,real,asmuch);
int enuf;                        Deduct credits from an offline account
char *userid;                    1=had enough, 0=didn't
                                User-ID
long amount;                     number of credits to deduct
int real;                        1=don't put into debt
int asmuch;                      if not enough: 1=take all, 0=none

enuf=ldedcrd(uptr,amount,real,asmuch);
int enuf;                        Deduct credits from an "active" user
struct usracc *uptr;             account structure residing in memory
                                1=had enough, 0=didn't
long amount;                     pointer to active user structure
int real;                        number of credits to deduct
int asmuch;                      1=don't put into debt
                                if not enough: 1=take all, 0=none

enuf=gdedcrd(userid,amount,real,asmuch);
int enuf;                        Deduct credits from any user's account
char *userid;                    1=had enough, 0=didn't
                                User-ID
long amount;                     number of credits to deduct
int real;                        1=don't put into debt
int asmuch;                      if not enough: 1=take all, 0=none

```

<pre> enuf=tstcrd(amount); int enuf; long amount; </pre>	<pre> Test if user has enough credits 1=had enough, 0=didn't number of credits (don't deduct) </pre>
<pre> enuf=rtstcrd(amount); int enuf; long amount; </pre>	<pre> Test if user has enough real credits 1=had enough, 0=didn't number of credits (don't deduct) (won't take debt or exemptions into account) </pre>
<pre> enuf=otstcrd(unum,amount,real); int enuf; int unum; long amount; int real; </pre>	<pre> Test if user has enough credits 1=had enough, 0=didn't user number number of credits (don't deduct) 1=don't take debt or exemptions into account </pre>
<pre> enuf=ntstcrd(userid,amount,real); int enuf; char *userid; long amount; int real; </pre>	<pre> Test if <i>offline</i> user has enough credits 1=had enough, 0=didn't User-ID number of credits (don't deduct) 1=don't take debt or exemptions into account </pre>
<pre> enuf=ltstcrd(uptr,amount,real); int enuf; struct usracc *uptr; long amount; int real; </pre>	<pre> Test if user account structure has enough credits 1=had enough, 0=didn't pointer to active user structure number of credits (don't deduct) 1=don't take debt or exemptions into account </pre>
<pre> enuf=gtstcrd(userid,amount,real); int enuf; char *userid; long amount; int real; </pre>	<pre> Test if any user has enough credits 1=had enough, 0=didn't User-ID number of credits (don't deduct) 1=don't take debt or exemptions into account </pre>

Credit Consumption Rate

To change a user's credit consumption rate, you can set `usrptr->crdrat` to the credits to consume per minute. For example:

```
usrptr->crdrat=120;      /* consume credits at twice the normal rate */
```

Whenever a user exits a module of The Major BBS, his credit consumption rate is restored to the default value (as specified by the Sysop in the MMUCRR offline Security and Accounting option).

Global Commands

Global commands are commands that users can enter from almost any prompt on the BBS. (One exception is: you can't use global commands while inside the Full Screen Editor.) Making your own global command means two things: making a handler routine, and registering the routine. The handler routine intercepts every line of user input and, if it recognizes your special global command, responds to it and returns true (otherwise returns false). Registering the routine allows the mainline program to call it with each line of user input.

The handler routine has at its disposal all the global variables associated with line input, including `margc`, `margv`, `input`, and so on (see page 74), in addition to global variables for user session information such as `usrptr->`, and `usaptr->` fields (see `USRACC.H` and `MAJORBBS.H`).

IMPORTANT: The global command should be coded efficiently. It must very quickly reject user input (return false) when it doesn't recognize the command. For example, the global command handler should probably never access a database in the quiescent (return false) case.

Here's an example of a global command called `"/now"` to tell the time of day:

```
int
glotime(void)      /* global command for telling the time of day */
{
    if (margc == 1 && sameas(margv[0],"/now")) {
        prf("At the tone, the time will be %s\7\r",nctime(now()));
        outprf(usrnum);
        return(1);
    }
    return(0);
}
```

The check for `(margc == 1)` is necessary, because `margv[n]` is undefined when `n >= margc`. The `sameas()` check is case-ignoring so users can also type `"/NOW"`. The routine returns a 1 if it recognizes the user's input as the global command it's looking for, or a 0 if it does not.

Here are all the possible return values for the global command handler:

- 0 Command not recognized. Executive will pass entire command on to some module's input handler (sttrou(), lonrou(), or lofrou(), as appropriate). You must always return 0 when you don't recognize the incoming command, and especially when margc == 0.
- 1 Command recognized and processed. Executive will ask the module in effect to reprompt by simulating a <CR> from the user, calling the module's sttrou(), lonrou(), or lofrou() routine with margc == 0. The (usrptr->flags&INJOIP) flag will be set so the routine could recognize this condition. If you have any prf() or prfmsg() output, you must do an outprf(usrnum) before you return the 1 (as in the glotime() routine, above).
- 1 Command recognized and processed -- don't reprompt. Executive will not reprompt the user. This is also a return value where no outprf() is likely to take place unless you do it yourself.
- 2 Command recognized and processed -- don't reprompt, but do prf() or prfmsg(). Executive will not ask the module to reprompt, but it will assume you have something in the prfbuf and will do an outprf() for you.

You can look at the hdlinp() routine in MAJORBBS.C for exactly how these return values are used.

The next step is registering the global command as part of your initialization routine (page 29):

```
int glotime(void);                /* this is the prototype */
:
:
void EXPORT
init_myroutine()                 /* the module initialization routine */
{
    :
    globalcmd(glotime);
    :
}
```

TIP: The global command feature has possible utility beyond defining global commands for users. For example, you could make a routine to intercept all user input for diagnostic or management purposes.

You can define up to 25 global command handler routines using this function:

```
globalcmd(rouptr)           define global command handler routine
int (*rouptr)();           pointer to routine
```

All global command handlers can be temporarily disabled for a channel by setting the special NOGLOB flag, as in:

```
usrptr->flags|=NOGLOB;
```

and later cleared with:

```
usrptr->flags&=~NOGLOB;
```

This is done during teleconference chat modes, for example.

Here are a few examples of global commands and where they're coded:

<u>Command</u>	<u>Purpose</u>	<u>Source code</u>
/R <userid>	registry report	REGISTRY.C
/P <userid> <message>	page	MJRTLC.C or ENTTL.C
/#	who's online	MJRTLC.C or ENTTL.C
/L <userid>	lookup user account	MAJORBBS.C
/INVIS	invisible sysop	MAJORBBS.C
/GO <page-name>	global Menu Tree "GO"	MAJORBBS.C and MENUING.C
/RECENT	recent logoffs	GALGLO.C

The following commands are not registered global commands. These are special commands available from all menu pages:

<u>Command</u>	<u>Purpose</u>	<u>Source code</u>
FIND	Search menu pages	MENUING.C and MAJORBBS.C
DISABLE (<i>Sysop only</i>)	Disable a page	MENUING.C and MAJORBBS.C
ENABLE (<i>Sysop only</i>)	Enable a page	MENUING.C and MAJORBBS.C

Full Screen Editor

The Full Screen Editor is a sub-service used by Electronic Mail and Forums for message editing. It allows a user to edit a block of text of a certain number of 80-column lines. See the Operations Manual for instructions on using the editor from the user's point of view.

There are actually two editors, the Full Screen Editor and the Line Editor that depend on whether the user's terminal has ANSI capability or not. Fortunately for you, the developer, this distinction is transparent. The `bgnedt()` routine will make use of what the BBS already knows about the user's terminal, and fire up the appropriate editor.

```
bgnedt(siz,buf,tsiz,topic,whndun,flags)
                                begin editing a message
int siz;                          max size of text
char *buf;                        buffer for text
int tsiz;                         maximum size of topic (incl NUL)
char *topic;                      buffer for topic (NULL if no topic)
int (*whndun)(int quitex);       routine to call when done editing
int flags;                        special editor option bits
```

An excerpt from MAJORBBS.H, defining the bits of the last parameter:

```
#define ED_READON      2  /* flags that can be passed to bgnedt() */
#define ED_CLRTOP     4  /* "read only" mode */
#define ED_CLRTEXT    8  /* clear topic buffer upon entry */
#define ED_FILESD    16  /* clear text buffer upon entry */
#define ED_FILESD    16  /* use "file" flavor of editor */
#define ED_LINEMO    32  /* force use of the line editor */
#define ED_FIXTOP    64  /* don't allow changing of the topic field */
```

(Note: the `ED_FILIMP` flag that appears in MAJORBBS.H is not supported.)

Call `bgnedt()` to allow the user to begin entering text. You might as well make the `siz` parameter a multiple of 80 bytes plus 1 -- only an integral number of 80-column lines will be available to the user. The buffer should be somewhere that will stay active and available throughout the editing process (a subset of the Volatile Data Area is ideal for this -- just make sure you've allowed enough room with `dclvda()`). If you want a topic field, allocate another buffer for it (up to 51 bytes long) that has the same durability (for example, another portion of the Volatile Data Area).

After you call `bgnedt()`, the editor will usurp your state and substate code for the entire editing session. That means, for example, that your module's hang-up entry point, `huprou()`, will get called with the editor's state in effect in the event that the user hangs up while still in the editor.

If you want to detect that condition (and the editor is active on the channel due to your module's invocation, of course), you need to set it up somehow. Remember that your `huprou()` entry point will be called regardless of what state or module the user is in. You can detect that the user was editing on your behalf by setting a flag when you call `bgnedt()` and clearing it when your `(*whndun)()` routine gets called.

```
editmr(imradr)                    specify import message routine
int (*imradr)();                  address of import message routine

got=(*imradr)(msgno);             call to import routine ("New" command)
int got;                          1=message imported 0=error
long msgno;                       number of message to import
                                   (user typed this in)
```

The editor may be set up to allow users to import other messages into the message that they are editing. This is done by calling the edtimr() routine immediately after calling bgnedt(). Then when a user specifies <Ctrl-N> for "new" in the editor, he has a choice of importing another message or clearing the current message buffer. The "imradr" routine is passed the message number specified by the user, and is expected to do the actual import by filling the editor buffer (and possibly setting the topic and other items).

Your (*whndun)() routine must restore your state and substate (usrptr->state and usrptr->substt) to values for your own module, and prompt the user for the next action (the next question after the editor is over).

The (*whndun)() routine is passed one of these values:

- 0 user wants to save the editing he's done
- ED_QUITEX user wants to quit and abandon the results of his editing

You should check this flag to see if your code should save the buffer or discard it. (You must remember where the buffer is, it's what you passed to bgnedt() in the buf parameter.)

The return value of (*whndun)() can be one of these:

- 1 Ok, exiting the editor (I've restored my state and substate and prompted the user).
- 0 Exit the editor, and exit this module too (your sttrou() should return 0, and we should exit to the parent menu).

Here's an example of a Sysop Feedback Forum using the Full Screen Editor:

```

/*****
 *
 * GALFBK.C
 *
 * Copyright (C) 1989-1994 GALACTICOMM, Inc. All Rights Reserved.
 *
 * Feedback to Sysop (sample module discussed in the
 * Developer's Guide for The Major BBS)
 *
 * - RNStein, January 1989
 *
 *****/

#include "gcomm.h"
#include "majorbbs.h"
#include "galfbk.h"

STATIC int fbkinp(void);
STATIC int fbkdun(int flags);
STATIC void fbkfin(void);

int fbkstt; /* Feedback module state number */
FILE *fbkmb; /* feedback configuration variables */
FILE *fbkfp; /* feedback text file */

struct module fbkmodule=(
    "", /* module interface block */
    NULL, /* name used to refer to this module */
    fbkinp, /* user logon supplemental routine */
    dfsth, /* input routine if selected */
    NULL, /* status-input routine if selected */
    "injoth", /* "injoth" routine for this module */
    NULL, /* user logoff supplemental routine */
    NULL, /* hangup (lost carrier) routine */
    NULL, /* midnight cleanup routine */
    NULL, /* delete-account routine */
    fbkfin /* finish-up (sys shutdown) routine */
);

```

```

#define TPCSIZ 40 /* maximum characters in topic */
#define FBKSIZ 1921 /* max chars in feedback (for 24 lines) */

struct fbkusr { /* feedback to sysop user data block */
    char text[FBKSIZ]; /* text buffer */
    char topic[TPCSIZ]; /* topic buffer */
};

#define fbkptr ((struct fbkusr *)vdpapr)

void EXPORT
init__feedback() /* initialize feedback stuff */
{
    stzcpy(fbkmodule.descrp, gmdnam("GALFBK.MDF"), MNMSIZ);
    fbkstt=register_module(&fbkmodule);
    fbkmb=oprmsg("GALFBK.MCV");
    dclvda(sizeof(struct fbkusr));
}

STATIC int
fbkinp(void) /* feedback handler */
{
    setmbk(fbkmb);
    if (margc == 1 && sameas(margv[0], "X")) {
        return(0);
    }
    do {
        bgncnc();
        switch(usrptr->subst) {
            case 0:
                cncchr();
                prfmsg(HELLO);
                outprf(usrnum);
                bgnedt(FBKSIZ, fbkptr->text,
                    TPCSIZ, fbkptr->topic, fbkdun, ED_CLRTOP+ED_CLRTEXT);
                break;
        }
    } while (!endcnc());
    outprf(usrnum);
    return(1);
}

STATIC int
fbkdun() /* feedback editing when-done */
{
    int quitex;
    char *cp;

    usrptr->state=fbkstt;
    setmbk(fbkmb);
    if (quitex == 0) {
        for (cp=fbkptr->text ; *cp != '\0' ; cp++) {
            if (*cp == '\r') {
                *cp='\n';
            }
        }
        if ((fbkfp=fopen("GALFBK.TXT", FOPAA)) == NULL) {
            catastro("Cannot open GALFBK.TXT for append!");
        }
        fprintf(fbkfp, "*** From %s on %s at %-5.5s %s\n%s\n\n",
            usaptr->userid, ncddate(today()), nctime(now()),
            fbkptr->topic, fbkptr->text);
        fclose(fbkfp);
        prfmsg(THANKS, usaptr->userid);
        outprf(usrnum);
    }
    return(0);
}

STATIC void
fbkfin(void) /* feedback shutdown */
{
    clsmg(fbkmb);
}

```

Here are the offline Text Blocks that go with this example:

LEVEL6 {}

```
HELLO (<ESC>[0;1;32m
```

```
Hello, and welcome to the Sysop Feedback Forum. This service is provided  
to encourage your comments and criticisms.
```

```
When you are done typing, you can hit <ESC>[37m<Ctrl-G><ESC>[32m to save your comments.
```

```
) T Feedback welcome message
```

```
THANKS (<ESC>[0;1;32m
```

```
Thank you for taking the time to leave your comments, <ESC>[33m%s<ESC>[32m!
```

```
) T Feedback thanks for comments
```

This source code and all support files are available on the Galacticomm Demo System, (305) 583-7808, in a file named GALFBK.ZIP. (Note: <ESC> represents the ASCII escape code '\x1B'.)

When a user selects this service, he is introduced to it with the HELLO{} message, and a "(N)onstop, (Q)uit or (C)ontinue?" choice. Then he enters the Full Screen Editing mode, where he types in a topic and a message. When the user hits <Ctrl-G>, the topic and message (along with other information) are appended onto the end of the text file GALFBK.TXT. The Sysop can periodically read this file and delete it.

struct module fbkmodule

This module structure defines the text-line input entry point (fbkinp()), the status handler (the standard system default status handler dfsth()), and a shutdown routine (fbkfin()).

struct fbkusr

This is the structure template for this module's use of the Volatile Data Area. The body and topic of the feedback will be stored here. The "fbkptr" macro casts vdptr into a pointer to an fbkusr structure, for convenient coding.

init_feedback()

This initialization routine registers the feedback module and opens the GALFBK.MCV file with the text blocks for the module. The call to dclvda() declares this module's requirements for the size of the Volatile Data Area.

fbkinp()

This is the input text line handler for the module. It is coded with the standard command concatenation and "X"-to-exit features, although neither of them are actually used. They're in there to make it easier for you to edit this source code into a module of your own. But the only action happening in fbkinp() is this: when the user enters the module, he's greeted, and then shuffled straight off to the Full Screen Editor.

fbkdun()

This function is the when-done routine associated with the module's invocation of `bgnedt()`. (Notice how it's identified in the `bgnedt()` call?) If the user did not `<Ctrl-O>` quit the editing, then the text is written to disk. First the `'\r'` line-terminators that the FSE uses are translated into the `'\n'` line-terminators that `fprintf()` likes. Then the file `GALFBK.TXT` is opened in append-ASCII mode. Then the User-ID, date, time, topic and message body are written to the file. Finally the user is thanked for his efforts. Returning 0 means to return to the parent menu page, as opposed to staying in this module.

Full Screen Data Entry

FSD can perform the following functions:

- o Display data
- o Enter data, full-screen mode
- o Enter data, linear mode

Full-screen entry mode requires ANSI capability and a large enough user screen to hold the entire template. Data displaying, or linear entry, can take place whether the user has ANSI capability or not. To use FSD with The Major BBS, you'll need to create these:

- o Template (in `.MSG` file, level 99)
- o Field specification string (usually in memory)
- o Memory for the session's variable-length data structures
- o Default answer string (usually created on the fly)
- o Field-verification routine (optional)
- o When-done routine (process answers, restore state/substate)
- o Calls to `FSDBBS.C` routines

Procedure:

1. Create a Template in an `.MSG` file. (See `UEDANSI{}` in `BSSSUP.MSG` for an example. See `FSD.H` for a complete definition of the template format. `FSDBBS` will automatically translate to `'\r\n'` terminators.) You will probably have a different template for ANSI users than for non-ANSI users.
2. Make a permanent copy of a Field Specification String in memory. (See `uinfsp[]` in `UINFED.C` for an example. See `FSD.H` for the complete specifications of this format also.)
3. Find out how much memory to allocate:

Make a call like this:

```
nbytes=fsdroom(tmpmsg, fldspc, 0);   Entry session
nbytes=fsdroom(tmpmsg, fldspc, -1);  Displaying
```

If the template
is for an:

Make a call like this from your `init_routine()` and identify the above Template and Field Specification strings (after opening the appropriate `.MCV` file of course). This will tell you the size of the region you must provide to support data entry or display. Call `fsdroom()` for all templates/field specification combinations you will be using to make sure you'll have enough room for all of them.

4. Allocate the space `fsdroom()` requires. (You can use `dclvda()` to put it in the Volatile Data Area.)

(By the way, `fsdroom()` will need to be called again, immediately before the display or entry session begins.)

5. Format your default or original answers into an Answer String or use "" to default to all blank. (See the use of `uinfmt[]` in `UINFED.C` for an example. See `FSD.H` for the specifications of an answer string.) The answer string can come from `getmsg()`, but it cannot be in the `prdbuf`. `vdatmp` is a good candidate, making sure it's big enough. Be sure to use only legal values in your default answer string (per your own field specifications string and validation routine).
- 6a. To display data call:

```
fsdroom(tmpmsg, fldspc, -1);
fsdapr(sesbuf, seslen, answers);
fsddsp(fsdrft());
```

- 6b. To begin a full-screen entry session, call:

```
fsdroom(tmpmsg, fldspc, 1);
fsdapr(sesbuf, seslen, answers);
fsdrhd(title);
fsdbkg(fsdrft());
fsdego(fldvfy, whndun);
```

- 6c. To begin a linear entry session, call:

```
fsdroom(tmpmsg, fldspc, 0);
fsdapr(sesbuf, seslen, answers);
fsdego(fldvfy, whndun);
```

Notes:

Fields are numbered 0 to N-1. How do you tell FSD what N is? N is computed from the field specs by `fsdroom()` and stored in `fsdscb->numfld`. The number of fields that are also represented in the template is `fsdscb->numtpl`, which usually equals but never exceeds N. (You can't display or enter a field outside the range 0 to `fsdscb->numtpl-1`.)

`tmpmsg` is the code for the template stored in the level 99 option in the `.MSG` file.

For entry sessions, you can supply a custom field-verification routine.

Remember that `fsdroom()` in step 6 outputs a bunch of stuff to the `prdbuf`. This stuff must be untouched between `fsdroom()` and `fsdapr()` calls.

The results of `fsdapr()` are all in the `sesbuf`. The `seslen` parameter is the size of `sesbuf`. This means that after calling `fsdroom()` and `fsdapr()`, you can call the other routines (`fsddsp()`, `fsdrft()`, `fsdrhd()`, `fsdbkg()`, `fsdego()`) any time later and in any order as long as you maintain the `sesbuf` passed to `fsdapr()`.

If you have any `prf'ing` you want to show up immediately before the entry/display, be sure and do it AFTER the call to `fsdapr()`, which leaves the `prfbuf` empty.

`vdaptr`, or a subset of `vdaptr`, is a good thing to use for `sesbuf`.

The `(*whndun)()` routine must restore your `usrptr->state` and `usrptr->subst` codes, as well as handle the end of the session.

The title in `fsdrhd()` is only for smooth operation for RIPscrip users -- this should simply be a character string title for viewing above the entry screen, for example "Contact Database".

Avoiding Fields

If your program needs to conditionally blank out some fields in the display, you need to (1) modify the template, and (2) flag the appropriate fields as "avoid". For (1), use the `tpwipe()` routine on the results of `fsdrft()` (before passed to `fsddsp()`) to modify the supporting text for the appropriate fields of the template. For (2), set the `FFFAVD` flag for the fields to be avoided (see `FSD.H`) after calling `fsdapr()`.

For example, to display all data but blank out field 5 and some of the supporting text surrounding field 5, you could code something like:

```
char *tp;

fsdroom(tp,msg, fldspc,0);
fsdapr(sesbuf,seslen,answers);
tp=fsdrft();
tpwipe(tp,5,1,1);
fsdscb->flddat[5].flags|=FFFAVD;
fsddsp(tp);
```

This works almost identically for "avoiding" fields in a full screen entry mode, except you need to intercept things before `fsdbkg()` is called (instead of before `fsddsp()`). On the other hand, to show a "protected" field that the user can see but can't change, the `FFFAVD` flag should be set after `fsdbkg()` is called, but before `fsdego()`, and don't call `tpwipe()` at all.

In linear entry mode, you just need to set the `FFFAVD` flag for the appropriate fields after calling `fsdapr()`.

Getting Answers After a Session

After an entry session is over there are a few ways to get the answers. See `FSD.H` for more details.

<pre> stg=fsdnan(fldno); char *stg; int fldno; fsdfxt(fldno,buffer,maxlen) int fldno; char *buffer; int maxlen; index=fsdord(fldno) int index; int fldno; </pre>	<pre> Get a field's answer pointer to answer field number 0 to N-1 Store answer for field into buffer field number 0 to N-1 store the answer here don't use more than this many bytes Find index of multiple choice answer. Returns -1 if the answer was not one of the ALT='s. the index, 0 to N-1, for the answer according to the N possible "ALT=" alternate values for the field field number 0 to N-1 </pre>
--	--

Handling Answers at Other Times

After a session, the data structures allocated by fsdapr() allow quick access to pieces of the answer string. But at other times, the following routines from FSD.C can be used to deal with answer strings (see FSD.H for more details):

<pre> length=stranslen(answers); int length; char *answers; value=fsdxan(answers,name); char *value; char *answers; char *name; fsdpan(answers,name,value); char *answers; char *name; char *value; fsddan(); </pre>	<pre> Find length of an answer string length including final double '\0'. answer string Get the value of a field of an answer, returning "" if not found. pointer to answer string value answer string name of answer Put a new value into an answer string. answer string name of answer pointer to answer string's new value Delete the answer fsdxan() just found </pre>
--	--

Here's an example of creating an answer string from scratch using sprintf():

```

sprintf(answers,"NAME=%s%CRANK=%s%CSERIALNO=%s%c",name,'\0',
rank,'\0',
serno,'\0');

```

For an example of a simple module that uses Full Screen Data Entry, download the file GALCTX.ZIP from the Galacticomm Demo System at (305) 583-7808.

File Transfer

Uploads

Assuming that you've already taken care of all interactive aspects of your application (if you haven't, see about creating interactive modules on page 29), then here are the steps to take to add file uploading capability:

1. In your source code, include the following special-purpose header file:

```
#include "filexfer.h"
```

2. Code your own upload handler routine. The upload handler routine includes all the ways that the file transfer service will be asking you for assistance after you've turned control over to it. This is most of the work, and it's discussed in detail below.
3. Call fileup() when you want to begin an upload, or to present the user with his protocol choices.

```
fileup(filnam,prot,fuphdl);      File upload
char *filnam;                   name of file ("=multi)
char *prot;                     protocol code
int (*fuphdl)(int fupcod));    upload handler routine
```

The filnam parameter is only used for indicating single file ("FILENAME.EXT") or multiple files (""), and for inclusion in some user prompts. Your upload handler routine will have to come up with the full file path in the FUPPTH, FUPBEG, and FUPEND cases. (If you do get a file name in ftfsb->fname, it came from the protocol, otherwise you'll get ".")

Invalid values for prot are handled appropriately, so you can pass unedited user input in the protocol parameter. The last parameter to fileup() is the address of your upload handler routine.

Calling fileup() usurps your state and substate (usrptr->state and usrptr->substt). It's up to your FUPFIN exit point to restore them. (More on this subject below.)

Upload Protocol Codes

single-file: "A" "M" "C" "1" "V"

single-file or multi-file: "B" "G" "Z" "K"

to log off after uploading: append "!" to any of the above

menu of download protocols: "?" or ""

To validate an upload protocol code, you could use valupc():

```
ok=valupc(proto);           Is this a valid upload protocol?
int ok;                    1=valid, 0=invalid
char *prot;                protocol code string
```

Upload Handler Routine

This routine is a collection of what we call "exit points". After your special-purpose module hands control over to the general-purpose file transfer service, FTF, there are several cases when FTF is going to need to consult back with your application.

Imagine you hire a decorator to remodel your house, and you move out temporarily so you're not in his way. He'll still need to get back in touch with you to go over the pool plans, verify the wallpaper, get your plumber's phone number, and most importantly, to tell you when you can move back in. This handler routine is the means for the FTF to "get back in touch with" your application, for all kinds of specific reasons.

For example there are three occasions when your application needs to come up with the file's full DOS path name:

```
case:      FTF service needs the DOS path in order to:
FUPBEG     create the file
FUPEND     update the file's time and date
FUPPTH     check if there's an existing file that's
           older or smaller (for ZMODEM features)
```

Other exit points are cues for your application to verify that the file name is valid, check if the user has authorization to upload it, handle a completed upload, handle an aborted upload, to "import" a file that's already available on disk, and most important of all, when the file upload session is over, for your application to prompt the user and resume control of his channel.

Here's an informal pseudo-code template for an upload handler routine. This is mostly in C code, but it's liberally laced with English descriptions where appropriate.

```
int
fupxxx(                /* Handle the application-specific */
int fupcod)           /* aspects of your uploads */
{                      /* (fupcod=code for each aspect) */

    int rc=0;

    setmbk(whatever your application uses);
    (be sure to set any other appropriate globals)
    switch(fupcod) {
    case FUPPTH:        /* Where would we put this file? */
        sprintf(ftfbuf,"<DOS path for the file>",ftfscb->fname);
        rc=<resume upload> ? 2 : 1;
        break;
    case FUPBEG:        /* Begin uploading this file */
        if (user can't upload this file) {
            sprintf(ftfbuf,"He can't upload this file because.");
        }
        else {
            sprintf(ftfbuf,"<DOS path for the file>",ftfscb->fname);
            reserve file
            rc=1;
        }
        break;
```

```

case FUPREF:                /* Refer to file, don't upload it */
    strcpy(<somewhere>,ftfbuf);
    break;
case FUPEND:                /* This file uploaded successfully */
    unreserve file
    record a completed upload
    sprintf(ftfbuf,"<DOS path for the file>",ftfscb->fname);
    break;
case FUPSKP:                /* This file upload aborted */
    unreserve file
    record an aborted upload
    break;
case FUPFIN:                /* End of uploading session */
    usrptr->state=your state
    usrptr->substt=your substate
    prompt(whatever comes next); /* (don't call outprf()) */
    rc=1;
    break;
case FUPHUP:                /* Channel hanging up */
    the FUPFIN exit point never got called, clean up as req'd
    break;
    )
    return(rc);
)

```

You might find it handy to download FUPXXX.C from the Galacticomm Demo System, (305) 583-7808, which contains the above pseudo-code, and then edit it line-by-line into what your upload handler will need.

In addition to the fupcod input to your upload handler routine, there are several global variables that you can always assume will be available: usrnum, usrptr, usaptr, and vduptr. In addition, these FTF variables are available:

```

struct ftfscb *ftfscb;      Session Control Block (see FTF.H) for the
                           current file transfer session

struct ftfpsp *ftfpsp;     protocol specifications (see FTF.H) for the
                           current file transfer session

char *ftfbuf;              multi-purpose buffer (context dependent)

```

If you need any other global variables, be sure to set them up in your routine. The meaning of your routine's return value depends on the type of exit point (which is coded in fupcod). These will be discussed individually for each exit point. In some cases, no return value is expected. You should return 0 in each of those cases to allow for future expansion.

Now we'll go into each of the exit points in detail. To simplify the discussion, we'll pretend that FTF is a person telling your application what it needs.

```

I,me,my = FTF file transfer service
You,your = application software

```

From the remodeling analogy, this is like the decorator talking to the homeowner.

FUPPTH - Where would we put this file?

Tell me what DOS path you plan to use for this file coming up, and store that path in `ftfbuf`. If the protocol was capable of telling us a file name, I've put it in `ftfscb->fname`, otherwise `ftfscb->fname` is "". Usually you'll return 1 in this case.

On the other hand, if you have a file fragment left behind from an earlier aborted upload of the same file, then give me the path for that file fragment and return 2. I may try to resume the upload if the protocol is capable (e.g. ZMODEM). You should only return 2 if you're reasonably confident that the existing file is the result of an aborted upload. Otherwise, a useless mix of two different files might end up on the disk.

You could also just return 0 (and skip putting the path in `ftfbuf`) if you don't plan on supporting file upload resume after abort, and don't plan on supporting the upload-if-exists/newer/bigger options that ZMODEM is capable of.

FUPBEG - Begin uploading this file

Verify whether the user is allowed to upload this file. See `ftfscb->fname` and `ftfscb->estbyt` for the file name and size, if the protocol has supplied them. The file time and date may be in one of three forms:

<u>Protocol provides:</u>	<u>ftfscb->dosdat,dostim</u>	<u>ftfscb->unxntim</u>
No information about date & time	0,0	0L
DOS time and date formats	date,time	0L
UNIX seconds since 1/1/70	0,0	UNIX time

See page 178 about time and date formats and handling routines. See page 182 for routines to read and set file time and date. After the file is uploaded I'll stamp this time and date on the file (if any), as long as you provide me with the proper path in the FUPEND exit point.

The main reason for the FUPBEG exit point is for you to check this user's upload permission and any other possible restrictions. Here are some things you might check for:

- o Does the file name have the proper syntax?
- o Does the file name conflict with a reserved name? (For example, CON.TXT is an alias for the Sysop's console!)
- o Does this user have permission to upload this file?
- o Does this user have permission to overwrite an existing file?
- o Are too many users opening files at once (thereby using up all file handles)?
- o Will users be able to use up all available disk space?
- o Will users be able to upload a very large number of small files, making directory access very slow?
- o If charges are associated with upload, can the user afford to pay?
- o Could this file name possibly conflict with one of the other online users who are also using your application?
- o Will other users online be able to see/download/modify this file while this user is in the process of uploading it?

If it's not OK to upload, return 0 and put an explanation of some kind in `ftfbuf`. The explanation should be a complete sentence (beginning with a capital letter and ending with a period), for example "You don't have access rights to that file.". The explanation can be up to 79 characters long, not including the terminating NUL '\0'.

If it's OK to upload, return 1 and tell me what DOS path to use for the file (store it in `ftfbuf`). Specify the maximum allowable size for this file, in bytes, in `ftfscb->maxbyt`. If there truly is no maximum size limit, then just leave `ftfscb->maxbyt` at the default value `MAXLONG` (about 2 gigabytes, see `FTF.H`).

You can check `ftfscb->estbyt` yourself if you like, and call things off if the file's going to be too big, or you can leave this work to me. Either way, you should put some kind of size restriction in `ftfscb->maxbyt`. Hacked terminal software could theoretically claim to be uploading a 1000-byte file then proceed to upload a 1,000,000,000-byte file.

Setting `ftfscb->maxbyt` does two things for you. I'll immediately make sure that `ftfscb->estbyt` doesn't exceed your limit (and abort the transfer if it does). And I'll also keep tabs on the size of the file while it's being uploaded, and abort if the limit is exceeded.

An important feature of the file uploading service is that you can count on the fact that for every `FUPBEG` call, there will be exactly one call to either `FUPSKP` (upload of this file aborted) or `FUPEND` (upload of this file was successful), except in extreme cases such as power loss.

FUPREF - Refer to file, don't upload it

You'll never get this case if you haven't willingly and knowingly set the `ftuptr->flags|=FTFREF` flag after you called `fileup()`. This all has to do with uploading a file "by reference". Electronic Mail and the Forums have this ability. The Sysop can "upload" a file that already exists on the BBS's disk using the "F" file-import protocol. The file may stay where it is, and the E-mail or Forum message that it's attached to just "refers" to the real location of the file.

You identify your application's capacity for upload-by-reference by setting the `ftuptr->flags|=FTFREF` flag immediately after you call `fileup()`. By the way, that's all you identify by setting `FTFREF` -- your capacity for upload by reference. You don't have to be concerned with the user's authority to use file importing. I'll only allow this if he has the key specified by the offline Security and Accounting option `FIMLOCK`, which is `SYSOP` by default.

Here's the situation if I ever happen to get around to calling the `FUPREF` exit point: The user hit the "F" protocol, and either the path he specified had no colon in it (in which case I assumed upload by reference was desired), or I confirmed with him that it would be OK to import this file by reference instead of actually making a copy of it.

I'm not going to use the return value from your `FUPREF` exit point, but you should still return 0 to allow for future features. If I call `FUPREF` at all, I'll only call it after a `FUPBEG` where you returned 0, and immediately before I call `FUPEND`. When `FUPREF` is called, `ftfbuf` contains the path just as your `FUPBEG` handler left it. This is your baby now -- you asked for it -- so do whatever you have to do to keep track of this uploaded-by-reference file.

FUPEND - This file uploaded successfully

The file was uploaded successfully. The actual size of the final file is available to you in `ftfscb->actbyt`. (If the upload was resumed, using ZMODEM's resume-after-abort feature, `ftfscb->actbyt` is the total bytes in the file, not just the portion stuck on in this session. There's no way to find out whether a resume took place or not, or to figure the size of the portion.)

I need to know the DOS path for this file one more time (again, store it in `ftfbuf`) so I can set its time and date. If you don't want me to store the time and date, just put an empty string in `ftfbuf`.

If you're in the habit of checking against conflicts or collisions with other users, now's the time to recognize that this user is all done with this file. So you can "unreserve" it if you did any reserving in the FUPBEG exit point.

I'm not expecting any return value from either your FUPEND or FUPSKP exit points, so you should return 0.

FUPSKP - This file upload aborted

The current file upload has been aborted for some reason. The size of the fragment is available to you in `ftfscb->actbyt`. If you don't want fragments of aborted uploads lying around you need to delete the file now. You can do this by coming up with the file path name and passing it to `unlink()`.

Here also, if you've been reserving the file name or a file handle since FUPBEG, now's the time to unreserve it.

FUPFIN - End of uploading session

This step winds up the upload session and returns control to your regularly scheduled program. This is distinguished from FUPEND which only winds up from the upload of a single file. So for multiple file uploads there could be several FUPBEG/FUPEND pairs (or FUPBEG/FUPSKP if things didn't work out).

In `ftfscb->actfil` you'll find a count of the total number of files successfully uploaded. In `ftfscb->tryfil` is the total files that we tried to upload. Of course it's always the case that `actfil <= tryfil`. When `actfil < tryfil`, not all the files made it. I've already told the user all about this, including why the last transfer aborted, or why the last of possibly several files were skipped.

Since you're taking back control of this channel, it's up to you to set things straight for what's up next for this user. Here are two alternatives:

You want control back

- o Restore your `usrptr->state`
- o Restore your `usrptr->subst`
- o Prompt the user (you don't need to call `outprf()`)
- o Return 1

You want to return to the parent menu

- o Restore your `usrptr->state`
- o Say bye to the user if you wish (you don't need to call `outprf()`)
- o Return 0

Either way, if you're prompting or saying goodbye using `prfmsg()`, you need to be sure to set your message block pointer using `setmbk()`.

The pseudo-code for FUPFIN handling on page 108 assumes you want to take control back. Here's an alternative pseudo-coding of the FUPFIN exit point to allow you to exit to your module's parent menu page:

```
case FUPFIN:                               /* End of uploading session    */
  usrptr->state=your state                 /* (don't call outprf())          */
  prompt(exiting);
  rc=0;
  break;
```

This may be appropriate if your module really doesn't want to regain control of the channel when the upload is done. In this case you only need to restore the user's state code (module number), not the substate. Your module never actually regains control of the user's channel. You can prompt him with some parting words, but you don't need to. For consistency you should do whatever you normally do when the user exits from your module to the parent menu.

In the case where you're supposedly retaking control, you may have to call `condev()`, and possibly wind up returning control to your parent Menu Tree menu afterward. This would be the case if you were about to return to your module's own internal main menu, and you found out you had gotten where you are through command concatenation. See page 78 about this whole `condev()` business.

Another handy feature of the file transfer service is that each `fileup()` invocation is followed (eventually) by exactly one FUPFIN or FUPHUP exit point invocation, except in dire cases (power loss for example).

FUPHUP - Channel hanging up

This is the alternative to FUPFIN that occurs when the user or the channel is hanging up for some reason in the middle of the upload session. No return value is expected, so you should return 0. You don't need to worry about termination of the individual file upload, if one had been in progress, because FUPSKP will have been called already. But if there's any session-level (as opposed to file-level) cleanup to be done, now's your chance. For example, in `ESGUTL.C`, FUPHUP is used to store the user's Forum quickscan information back to disk.

This brings up a tricky point that you may want to be aware of. When a user disconnects in the middle of one of your uploads, your module's own hang-up entry point (see page 38) will be called eventually, as it always is at logoff. But you may not be able to recognize that the user was "in" your module because his `usrptr->state` will be that of the file transfer's state code. One way out of this is your FUPHUP exit point. When the file transfer service's own `huprou()` gets called, it will call your FUPHUP exit point in your upload handler routine. That's when you can do your module's last minute housekeeping on this channel.

Uploading Example #1

Here's a very simple example of a module that uploads files on a BBS. Many shortcuts have been taken in this code for the sake of brevity. It uses a minimum of features, has few conveniences, and has none of the security precautions that should be in place before putting software online for users to access. It's sole purpose is to introduce you to the components of file uploading. You can download the source code and other files relevant to this example from the Galacticomm Demo System at (305) 583-7808. Look for GALUPX.ZIP in the File Libraries.

```

/*****
 *
 * GALUPX.C
 *
 * Copyright (C) 1994 GALACTICOMM, Inc. All Rights Reserved.
 *
 * Uploading example.
 *
 *
 * - R. Stein 12/6/93
 *
 *****/

#include "gcomm.h"
#include "majorbbs.h"
#include "filexfer.h"

STATIC int uplinp(void);
STATIC int fupup(int fupcod);

int uplstt; /* Uploading module state number */
struct module uplmodule={"",NULL,uplinp,dfsth};

void EXPORT
init_uploader(void) /* Uploader initialization */
{
    stzcpy(uplmodule.descrp,gmdnam("GALUPX.MDF"),MNMSIZ);
    uplstt=register_module(&uplmodule);
    mkdir("UPLDIR");
}

STATIC int
uplinp(void) /* Uploader input handler */
{
    switch (usrptr->substt) {
    case 0:
        prf("Name of file to upload: ");
        usrptr->substt=1;
        break;
    case 1:
        if (margc == 0) {
            return(0);
        }
        fileup(stzcpy(vdaptr,margv[0],"?",fupupl);
    }
    outprf(usrnum);
    return(1);
}

int
fupupl( /* Handle the application-specific */
int fupcod) /* aspects of the upload example */
{ /* (fupcod=code for each aspect) */
    int rc=0;

    switch(fupcod) {
    case FUPBEG: /* Begin uploading this file */
    case FUPEND: /* This file uploaded successfully */
        sprintf(ftfbuf,"UPLDIR\\%s",vdaptr);
        rc=1;
        break;
    case FUPFIN: /* End of uploading session */
        usrptr->state=uplstt;
    }
    return(rc);
}

```

This module allows users to upload files into the UPLDIR subdirectory of the BBS. As is the case with most modules, Sysops need to create a module page somewhere in their Menu Tree that uses it, normally the child page of some menu. When users are online and choose this service they are asked to type in a file name. When they do, control is turned over to the upload service and the user chooses a protocol. After the upload is complete, the user is returned to the parent menu page. Here is a discussion of the major components of this program.

struct module uplmodule

This module identifies only one custom entry point: `uplinp()` for the `sttrou()` text line input handler. The default status handler, `dfsthm()` is the `strou()` entry point for unusual status conditions.

init_uploader()

The initialization routine for this module registers the module, using the description in the corresponding module definition file. It also creates the UPLDIR subdirectory to store the uploaded files, if one doesn't exist already.

uplinp()

This routine handles text line input from the user after he selects this upload service. Upon entry, the user is prompted to enter a file name. If the user just hits RETURN, he is returned to the parent menu without uploading. If he types in a file name, that name is stored in his Volatile Data Area, and then he is handed over to the file transfer service. The middle parameter to `fileup()` is "?" to give the user a list of available upload protocols.

fupupl()

This is the upload handler routine as described starting on page 108. The FUPBEG and FUPEND exit points are used by FTF to get the file's full path name for opening the file, and for setting its time and date. The FUPFIN exit point merely retorets the upload service's state code and returns 0, which requests that the user be returned to the parent menu page. All other exit points simply return 0

Potential Improvements to Upload Example #1

Here are some of the features left out of this brief example that you should consider if you are using uploads in your application:

- o Limits on file size and quantity in the upload directory
- o Checking for conflicts between the file name and DOS devices
- o Deleting the fragment left behind from an aborted upload
- o Multiple-file uploads
- o Sysop-configurable prompts in an .MSG file
- o Sysop-configurable upload directory
- o Formal declaration and limitation on VDA usage
- o Command concatenation
- o Automatic reprompt after "/p" page command, etc.
- o Full module structure in source code, with comments

All of these features are included in upload example #2.

Uploading Example #2

```
/******  
*  
* GALUPX2.C *  
* *  
* Copyright (C) 1994 GALACTICOMM, Inc. All Rights Reserved. *  
* *  
* Uploading example II *  
* *  
* - R. Stein 12/6/93 *  
* *  
*****/  
  
#include "gcomm.h"  
#include "majorbbs.h"  
#include "filexfcr.h"  
#include "galupx2.h"  
  
STATIC int uplinp(void);  
STATIC void uplfil(char *filnam, char *protoc);  
STATIC int fupupl(int fupcod);  
STATIC void uplfin(void);  
  
int uplstt; /* Enhanced uploader module state number*/  
static FILE *uplmb; /* file pointer for GALUPX2.MCV */  
char *upldir; /* UPLDIR upload directory */  
long uplbmax; /* max bytes allowed in UPLDIR */  
long uplfbmax; /* max files allowed in UPLDIR */  
  
struct module uplmodule={ /* module interface block */  
    "", /* name used to refer to this module */  
    NULL, /* user logon supplemental routine */  
    uplinp, /* input routine if selected */  
    dfsth, /* status-input routine if selected */  
    NULL, /* "injoth" routine for this module */  
    NULL, /* user logoff supplemental routine */  
    NULL, /* hangup (lost carrier) routine */  
    NULL, /* midnight cleanup routine */  
    NULL, /* delete-account routine */  
    uplfin /* finish-up (sys shutdown) routine */  
};  
  
void EXPORT  
init_uploader(void) /* Uploader initialization */  
{  
    stzcpy(uplmodule.descrp, gndnam("GALUPX2.MDF"), MNMSIZ);  
    uplstt=register_module(&uplmodule);  
    uplmb=opnmsg("GALUPX2.MCV");  
    mkdir(spr("%s.", upldir=stgopt(UPLDIR)));  
    uplbmax=lngopt(UPLBMAX, 0, 2147483647L);  
    uplfbmax=numopt(UPLFBMAX, 0, 32767);  
    dclvda(8+1+3+1);  
}  
  
STATIC int  
uplinp(void) /* Uploader input handler */  
{  
    setmbk(uplmb);  
    if (margc == 1 && sameas(margv[0], "X")) {  
        return(0);  
    }  
    do {  
        bghnc();  
        switch (usrptr->subst) {  
            case 0:  
                cncchr();  
                prfmsg(usrptr->subst=UPLNAME);  
                break;  
            case UPLNAME:  
                if (usrptr->flags&INJOIP) {  
                    prfmsg(UPLNAME);  
                    break;  
                }  
                cncall();  
                parsinc();  
        }  
    }  
}
```

```

        switch (margc) {
        case 0:
            uplfil("", "?");
            break;
        case 1:
            uplfil(margv[0], "?");
            break;
        default:
            uplfil(margv[0], margv[1]);
        }
        break;
    }
} while (!endcnc());
outprf(usrnum);
return(1);
}

STATIC void
uplfil( /* upload file(s) */
char *filnam, /* file name, or "" for multi-file */
char *protoc) /* protocol, or "?" for list */
{
    if (sameas(filnam, "")) {
        filnam="";
    }
    if (rsvnam(filnam)
        || strchr(filnam, ':') != NULL
        || strchr(filnam, '\\') != NULL
        || strstr(filnam, "..") != NULL) {
        prfmsg(UPLRSV);
        prfmsg(UPLNAME);
    }
    else {
        stzcpy(vdaptr, filnam, 8+1+3+1);
        fileup(filnam, protoc, fupupl);
    }
}

int
fupupl( /* Upload handling routine */
int fupcod)
{
    int rc=0;

    setmbk(uplmb);
    switch(fupcod) {
    case FUPBEG: /* Begin upload, check permission, reserve */
        if (vdaptr[0] == '\\0' && rsvnam(ftfscb->fname)) {
            strcpy(ftfbuf, "File name is a reserved DOS device name.");
            break;
        }
        cntdir(spr("%s*.%", upldir));
        if (numfils >= uplmax) {
            strcpy(ftfbuf, "Upload directory is full.");
            break;
        }
        ftfscb->maxbyt=uplbmax-numbyts;
        sprintf(ftfbuf, "%s%s", upldir, vdaptr[0] == '\\0' ? ftfscb->fname
            : vdaptr);
        rc=1;
        break;
    case FUPEND: /* End complete upload of a file, unreserve */
        sprintf(ftfbuf, "%s%s", upldir, vdaptr[0] == '\\0' ? ftfscb->fname
            : vdaptr);
        break;
    case FUPSKP: /* Skip incomplete upload of a file */
        unlink(spr("%s%s", upldir, vdaptr[0] == '\\0' ? ftfscb->fname
            : vdaptr));
        break;
    case FUPFIN: /* Finish file upload session */
        usrptr->state=uplstt;
        if (ftfscb->actfil >= 1) {
            prfmsg(UPLTHX);
        }
        break; /* rc == 0, so we exit to parent menu page */
    }
    return(rc);
}

```

```

STATIC void
uplfin(void)                               /* Finalize uploading example */
(
    c!smg(up!mb);
)

```

Here are the CNF options for upload example #2:

```

LEVEL4 ()

This is the directory where the files will go.
Be sure to specify a proper path PREFIX (e.g.
ending with a backslash, or whatever)

UPDIR (UPDIR\) S 0 Upload directory:

This is the maximum number of files allowed in the
upload directory.

UPLMAX (Maximum files allowed in upload directory: 1000) N 0 32767

This is the maximum number of bytes (the total of all
files) allowed in the upload directory.

UPLBMAX (Maximum bytes allowed in upload directory: 1000000) L 0 2147483647

LEVEL6 ()

UPLNAME (<ESC>[0;1;36m
Name of file to upload (or "" for multiple files): ) T Upload example II file name

UPLRSV (<ESC>[0;1;35m
That's a reserved or invalid DOS file name, please choose another name.
) T Upload example II file name collides with device list

UPLFUL (<ESC>[0;1;35m
The upload directory is full.
) T Upload example II too many files

UPLTHX (<ESC>[0;1;32m
Thanks for uploading.
) T Upload example II finished

```

This code, plus support files, is available for download on the Galacticcomm Demo system in the file GALUPX2.ZIP. (Note: <ESC> represents the ASCII escape code '\x1B'.) Here's what the module would look like online:

```

TOP (TOP)
Make your selection (T,I,F,E,L,A,P,R,D,O,W,U,N,S,? for help, or X to exit): U

Name of file to upload (or "" for multiple files): COLDEMO.EXE

To start uploading COLDEMO.EXE, type:

A ... ASCII                      B ... YMODEM Batch
M ... XMODEM-Checksum             G ... YMODEM-g
C ... XMODEM-CRC                 Z ... ZMODEM
1 ... XMODEM-1K                  K ... Kermit / Super Kermit

(Add '!' to automatically log off when done)

Your choice (or 'X' to exit): Z

(Hit Ctrl-X a few times to abort)
Beginning ZMODEM upload of the file COLDEMO.EXE
**B0100000023be50

      (uploading takes place)

***  UPLOAD COMPLETE  ***

Thanks for uploading.

TOP (TOP)
Make your selection (T,I,F,E,L,A,P,R,D,O,W,U,N,S,? for help, or X to exit): _

```

Assuming that the menu selection to invoke the uploading service is "U", the user can enter any of the following concatenated commands from that menu:

U	Enter the upload service
U <filename>	Upload a specific file, prompt for protocol
U <filename> <protocol>	Upload a file using a protocol
U *	Upload multiple files, prompt for protocol
U * <protocol>	Upload multiple files using a protocol

struct module uplmodule

In this example, the module structure is fleshed out with a helpful comment for each field. An uplfin() routine has been added to clean up before shutdown.

init_uploader()

The initialization routine does the same work as does the one in the first example, plus it also supports a Sysop-configurable directory for uploads, and reads in Sysop-configurable byte and file limits. The dclvda() call formally declares the need for enough space in the VDA to store a file name.

uplinp()

Some immediately obvious renovations are the checking for "X" to exit, and the use of command concatenation routines (bgncnc(), cncxxx(), endcnc()). When first entering this module, the cncchr() call helps with concatenated commands the user could have entered from the parent menu page. The user is prompted to enter a name for the uploaded file.

The UPLNAME substate handles the reaction to the upload file name prompt. The user can respond by hitting "*" or RETURN to signify that multiple files will be uploaded. But the first special case we handle is when the INJOIP flag is set, meaning we need to reprompt after a "/p" page message or other unexpected event. After that, parsin() reparses the input into separate margv[] words (see about how bgncnc() unparses on page 75). These are passed to an internal function, uplfil(), with default values for protocol and file name as appropriate.

uplfil()

This function translates "*" for filename into the "" that the first parameter of fileup() needs to signify multiple file uploads. It then proceeds to check the file name for dangerous characters like ":", "\", and ".". There are many other (less dangerous) illegal characters for file names, but most of these are caught eventually when FTF tries to create the file. If the file name is safe, it's stored in the VDA and fileup() is called. Notice that the file name is checked for size limitations before writing to the VDA. Avoiding buffer overruns is a wonderful habit to get into, although we're not exactly at high risk here.

fupupl()

In the FUPBEG exit point, we recheck for reserved names. This is necessary for multi-file uploads when we don't know the names until this point. We also need to check file size and quantity limits since these too are dynamic. This may be a little impolite to the user to bring up this file quantity limitation so late in the game, but it must be checked for each file in a multi-file upload, so it needs to be in FUPBEG anyway.

Even this check is not completely air-tight: if two users started uploading 5-Megabyte files at the same time, and the UPLBMAX setting is 6-meg, they will probably both be allowed to complete their uploads. That's because the directory contents are measured only at the beginning of each upload, without taking into account uploads that are already underway.

uplfin()

This routine politely closes the GALUPX2.MCV file when the BBS shuts down.

ASCII Downloads

To dump an ASCII file to the user's terminal, you can use:

<code>listing(path,whndun);</code>	list an ASCII text file to the user's terminal
<code>char *path;</code>	DOS path of the file (must be a permanent storage location)
<code>void (*whndun)(all);</code>	restore state & substate, prompt the user for what to do next
<code>int all;</code>	1=all of file was output, 0=aborted

The "path" parameter must point to a location where the file's full path specification will reside throughout the listing. For example, a region of the volatile data area, a private malloc()'d region, a literal file name, etc. Do not use spr(), a portion of input[], an automatic (stack) buffer, or any other location where the contents will change before the (*whndun)() routine gets called.

The listing() routine will usurp the channel's state and substate (usrptr->state and usrptr->substt). It's up to the (*whndun)() routine to restore your state (return value from register_module()) and substate. The (*whndun)() routine gets passed a single parameter which is 1=all of the file was downloaded, or 0=file was aborted by the user.

The listing() function will not be able to operate if the user has tagged too many files for download (see about ftgnew() on page 121).

An example initiation of an ASCII download:

```
listing("E:\DOC93\SATNAV.HLP",lstback);
```

An example (*whndun)() routine:

```
void
lstback(int all)
{
    usrptr->state=snstate;
    prfmsg(usrptr->substt=all ? FULLPMT : SHORTPMT);
}
```

Note that the (*whndun)() routine does not need to call outprf(). (If it does call outprf() for any reason, it should then call clrprf() to avoid double prompting.)

Downloads

Assuming that you've already taken care of all interactive aspects of your application, then here are the steps to take to add file downloading capability:

1. In your source code, include the following special-purpose header file:

```
#include "filexfer.h"
```

2. Define your own "tagspec" data structure. This can be up to 17 bytes of data for storing information on your file, in any format you choose. A tagspec may refer to a single file or to multiple files (for example you could store "FILE.TXT" or "*.TXT"). To allow your files to be tagged for later download, you'll need to store enough information in this 17-byte structure to later reconstruct the file's DOS path, and any security and accounting information. We'll talk more about tagspec's below.
3. Code your own download handler routine. This routine will be called by the file transfer service to perform application-specific tasks throughout the download session. This is usually where your most work is, and it will be discussed in detail below.
4. Call ftgnew() to reserve an entry in the tag table.

```
navail=ftgnew();           Reserve space in the tag table
int navail;                Number of spaces available

struct ftg *ftgptr;        tag table entry
```

Each user has his own row of entries in the 2D tag table, the length of each row being specified by the offline Configuration option MAXTAGS. All downloads, whether explicitly tagged or not, are handled via an entry in the tag table. If ftgnew() returns 0, there is no room. If it returns nonzero, then that's the number of spaces available, and ftgptr will point to your spot in the user's tag table.

5. Now fill in the tag table entry. Store your tagspec in `ftgptr->tagspc`, a 17-byte character array. Again, you know the format of what's stored here, the file transfer service doesn't care. Set `ftgptr->flags` according to the flags: `FTGWLD` (multi-file), and `FTGABL` (whether possible to tag or not). And set `ftgptr->tshndl` to point to your tagspec handler routine.
6. Call the `ftgsbm(prot)` routine to submit the tagspec.

```

usrup=ftgsbm(prot);      Submit the tagspec for download
int usurp;              1=FTF has usurped control of session
                        0=you still have control of session
char *prot;             protocol code

```

Download Protocol Codes	
single-file for immediate download:	"M" "C" "1" "V"
single-file or multi-file for immediate download:	"L" "A" "B" "G" "Z" "ZR" "K"
to log off after downloading:	append "!" to any of the above
tag for later download:	"T"
tag (quietly) for later download:	"TQ"
for compressed file viewing:	"V"
menu of download protocols:	"?" or ""

You can use the "TQ" protocol internally to tag a file without notifying the user. It's otherwise an invalid protocol though: users are not able to specify it.

To validate a download protocol code, you can use `valdpc()`:

```

ok=valdpc(prot);        Is this a valid download protocol?
int ok;                 1=valid, 0=invalid
char *prot;            protocol code string

```

The return value of `ftgsbm()` tells you whether or not FTF has taken control of your session.

`ftgsbm()` returns 0 in these cases:

```

ftgsbm(anything), after ftgnew() has returned 0, outputs a warning
ftgsbm("T") tags a file for download & notifies the user
ftgsbm("TQ") silently tags a file for download
ftgsbm(protocol) when your TSHVIS routine reports that your file is
invisible (in this case, ftgsbm() calls your TSHFIN exit-point)

```

`ftgsbm()` returns 1, and changes `usrptr->state,substt` in these cases:

```

ftgsbm("?") changes state/substt to prompt for protocol/options
ftgsbm(protocol) changes state/substt to proceed with download
ftgsbm(trash) rebuffs, and then does the same thing as ftgsbm("?")

```

Here's what you can count on:

If `ftgsbm()` returns 1, then it has changed the `usrptr->state`, and either `TSHFIN` or `TSHHUP` will get invoked exactly once eventually. If `ftgsbm()` returns 0, then `usrptr->state` and `usrptr->substt` have either not been changed, or already restored by the `TSHFIN` exit point of your download handler routine.

Tagspecs

A tagspec is a 17-byte application-specific structure for keeping track of each file downloaded. Its main purpose is to allow file tagging, where a user identifies a file for download at some later time. But all files that are downloaded use tagspecs, even if they aren't explicitly tagged.

These 17-byte tagspecs were designed as small as possible so that users could have room to tag numerous files without wasting large amounts of memory. There's just enough room for a 4-byte pointer and a 12-character file name plus its NUL terminator. The 4-byte pointer could be used to refer to some directory, Forum, Library, category, or other structure somehow. For example, you could store a 32-bit absolute database pointer here. You can use any format you need for the 17 bytes, as long as you keep in mind the asynchronous nature of file tagging (identifying the file now, downloading it later).

You must take special care that your application can handle file tagging before you set the `FTGABL` (tagable) flag in `ftgptr->flags` (see step 5 above). For one example of a disaster waiting to happen, suppose you store the 12-character file name in the tagspec, and a 60-character path prefix in your Volatile Data Area. In some of your download handler exit points (`TSHVIS`, `TSHBEG` and possibly `TSHSCN` and others) you assemble these two things together to get the DOS path for the file. This will work just dandy if the user never tags these files.

But if you set `ftgptr->flags|=FTGABL` and the user picks "T" to tag a file in your module, you're probably in for big trouble. For one thing, if he tags two files from different directories, and then downloads them both, they will both use the path prefix meant for the second file. For another, the user may be off in some other module, with entirely different data stored in the Volatile Data Area, when he gets around to downloading his tagged files. Then when your download handler routine gets called, and goes to the Volatile Data Area for that 60-character prefix, something rather unexpected may be there in its place. This is the worst kind of bug to have on your hands -- intermittent cause and unpredictable effect.

Possible corrections to this kind of bug are either to find somewhere else to store the path prefix, or to rethink the strategy. You could store the path prefix in a database and store a database pointer in the tagspec. Or perhaps you could restrict your application to using the same path prefix for all files. If you were desperate to give users the ability to tag up to `MAXTAGS` number of files, each with an arbitrary path prefix, then your application could allocate a monster 3D array, 61 bytes by `MAXTAGS` by `nterms`, and store all the path prefixes there.

Download Handler Routine

This routine is a collection of "exit points" for all the application-specific tasks that need to be done during the general-purpose downloading session. See page 108 for a discussion of the concept of exit points as regards the upload handler routine.

This pseudo-code template roughly outlines the tasks expected at each of the exit points.

```
int
tshxxx(                               /* Handle the application-specific */
int tshcod)                            /* aspects of your downloads      */
{                                       /* (tshcod=code for each aspect) */

    int rc=0;

    setmbk(whatever your application uses);
    (be sure to set any other appropriate globals)
    switch(tshcod) {
    case TSHDSC:                        /* Describe the file(s) in English */
        sprintf(tshmsg,"app-specific description of file",ftgptr->tagspc);
        break;
    case TSHVIS:                        /* Visible to this user?          */
        if (file exists, or user is allowed to know it doesn't) {
            open & read first TSHLEN bytes into tshmsg, as in:
            if ((fp=fopen("<DOS path for the file>",FOPRB)) != NULL) {
                fread(tshmsg,1,TSHLEN,fp);
                rc=1;
            }
        }
        break;
    case TSHSCN:                        /* Break down multiple filespec   */
        if (there's at least one file in this multi-file tagspec) {
            store tagspec for the individual file in tshmsg
            rc=1;
        }
        break;
    case TSHNXT:                        /* Next file in multi-file spec   */
        if (there are more subfiles) {
            store tagspec for the individual file in tshmsg
            rc=1;
        }
        break;
    case TSHBEG:                        /* Begin downloading this file   */
        if (file can't be downloaded by this user) {
            sprintf(tshmsg,"You can't download the file because...");
        }
        else {
            reserve it
            sprintf(tshmsg,"<DOS path for file>",ftgptr->tagspc);
            strcpy(fftscb->fname,"<file name for the protocol>");
            rc=1;
        }
        break;
    case TSHEND:                        /* File download was successful   */
        unreserve it
        record a completed download
        break;
    case TSHSKP:                        /* This file download aborted     */
        unreserve it
        record an aborted download
        break;
    case TSHFIN:                        /* End of downloading session    */
        usrptr->state=your state
        usrptr->substt=your substate
        prompt (you don't need to call outprf here)
        rc=1;
        break;
    case TSHHUP:                        /* Channel hanging up            */
        the TSHFIN exit point never got called, clean up as req'd
        break;
    }
    return(rc);
}
```

This code is available on the Galacticomm Demo System, (305) 583-7808. You may want to download it and use it as a template to write your own download handler routine.

The global variables `usrnum`, `usrptr`, `usaptr`, and `vdaptr` are available for all exit points of the download handler routine. In addition, these FTF variables are available:

<code>struct ftfsbc *ftfsbc;</code>	Session Control Block (see FTF.H) for the current file transfer session
<code>struct ftfsp *ftfsp;</code>	protocol specifications (see FTF.H) for the current file transfer session
<code>struct ftg *ftgptr;</code>	current tag table entry
<code>ftgptr->tagspc</code>	current tagspec
<code>char *tshmsg;</code>	multi-purpose buffer (context-dependent)

If you need any other global variables, be sure to set them up in your routine. The meaning of your routine's return value depends on the type of exit point (which is coded in `tshcod`). These will be discussed individually for each exit point. In some cases, no return value is expected. You should return 0 in each of those cases to allow for future expansion.

TSHDSC - Describe the file(s) in English

Format a description for the single file or multiple files in the `tshmsg` buffer. You should word the description so that it looks right when following the word "the", as in "Do you want to download the %s (y/n)?" (see offline Text Block `SRETRYV`).

The `tagspec` you originally submitted is in `ftgptr->tagspc`. This exit-point must work with a multi-file `tagspec`, as well as the single-file `tagspec`'s that you'll be breaking it down into. The return value doesn't matter, but it's a good practice to return 0 for future expansion.

TSHVIS - Visible to this user?

Is this file visible? Return 1=visible to user, or 0=not visible.

This exit-point should handle multi-file `tagspec`s (which are not now checked for visibility, but may be in future versions) as well as single-file `tagspec`s. Return 1 if the user is allowed to know whether this file exists.

The purpose of this exit point is to allow you to decide to totally restrict access to a given file, to the point where certain users don't even know it exists. It also allows you to break-down multi-file `tagspec`s in `TSHSCN` and `TSHNXT` without doing any security checks.

If the file is visible, read the first `TSHLEN` bytes (80 bytes) into the `tshmsg` buffer. This will allow tests for a compressed file, such as a .ZIP file, to know whether or not to present the user with the "V" protocol choice. If you don't read in the first `TSHLEN` bytes of this file, then the "V" protocol will never be available.

TSHSCN - Break down multiple filespec

If you submit a multi-file tagspec (by setting `ftgptr->flags|=FTGWLD`), then you'll be asked to break it down into single-file tagspecs when the time comes for downloading.

If the multi-file tagspec refers to one or more files, then return 1 and store the tagspec for the first file in the `tshmsg` buffer. Return 0 if the multi-file tagspec ends up referring to no files at all. You can also return 2 to indicate that there may be files available, but that you aren't supplying a tagspec in `tshmsg` yet -- you'll do that in `TSHNXT`.

TSHNXT - Next file in multi-file spec

This continues the work that `TSHSCN` started. Return 1 if there are more single-file tagspecs, and store the next one in `tshmsg`. Return 0 if there are no more files. Return 2 if there may be more files, but you want to be called again to make a tagspec out of them. This may help simplify certain multi-file breaking-down schemes.

See the example module source code starting on page 131 for a way to do `TSHSCN` and `TSHNXT` with `findlst()` and `findnxt()` (see page 183 about the routines themselves). For your convenience, `ftuptr->fb` is a user-specific "fndblk" structure that you can use with `findlst()` and `findnxt()` in this context.

TSHBEG - Begin downloading this file

Verify that the user is allowed to download this file. If not, put a reason in `tshmsg` (a complete sentence, as in "You don't have access rights to that file.") and return 0.

Here are some things you might check for:

- o Does the file name have the proper syntax?
- o Does the file name conflict with a reserved name?
- o Does the user have permission to download this file?
- o Are too many users opening files at once (thereby using up all file handles)?
- o Can this user afford the download charges, if any?
- o Is another user currently modifying/uploading/deleting this file?

If downloading is OK, store the DOS path for the file in `tshmsg` and return 1. In case this protocol can communicate file names, put the file name into `ftfscb->fname`. For example, if downloading with `ZMODEM`, this file name will be the one used on the user's terminal. It doesn't have to be the same as the one used on the BBS, but it usually is.

There is another special option in the `TSHBEG` exit point. You can return -1 to indicate "the file is not yet available for download, but it will be later". The file won't be downloaded during this download session, but it will remain tagged for download, and can be downloaded later. This is used to accommodate files that are not instantly available, such as those on a multi-disk CD ROM drive (as opposed to those that are, such as on your hard disk).

TSHEND - File download was successful

You may wish to record the download, or charge the user for it at this point. Use this exit point to cancel whatever reserving was done in the TSHBEG entry point. For example, if you had some scheme for preventing the Sysop from deleting this file while this user was downloading it, now's the time to recognize that it's OK to delete the file. The return value doesn't matter, but it's a good practice to return 0 for future expansion.

You can count on the fact that for every TSHBEG call there will be exactly one TSHEND or TSHSKP call. The only exception would be a very abrupt termination like a power-loss.

TSHSKP - This file download aborted

Here you'll need to do the same "unreserving" that is done in the TSHEND exit point. You may also wish to make some record of the aborted download.

TSHFIN - End of downloading session

This is the most important exit point -- the one where FTF turns control back over to your application. Be sure to restore your `usrptr->state` and `usrptr->substt`, and prompt the user for what comes next.

If your application supports file tagging, you should recognize that TSHFIN only means that the file transfer service is done controlling this user's session for the moment. The user may still cause your download handler to get called for this same tagspec later, with any of the cases except TSHFIN and TSHHUP. This could happen for example when the user tries to log off, and is given a chance to download all files that he has tagged.

In `ftfscb->actfil` you'll find a count of the total number of files successfully downloaded. In `ftfscb->tryfil` is the total files that we tried to download. Of course it's always the case that `actfil <= tryfil`. When `actfil < tryfil`, not all the files made it. The user has already seen a report about this, including why the last transfer aborted, or why the last of possibly several files were skipped.

Since you're taking back control of this channel, it's up to you to set things straight for what's up next for this user. Here are two alternatives:

You want control back

- o Restore your `usrptr->state`
- o Restore your `usrptr->substt`
- o Prompt the user (you don't need to call `outprf()`)
- o Return 1

You want to return to the parent menu

- o Restore your `usrptr->state`
- o Say bye to the user if you wish (you don't need to call `outprf()`)
- o Return 0

Either way, if you're prompting or saying goodbye using `prfmsg()`, you need to be sure to set your message block pointer using `setmbk()`.

The pseudo-code for TSHFIN handling on page 124 assumes you want to take control back. Here's an alternative of the TSHFIN exit point to allow you to exit to your module's parent menu page:

```
case TSHFIN:                                /* End of downloading session */
    usrptr->state=your state
    prompt (exiting);                        /* (don't call outprf()) */
    rc=0;
    break;
```

This may be appropriate if your module really doesn't want to regain control of the channel when the download is done. In this case you only need to restore the user's state code (module number), not the substate. Your module never actually regains control of the user's channel. You can prompt him with some parting words, but you don't need to. For consistency you should do whatever you normally do when the user exits from your module to the parent menu.

In the case where you're supposedly retaking control, you may have to call `condex()`, and possibly wind up returning control to your parent Menu Tree menu afterward. This would be the case if you were about to return to your module's own internal main menu, and you found out you had gotten where you are through command concatenation. See page 78 about this whole `condex()` business.

Another handy feature of the file transfer service is that each `ftgsbm()` invocation that returns 1 is followed (eventually) by exactly one TSHFIN or TSHHUP exit point invocation, except in dire cases (power loss for example).

TSHHUP - Channel hanging up

Use this exit point to clean up your affairs in case the user hangs up or the channel disconnects while the user is in the download session. This only occurs in place of a TSHFIN, and will probably not be called if one of your files is tagged for download and the actual download is interrupted by a disconnect. In that case, TSHFIN had long since been called, after the user tagged the file.

If a disconnect occurs in the middle of downloading a file the user didn't tag for download (he asked to download it immediately), then a TSHSKP exit point will be called first, to properly terminate the download, before TSHHUP is called.

See the discussion on the upload handler exit point FUPHUP on page 113 about making sure your module's cleanup code executes exactly once for users in your module.

Downloading Example #1

```
/******  
*  
* GALDNX.C  
* Copyright (C) 1994 GALACTICOMM, Inc. All Rights Reserved.  
* Downloading example.  
*  
* - R. Stein 12/6/93  
*  
*****/  
  
#include "gcomm.h"  
#include "majorbbs.h"  
#include "filexfer.h"  
  
STATIC int dnlinp(void);  
STATIC int tshdnl(int tshcod);  
  
int dnlstt; /* Downloading module state number */  
struct module dnlmodule=(NULL,NULL,dnlinp,dfsthn,NULL,NULL,NULL,NULL,NULL);  
  
void EXPORT  
init_downloader(void) /* Downloader initialization */  
{  
    stzcpy(dnlmodule.descrp,gmdnam("GALDNX.MDF"),MNMS1Z);  
    dnlstt=register_module(&dnlmodule);  
}  
  
STATIC int  
dnlinp(void) /* Downloader input handler */  
{  
    int rc=1;  
  
    switch (usrptr->substt) {  
    case 0:  
        prf("Name of file to download: ");  
        usrptr->substt=1;  
        break;  
    case 1:  
        if (margc == 0) {  
            rc=0;  
        }  
        else if (ftgnew() == 0) {  
            ftgsbm(""); /* use warning feature of ftgsbm() */  
            rc=0;  
        }  
        else {  
            stzcpy(ftgptr->tagspc,margv[0],TSENG);  
            ftgptr->tshndl=tshdnl;  
            ftgsbm(" ");  
        }  
    }  
    outprf(usrnum);  
    return(rc);  
}  
  
int  
tshdnl(  
int tshcod) /* Handle the application-specific */  
/* aspects of your downloads */  
/* (tshcod=code for each aspect) */  
{  
    int rc=0;  
  
    switch(tshcod) {  
    case TSHVIS: /* Visible to this user? */  
        rc=1;  
        break;  
    case TSHDSC: /* Describe the file(s) in English */  
        sprintf(tshmsg,"file %s",ftgptr->tagspc);  
        break;  
    case TSHBEG: /* Begin downloading this file */  
        sprintf(tshmsg,"DNLDIR\\%s",ftgptr->tagspc);  
        strcpy(ftfscb->fname,ftgptr->tagspc);  
        rc=1;  
        break;  
    case TSHFIN: /* End of downloading session */  
        usrptr->state=dnlstt;
```

```
    }  
    return(rc);  
}
```

This simple module allows users to download files from the DNLDIR subdirectory of the BBS. Users of this service type in a file name (that they must know somehow), and choose a protocol to download it.

init_downloader()

This routine registers the downloading module using the description in the GALDNX.MDF file.

dnlinp()

This routine handles text line input from the user after he selects this download service. Upon entry, the user is prompted for the file name. If he types RETURN, he's returned to the parent menu page without any downloading. If there is no room to download any more files because the user's tag table is completely full, then the user is notified (ftgsbm() does this) and he's also returned to the parent menu.

Otherwise, the tagspec structure is simply the file name, and it's copied to the tagspec in the current tag table entry that ftgnew() identified for us. Neither the FTGWLD nor FTGABL flags are set, indicating that these will be single-file non-tagtable downloads. The call to ftgsbm() turns control over to the file transfer service and a list of protocols is presented to the user.

tshdnl()

The TSHVIS exit point declares that all files are visible if they exist. The description for the file will be "file <filename>". In the TSHBEG exit point the path and protocol names are passed to FTF. There are no TSHEND, TSHSKP, or TSHHUP exit points coded, because they only need to return 0 and do nothing else. TSHFIN restores the usrptr->state and returns 0, telling FTF to return control to the parent menu.

Potential Improvements to Download Example #1

Here are some of the features left out of this brief example that you should consider if you are using downloads in your application:

- o Viewing the contents of compressed files, such as .ZIP files
- o Multiple-file downloads
- o Checking for conflicts between the file name and DOS devices
- o Checking for attempts to download files like DNLLIB\..\BBS.BAT
- o File tagging
- o Concatenated commands
- o Allowing "X" to exit
- o Telling the user what files are available in the directory
- o Sysop-configurable prompts an .MSG file
- o Sysop-configurable download directory
- o Full module structure in source code, with comments

All of these features are included in download example #2.

Downloading Example #2

```

/*****
 *
 *   GALDNX2.C
 *
 *   Copyright (C) 1994 GALACTICOMM, Inc.   All Rights Reserved.
 *
 *   Downloading example.
 *
 *
 *                               - R. Stein 12/6/93
 *
 *****/

#include "gcomm.h"
#include "majorbbs.h"
#include "filexfer.h"
#include "galndx2.h"

STATIC int dnlinp(void);
STATIC int fllist(void);
STATIC int tshdnl(int tshcod);
STATIC void dnlfin(void);

int dnlstt;                               /* Downloading module state number */
static FILE *dnlmb;                       /* file pointer for GALDNX2.MCV */
char *dnlmdir;                            /* DNLDIR download directory */

struct module dnlmodule=(                 /* module interface block */
    "",                                    /* name used to refer to this module */
    NULL,                                  /* user logon supplemental routine */
    dnlinp,                                 /* input routine if selected */
    dfsthn,                                 /* status-input routine if selected */
    NULL,                                   /* "injoth" routine for this module */
    NULL,                                  /* user logoff supplemental routine */
    NULL,                                  /* hangup (lost carrier) routine */
    NULL,                                  /* midnight cleanup routine */
    NULL,                                  /* delete-account routine */
    dnlfin                                 /* finish-up (sys shutdown) routine */
);

void EXPORT
init__downloader(void)                   /* Downloader initialization */
{
    stzcpy(dnlmodule.descrp, gndnam("GALDNX2.MDF"), MNMSIZ);
    dnlstt=register_module(&dnlmodule);
    dnlmb=opnmsg("GALDNX2.MCV");
    dnlmdir=stgopt(DNLDIR);
}

STATIC int
dnlinp(void)                             /* Downloader input handler */
{
    int rc=1;

    setmbk(dnlmb);
    if (margc == 1 && sameas(margv[0], "X")) {
        return(0);
    }
    do {
        bgncnc();
        switch (usrptr->subst) {
            case 0:
                cncchr();
                if (!fllist()) {
                    cncall();
                    rc=0;
                }
                prfmsg(usrptr->subst=FLNAME);
                break;
        }
    }
}

```

```

case FLNAME:
    cncal();
    parsin();
    if (margc == 0 || sameas(margv[0], "?")) {
        fllist();
        prfmsg(FLNAME);
    }
    else if (rsvnam(margv[0])
        || strchr(margv[0], ':') != NULL
        || strchr(margv[0], '\\') != NULL
        || strstr(margv[0], "..") != NULL) {
        prfmsg(FLRSV);
        prfmsg(FLNAME);
    }
    else if (ftgnew() == 0) {
        ftgsbm(""); /* use ftgsbm() to say out-of-tags */
        rc=0;
    }
    else {
        stzcpy(ftgptr->tagspc, margv[0], TSENG);
        ftgptr->tshndl=tshndl;
        ftgptr->flags=FTGABL;
        if (strchr(margv[0], '?') != NULL
            || strchr(margv[0], '*') != NULL) {
            ftgptr->flags|=FTGWLD;
        }
        rc=ftgsbm(margc > 1 ? margv[1] : "");
    }
}
) while (!endcnc());
outprf(usnum);
return(rc);
}

STATIC int
fllist(void) /* Display listing of files */
{
    struct fndblk fb;

    if (!fnd1st(&fb, spr("%s*.*", dnldir), 0)) {
        prfmsg(FLNONE);
        return(0);
    }
    prfmsg(FLHEAD);
    do {
        prfmsg(FLLINE, fb.name, l2as(fb.size), ncdte(fb.date), nctime(fb.time));
    } while (fndxt(&fb));
    return(1);
}

int
tshndl(
int tshcod) /* Handle the application-specific */
/* aspects of your downloads */
/* (tshcod=code for each aspect) */
{
    int rc=0;
    FILE *fp;

    setmbk(dnlmb);
    switch(tshcod) {
case TSHDSC: /* Describe the file(s) in English */
        sprintf(tshmsg, "file %s", ftgptr->tagspc);
        break;
case TSHVIS: /* Visible to this user? */
        if (ftgptr->flags&FTGWLD) {
            rc=fnd1st(&ftuptr->fb, ftgptr->tagspc, 0);
            break;
        }
        if ((fp=fopen(spr("%s%s", dnldir, ftgptr->tagspc), FOPRB)) != NULL) {
            fread(tshmsg, 1, TSHLEN, fp);
            rc=1;
        }
        break;
case TSHSCN: /* Break down multiple filespec */
        if (fnd1st(&ftuptr->fb, spr("%s%s", dnldir, ftgptr->tagspc), 0)) {
            strcpy(tshmsg, ftuptr->fb.name);
            rc=1;
        }
        break;
}
}

```

```

    case TSHNXT:                /* Next file in multi-file spec */
        if (fndnxt(&ftuptr->fb) {
            strcpy(tshmsg,ftuptr->fb.name);
            rc=1;
        }
        break;
    case TSHBEG:                /* Begin downloading this file */
        sprintf(tshmsg,"%s%s",dnldir,ftgptr->tagspc);
        strcpy(ftfscb->fname,ftgptr->tagspc);
        rc=1;
        break;
    case TSHFIN:                /* End of downloading session */
        usrptr->state=dnlsttt;
    }
    return(rc);
}

STATIC void
dnlfm(void)                    /* Finalize downloading example */
{
    ctsmsg(dnlmb);
}

```

Here are the CNF options for download example #2:

LEVEL4 ()

This is the directory for the files to download.
Be sure to specify a proper path PREFIX (e.g.
ending with a backslash, or whatever)

DNLDIR (DNLDIR) S 0 Download directory:

LEVEL6 ()

FLHEAD (<ESC>[0;1;32m
Files available:

) T Download example II file listing header

FLLINE (<ESC>[0;1;36m%-12.12s<ESC>[33m %10s %8s %-5.5s
) T Download example II file listing line

FLNONE (<ESC>[0;1;35m
No files are available for download.

) T Download example II no files available

FLNAME (<ESC>[0;1;36m
Name of file(s) to download:) T Download example II file name prompt

FLRSV (<ESC>[0;1;35m
That's not a valid file name
) T Download example II file name invalid

This code, plus support files, is available for download on the Galacticomm Demo system in the file GALDNX2.ZIP. (Note: <ESC> represents the ASCII escape code '\x1B'.)

Here's what the module would look like online:

```
TOP (TOP)
Make your selection (T,I,F,E,L,A,P,R,D,O,W,U,N,S,? for help, or X to exit): D

Files available:

LOADER.BAT          1524 11/26/93 20:00
DPATCH.ZIP         57001 12/07/93 12:26
GALCONDL.ZIP       4162 12/07/93 16:05
DINSTALL.BAT      237 12/02/93 13:31

Name of file(s) to download: *.ZIP

L ... Listing (a screen at a time)      Z ... ZMODEM
A ... ASCII (continuous dump)          ZR... ZMODEM (resume after abort)
B ... YMODEM Batch                     K ... Kermit / Super Kermit
G ... YMODEM-g
T ... Tag file(s) for later download

(Add '!' to automatically log off when done)

Choose a download option (or 'X' to exit): Z

(Hit Ctrl-X a few times to abort)
Beginning ZMODEM download of the file *.BAT
rZ
**B00000000000000

      (downloading takes place)

*** DOWNLOAD COMPLETE ***

TOP (TOP)
Make your selection (T,I,F,E,L,A,P,R,D,O,W,U,N,S,? for help, or X to exit): _
```

init_downloader()

This function registers the GALDNX2 module, opens the GALDNX2.MCV file, and reads in the Sysop-configured download directory prefix.

dnlinp()

This text-line input handling function returns to the parent menu if the user enters "X" and otherwise uses command concatenation routines to parse user input. When a user enters this module he's given a list of the files available (more on fllist() below), and asked to type in the name of a file, or file specification with wildcards, to download.

If he replies with "?" he's given the list of files again. If the name he gives is reserved, like "CON.TXT", then he is warned and reprompted for a file name.

An entry in the tag table is obtained, if available, by ftgnew(). In this example, the use of wildcard characters '?' or '*' in the file name signifies a multi-file download, and the FTGWLD flag is set. The FTGABL flag is always set. If the user concatenated a protocol code after his file specification, that is passed to ftgsbm(). Otherwise, a list of protocols is requested. FTF will handle invalid protocol codes with a warning and a list of the available codes.

fllist()

This function scans through the files in the download directory and lists them on the user's terminal, with file size, date, and time.

The size of the output buffer (offline Hardware Setup option OUTBSZ) effectively limits the number of files that can be put online for download. With OUTBSZ set to 4096, you can probably handle somewhere around 100 files. Much more than that, and the output buffer will overflow.

tshdnl()

Here the description is the same simple "file <filename>". Multi-file tagspecs are "visible" if any matching files exist. Single-file tagspecs are visible if the file can be opened, and if so, the first TSHLEN bytes are read in so the "v" protocol can be supported.

The TSHSCN and TSHNXT exit points use the classic technique for breaking down multi-file tagspecs into single files: `fndlst()` and `fndnxt()`. In each case a single-file tagspec (the file name) is copied into `tshmsg`.

There are no security restrictions on the files in the download directory. In the TSHBEG exit point, the path is constructed in `tshmsg`, and the protocol name is copied to `ftfscb->fname`.

The TSHFIN exit point just restores the state code and returns 0, indicating that control should return to the download module's parent menu page when downloading is complete.

File Transfer Protocol

To define a file transfer protocol for The Major BBS:

1. Define a file transfer protocol specification structure. Here is an example of the structure for XMODEM-CRC downloads.

```
struct ftfpsp ftpxcx=(                                /* XMODEM-CRC transmitting */
    NULL,                                             /* 1-3 code letters for protocol */
    "C",                                             /* name of protocol */
    "XMODEM-CRC",                                    /* name of protocol */
    FTFXMT+FTFXTD,                                   /* protocol capability flags */
    sizeof(struct xymdat),                            /* total length of session control block */
    3*16,      /* .byttmo          default byte timeout */
    10*16,     /* .paktmo          default packet timeout */
    10,        /* .retrys          default max retries */
    1L,        /* .window      max window size (packets/bytes as appropriate) */
    128,       /* .paksiz       packet size 0=auto-figure */
    yxini,     /* .initze()      Initialize this protocol (recompute scblen) */
    xcxsrt,    /* .start()       Start a transfer */
    yxctn,     /* .contin()      Continuously call, 1=more, 0=done */
    yxinc,     /* .hdlinc()      Handle one incoming byte */
    NULL,      /* .hdlins()      Handle incoming line of text */
    ftfabt,    /* .term()        Initiate graceful termination of transfer */
    ftfxca,    /* .abort()       Immediately unconditionally abort the transfer */
    ftfincb,   /* .hdlinb()      Handle an array of incoming bytes */
    "",        /* .secur         App-specific security of some kind */
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0});
```

Here are the protocol capability flags:

```
/*-- File Transfer Protocol Capability and Characteristic Flags --*/
#define FTFXMT 0x01      /* Transmit protocol (versus receive) */
#define FTFASC 0x02      /* ASCII-session (versus binary-session) */
                        /* where '\r' is the sole line terminator */
                        /* note: ASCII sessions do not detect errors */
#define FTFMUL 0x04      /* capable of multiple files? */
#define FTF7BT 0x08      /* capable of using 7-bit data path? */
#define FTFASF 0x10      /* ASCII-file (versus binary-file) */
                        /* where '\n' is the sole line terminator */
#define FTFAFN 0x40      /* abort is final, don't ask "try again?" */
#define FTFXTD 0x80      /* extended ftfpsp structure (flags2 & hdlinb) */
```

See FTF.H for more details. See FTFXYMD.C for the full implementation of XMODEM. See FTF*.C for examples of the implementation of other file transfer protocols. The 'F' file import/export file transfer "protocol" is implemented in FILEXFER.C with lots of cheating.

2. Register the structure in your `init_XXX()` routine using `ftplog()`:

```
ftplog(fpnr);
```

where `fpnr` is a pointer to the structure from step 1. The new protocol will appear in the appropriate lists with all the other protocols.

6. OPERATOR INTERFACE

Video Output (printf())

Every DOS program that has any need for speed must write directly to the video screen memory. Our method is to replace the standard library version of the "printf()" function with our own. We also provide windowing, cursor positioning, colors, invisible screen-image updating (so that, for example, we can update both the main console and two channel emulation screens simultaneously), and a few other miscellaneous functions. All of these routines are available in GCOMM.LIB.

<code>printf(ctlstg,p1,p2,...,pn);</code>	substitute for standard printf()
<code>char *ctlstg;</code>	control string, functions supported are %s,%c,%d,%u,%x, all of them with field len, zero or blank fill, left/right justification options.
<code>TYPE p1,p2,...,pn;</code>	just like printf/cprintf parms. (note: no "longs" or "floats")

There is no limit to the number of parameters (p1,p2,...,pn) than you may pass to printf(). They should correspond one-for-one with the "%" directives in the control string. See page 141 for a description of the ANSI graphics capability of printf().

<code>setatr(attrib);</code>	sets video attributes
<code>int attrib;</code>	attribute code: sum of...
	0x80 = blink foreground
	0x40 = red background
	0x20 = green background
	0x10 = blue background
	0x08 = bright foreground
	0x04 = red foreground
	0x02 = green foreground
	0x01 = blue foreground

Another way to compute the attribute is to add together three numbers, one from each of these columns:

<u>Foreground Attribute</u>	+	<u>Background Attribute</u>	+	<u>Blink</u>
0x00 Black		0x00 Black		0x00 non-blinking
0x01 Dark blue		0x10 Dark blue		0x80 blinking
0x02 Green		0x20 Green		foreground
0x03 Cyan (blue-green)		0x30 Cyan (blue-green)		
0x04 Red		0x40 Red		
0x05 Magenta (purple)		0x50 Magenta (purple)		
0x06 Brown		0x60 Brown		
0x07 Grey		0x70 Grey		
0x08 Dark Grey				
0x09 Bright blue				
0x0A Bright green				
0x0B Bright cyan				
0x0C Bright red				
0x0D Bright magenta (pink)				
0x0E Bright yellow				
0x0F Bright white				

This function affects subsequent `printf()`'s. You can change the background color of the entire screen, for example to magenta, by coding:

```
setatr(0x5E);
printf("\f");
```

Then all subsequent `printf()`'s will show bright yellow on magenta. The `creol()` function also sets the background color for the remainder of the line according to the latest `setatr()` attribute.

See page 142 for converting IBM screen attribute codes into ANSI color coding sequences.

See page 164 for what `setatr()` does on monochrome screens.

```
setwin(scn,xul,yul,xlr,ylr,sen); set window parameters
char *scn;                          seg:off start addr of screen
                                      (if NULL, default display)
int xul;                             upper left x coord
int yul;                             upper left y coord
int xlr;                             lower right x coord
int ylr;                             lower right y coord
int sen;                             scroll enable (1=yes)

rstwin();                            restore previous window parameters
```

The `setwin()` function defines a window on the screen. All coordinates are inclusive (they are inside the window). The "scn" parameter can be used to direct all subsequent screen output to a SCNSIZ-byte buffer (SCNSIZ is 4000) instead of to the visible video memory. Make "scn" NULL for the default condition of writing directly to video memory. The "sen" parameter, when 1, means that when you `printf()` a newline ('\n') on the last line of the window, the entire window gets scrolled up one line (and the bottom line is filled with the current `setatr()` background attribute). When "sen" is 0, then a newline has the same effect as a carriage return ('\r') on the bottom line of the window. The `rstwin()` function undoes the effect of the most recent `setwin()` call.

scnstt=frzseg();	get video ram base address
char *scnstt;	seg:off start addr of screen
unfrez();	release video ram address (in a multitasking environment).

The frzseg() function returns a pointer to the physical video memory (or in some multitasking environments, to the "hidden" screen).

The unfrez() function releases the memory indicated by frzseg() (that is, the "scnstt" return value of frzseg() should not be used after calling unfrez()). This is only required in certain multitasking environments that permit writing directly to screen memory. Even so, since printf() calls both frzseg() and unfrez() (when the first parameter in the most recent call to setwin() was NULL), you may not need to call unfrez() at all -- just wait until the next printf().

To blank out a SCNSIZ-byte screen buffer area:

sdblank(buffer,attr);	Blank the screen buffer memory
char *buffer;	SCNSIZ-byte buffer (4000 bytes)
char attr;	Attribute to use

2000 spaces with the specified attribute are written to the buffer.

scnstt=auxcrt();	get auxiliary CRT address
char *scnstt;	pointer to auxiliary screen address
locate(x,y);	move cursor to x,y
int x;	dest x (0=left-most column)
int y;	dest y (0=top line)
rstloc();	restore previous cursor location
x=curcurx();	get current cursor x coordinate
int x;	
y=curcury();	get current cursor y coordinate
int y;	

The locate() function moves the video cursor to a new location. Even if setwin() has directed video output to an invisible buffer, the visible cursor may still move to track the locate() function (see cursact() on page 140). The rstloc() function undoes the effect of the most recent locate() call. All cursor positions are relative to the upper left corner of the display buffer (not to the upper left corner of the setwin() window).

clreol();	clear to end of line (in window)
-----------	----------------------------------

This function clears from the cursor position to the right margin of the current window, setting the background attribute for this line segment to the current setatr() attribute.

```

printfat(x,y,ctlstg,pl,...,pn);  combination of locate() and printf()
                                into one routine (saves code space)
int x,y;                          screen coordinates
char *ctlstg;                      control string
pl,...,pn                          parameters (max 8 bytes)

```

This routine is just like printf(), except that the control string is preceded by a screen position. The parameters (pl,...,pn) can be no more than 8 bytes.

The following routines are like printfat(), except you can set an origin and specify offsets:

```

proff(xbase,ybase);                set origin for prat() locations
int xbase,ybase;

prat(x,y,ctlstg,pl,...,pn);        combination of locate() and printf()
int x,y;                          screen coordinates (relative to
char *ctlstg;                      the most recent proff() setting)
pl,...,pn                          control string
                                parameters (max 12 bytes)

```

The explode() family of routines (page 162) calls proff() to set the upper left corner of the exploded region. Then the choose() family (page 166) and edtval() (page 164) routines use prat() so that their coordinates are relative to the upper left corner of the exploded region.

```

cursact(movit);                    enable moving of blinking cursor
int movit;                          1=move blinking cursor, 0=still

```

This routine selects whether the locate() routine will move the actual visible cursor or not. Whatever you pass to cursact(), locate() will still select the location written to by printf(), etc. But with cursact(0), the blinking cursor will remain stationary. cursact(1) is the default condition.

```

belper(pitch);                    beep the operator console
int pitch;                          0=silent 200-1000 high-low pitch

```

This routine defines the handling of printf() when sending an ASCII BEL character to the operator console. This will probably be a much shorter beep than DOS uses. For example:

```

belper(200);
printf("\7"); /* high-pitched beep */
belper(1000);
printf("\7"); /* low-pitched beep */
belper(150);
printf("\7"); /* very high-pitched beep */
belper(0);
printf("\7"); /* silent */

```

The belper() routine is used in The Major BBS to set the signup notification (SGNBEL), the page-sysop warning (SOPBEL), and the emulation screen beep (EMUBEL).

```

ansion(on);                        enable/disable ANSI graphics
int on;                             1=process ANSI graphics sequences
                                0=ignore ANSI graphics sequences
                                (display as literal)

```

This function turns on or off the interpretation by printf() of ANSI graphics characters embedded in the text stream. The following ANSI commands are supported when ansion(1) (the non-default condition) is in effect:

<ESC> [<row> ; <column> H	Move cursor to <row>,<column>
<ESC> [<row> ; <column> f	Move cursor to <row>,<column>
<ESC> [<nrows> A	Move cursor up <nrows> rows
<ESC> [<nrows> B	Move cursor down <nrows> rows
<ESC> [<ncols> C	Move cursor forward <ncols> columns
<ESC> [<ncols> D	Move cursor backward <ncols> columns
<ESC> [s	Save cursor position
<ESC> [u	Restore cursor position
<ESC> [2 J	Erase display
<ESC> [K	Erase to the end of the current line
<ESC> [0 m	Normal display attribute
<ESC> [1 m	Bold display attribute
<ESC> [4 m	Underscore display attribute
<ESC> [5 m	Blink display attribute
<ESC> [7 m	Reverse display attribute
<ESC> [8 m	Invisible display attribute
<ESC> [3 0 m	Black foreground
<ESC> [3 1 m	Red foreground
<ESC> [3 2 m	Green foreground
<ESC> [3 3 m	Yellow foreground
<ESC> [3 4 m	Blue foreground
<ESC> [3 5 m	Magenta foreground
<ESC> [3 6 m	Cyan foreground
<ESC> [3 7 m	White foreground
<ESC> [4 0 m	Black background
<ESC> [4 1 m	Red background
<ESC> [4 2 m	Green background
<ESC> [4 3 m	Yellow background
<ESC> [4 4 m	Blue background
<ESC> [4 5 m	Magenta background
<ESC> [4 6 m	Cyan background
<ESC> [4 7 m	White background

Notes

None of the above commands include any spaces.

<ESC> the ASCII escape code '\x1B'.

<row> one or two ASCII digits representing the screen row, between 1 and 25. Defaults to 1 if omitted.

<column> one or two ASCII digits representing the screen column, between 1 and 80. Defaults to 1 if omitted.

<nrows> one or two ASCII digits representing the number of screen rows, between 1 and 25. Defaults to 1 if omitted.

<ncols> one or two ASCII digits representing the number of screen columns, between 1 and 80. Defaults to 1 if omitted.

The ";" may be omitted if the <column> parameter is omitted.

The "m" commands (display attribute) may be combined using semicolons. For example:

```
<ESC> [ 1 m <ESC> [ 3 3 m <ESC> [ 4 5 m
```

has the same effect as:

```
<ESC> [ 1 ; 3 3 ; 4 5 m
```

Both of these set the display attribute to bright yellow on magenta. You could use the following code to display a message with these settings in the current display window:

```
printf("\33[1;33;45mWafer yield for the 128 MHz 80586: 95%");
```

The individual characters of the above commands may be split across different calls to `printf()`. There may even be intervening calls to `printf()` as long as the intervening calls have `ansion(0)` (ANSI graphics disabled). Also, the display attribute is preserved across calls to `setatr()` when `ansion(0)`. All of these features enable the emulation of a single user's channel with ANSI graphics while The Major BBS simultaneously updates various other information on the console. For more on how The Major BBS emulates multiple screens at once, see page 143.

Note: the move cursor command is relative to the upper left corner defined in the most recent call to `setwin()` (not to the upper left corner of the screen buffer, as is the `locate()` function).

To convert IBM display attributes into ANSI sequences, use `ibm2ans()`:

```
bufptr=ibm2ans(attr,buffer);    Convert IBM attribute to ANSI colors
char *bufptr;                  copy of buffer
char attr;                     attribute (see page 138)
char *buffer;                  where to put ANSI sequence (up to
                               15 bytes, including terminator)
```

The following coded example shows some of these video routines in action. The `zipred()` function makes a red box with an exclamation point zip across the screen from left to right, and then disappear. Note that the image (of the original screen with a red box on it) is constructed in a buffer and then copied to the visible screen, so the red box does not "flicker".

```

void
zipred(void)
{
    static char savebf[4000]; /* buffer to save original screen image */
    static char drawbf[4000]; /* buffer to use as a drawing board */
    char *frzseg();          /* frzseg() returns a char pointer! */
    int savx,savy;           /* to save cursor position */
    int x;

    savx=curcurx();          /* save cursor position */
    savy=curcury();
    movmem(frzseg(),savebf,4000); /* save screen image */
    setatr(0x4F);            /* bright white on red */
    for (x=0 ; x < 70 ; x++) { /* From left to right... */
        movmem(savebf,drawbf,4000); /* prepare drawing board */
        setwin(drawbf,x,9,x+10,15,0); /* define an 11 by 7 window */
        printf("\f");           /* fill it with red */
        locate(x+5,12);        /* and in the center */
        printf("!");           /* put an "!" */
        rstwin();              /* (restore window settings) */
        movmem(drawbf,frzseg(),4000); /* make this picture visible */
    }
    movmem(savebf,frzseg(),4000); /* restore original screen image */
    locate(savx,savy);         /* restore cursor position */
    setatr(0x07);             /* white on black */
    unfrez();
}

```

To read from the video screen or buffer, you can use `scnoff()`:

```

offset=scnoff(x,y);          Compute the offset
int offset;
int x,y;

```

For example, to find the character and attribute at the lower right corner of a SCNSIZ-byte screen buffer:

```

lrchar=scnbuf[scnoff(79,24)];
lrattr=scnbuf[scnoff(79,24)+1];

```

Writing to Several ANSI Screens at Once

To support ANSI capability on several screens at once, you must save some internal variables. The `printf()` routine only supports one screen and can keep track of partial ANSI commands. To support several screens, you must save the entire `curatr` structure. (Note that `curatr` is the name for a structure type as well as the name of an instance of that structure type.)

For example, you could maintain an array of curatr structures and make one of them "active" whenever you wrote to one of the screens.

```
struct curatr ansave[NUMSCNS];
:
movmem(&ansave[actscn],&curatr,sizeof(struct curatr));
printf(ANSI commands to screen);
movmem(&curatr,&ansave[actscn],sizeof(struct curatr));
```

The first movmem() puts the curatr structure for the active screen where printf() will use it and update it. The second movmem() saves it away again.

(Note: curatr.attrib is the attribute setting of the most recent setmbk.)

Keyboard Input (getchc())

```
yes=kbhit();                Has the operator hit a key?
int yes;                    1=yes 0=no
```

After checking the standard library routine kbhit(), The Major BBS uses this routine to input a single keystroke.

```
char=getchc();              Get a keystroke from the keyboard
int char;
```

Note that getchc() returns a 16-bit value. GCOMM.H contains numerous constants for the return value of getchc(). The values are either extended ASCII in the lower 8 bits, or the keyboard scan code in the upper 8 bits.

Here are appropriate constants for representing the return values of getchc() (you can use these in the C-language cases of a switch statement):

' ' through '~' (for the printable ASCII characters)

'\x00' through '\xFF' (for all extended ASCII characters)

F1	SHIFT+F1	CTRL+F1	ALT+F1
F2	SHIFT+F2	CTRL+F2	ALT+F2
F3	SHIFT+F3	CTRL+F3	ALT+F3
F4	SHIFT+F4	CTRL+F4	ALT+F4
F5	SHIFT+F5	CTRL+F5	ALT+F5
F6	SHIFT+F6	CTRL+F6	ALT+F6
F7	SHIFT+F7	CTRL+F7	ALT+F7
F8	SHIFT+F8	CTRL+F8	ALT+F8
F9	SHIFT+F9	CTRL+F9	ALT+F9
F10	SHIFT+F10	CTRL+F10	ALT+F10

HOME	CTRLHOME	BAKTAB
END	CTRLEND	INS
PGUP	CTRLPGUP	DEL
PGDN	CTRLPGDN	TAB
CRSRLF	CTRLLF	ESC
CRSRRT	CTRLRT	
CRSRUP	CTRLUP	
CRSRDN	CTRLDN	

ALT_A	ALT_K	ALT_U	ALT_0
ALT_B	ALT_L	ALT_V	ALT_1
ALT_C	ALT_M	ALT_W	ALT_2
ALT_D	ALT_N	ALT_X	ALT_3
ALT_E	ALT_O	ALT_Y	ALT_4
ALT_F	ALT_P	ALT_Z	ALT_5
ALT_G	ALT_Q		ALT_6
ALT_H	ALT_R		ALT_7
ALT_I	ALT_S		ALT_8
ALT_J	ALT_T		ALT_9

The codes that these constants represent are used in many contexts, online and offline. See AIN.H for converting incoming ANSI sequences into these keystroke codes.

Cursor

To control the video screen cursor:

<code> cursiz(howbig);</code>	Set the size of the video cursor
<code> int howbig;</code>	NOCURS cursor disappears
	LILCURS small standard cursor
	BIGCURS big insert-mode cursor
 <code> rstcur();</code>	 Restore the cursor to the size
	it was before the last <code>cursiz()</code>
 <code> howbig=curcurs();</code>	 Find out how big the current cursor
	is.
<code> int howbig;</code>	NOCURS, LILCURS, or BIGCURS

7. OPERATOR SERVICES

Statistics

You can add your own statistical graphs to those already available on the operator console. There are two parts to this:

1. Generating the graph
2. Displaying the graph

The first part is up to you. You could create a 4000 byte file that stores the exact display image of the statistics screen. Only a 42 by 17 character portion of that screen may be used for your graph:

Statistics Sub-Screen

columns 15 through 56, inclusive, out of 0 through 79
rows 1 through 17, inclusive, out of 0 through 24

You'll create this file offline, perhaps during the nightly auto-cleanup (that's when we create the standard screens used in DFTSTATS.C).

Or you could just create a "background" file. At the moment when the Sysop brings up your screen, you can have special code that fills in all the figures or draws some kind of drawings.

The second part is up to the Sysop to do, and you to prepare for. Use the function `register_stascn()` to register your statistics screen. Then your screen will be available on the scrolling menu of the Statistics and Graphs screen when the BBS is on the air.

So, to register your screen:

1. Create one copy of the `stasc` data structure in your online code. This structure is defined in `STATSCNS.H`. Here is an example, with the blanks filled in:

```
struct stasc mygraph=(          /* statistic screen interface structure */
    "Sat activity",            /* name of statistic screen */
    "DDDDSTA1.BIN",          /* file name to get screen from */
    NULL,                     /* initialize (bring up scn) routine */
    NULL,                     /* key hit handler routine */
    NULL,                     /* occasional update (every 60 secs) */
    NULL,                     /* once-per-cycle routine */
    NULL,                     /* take down screen routine */
    NULL);
```

The "name" appears in the menu of choices on the Statistics and Graphs screen. The 42 by 17 region of the file is displayed first (as background, or as your finished display) when the Sysop calls up your screen. Note that the statistics screen file has the Developer-ID prefix on it. All the NULLS are the non-implementation of five special purpose routines for your screen.

Here are what these routines can do:

<code>void (*inirou)();</code>	A routine to call whenever your screen should appear. The routine could make computations and display them on your screen. Of course, you should only write to the 42 by 17 character area reserved for your screen as shown above.
<code>unsigned (*keyhit) (unsigned scncod);</code>	This routine is called with each and every keystroke from the Sysop when your screen is on display. The parameter of the routine is the same as the <code>getchc()</code> return value (page 144). The routine should either handle the keystroke and return 0, or just return the keystroke value if it doesn't know what to do with it.
<code>void (*occrou)();</code>	This routine will get called every 60 seconds that your screen is on display. You can update your display with the up-to-the-minute information.
<code>void (*cycrou)();</code>	This routine gets called about 16 times a second when your screen is on display. If you use it (put something other than NULL here), be sure that it executes fast, so the system doesn't get bogged down updating your display.
<code>void (*finrou)();</code>	This routine gets called when your display goes away. It gets called once for each call to the <code>inirou()</code> routine.

2. Register the statistics screen in your initialization routine.

```
register_stascn(&mygraph);
```

This all swings into place when the Sysop switches to the statistics screen (by typing `<Alt-T>`) and selecting your statistics screen from the list of choices.

Audit Trail

To display a message in the audit trail:

```
shocst(tex1,tex2,p1,...,pN);   Enter a string into the Audit Trail
char *tex1;                   summary string (up to 32 chars long)
char *tex2;                   detail string, as in printf()
p1,...,pN                     parameters for control string
                               The detail string can be up to 65
                               characters long. The parameters
                               passed can take up to 12 bytes on
                               the stack.
```

This function makes an an entry in the audit trail. Just the date, time and summary information appear on the Summary screen. The summary, detail and source information, along with time and date, appear on the Audit Trail Detail screen, and get written to the Audit Trail database.

Sources for Audit Trail messages

```
Cleanup
Event N (1 to 4)
Console
Chan NN (00 to FF)
```

The global variable "usrnum" is an implicit input to shocst(). It must be set to a valid user number (0 to nterms-1) or -1 to -3 (see page 39).

Channel Status Reporting

```
shochl(legend,sing,attr)     Show a line on the Online User Info
                               screen
                               information, up to 29 characters
char *legend;                single-character indicator for
char sing;                   the user matrix (Summary screen too)
                               IBM color display attribute
int attr;
```

If your Add-on Option does not manage sessions or connections in some way, you probably don't want to use this routine. The convention is that the User-ID appears on the Online User Information screen and a double-arrow appears in the user matrix there and on the Summary screen.

One of the conventions of the Online User Information screen is to color-code the information based on the user's baud rate. You can do this by computing the last parameter of shochl() using baudat() (as we do many times in MAJORBBS.C).

```
attr=baudat(baud,blink)     Compute the display attribute based
                               on the user's baud rate
                               IBM 8-bit display attribute
int attr;                   baud rate, 300 to 38400
unsigned baud;              1=give us a blinking attribute
int blink;
```

8. DATABASES

Database Functions (xxxbtv())

Btrieve, by Novell, Inc., provides a powerful collection of database-management primitives. The Major BBS has a plethora of routines that provide a smooth C language interface.

Btrieve File Identifiers

The functions `opnbtv()`, `setbtv()`, and `clsbtv()` are the only functions that explicitly deal with a single database. All other database functions implicitly deal with a single database using the file identified by the most recent `setbtv()`. A Btrieve file identifier is a pointer to a structure defined in `BTVSTF.H`:

```
struct btvblk {                                /* btrieve file data block def */
    long posblk[128/4];                        /* position block */
    char *filnam;                               /* file name */
    int reclen;                                /* record length */
    char *key;                                  /* key for searching, etc. */
    char *data;                                 /* actual record contents */
    int lastkn;                                 /* last key number used */
    int keylns[SEGMAX];                        /* lengths of all possible keys*/
};

#define BTVFILE struct btvblk                 /* shorthand for btrieve file id */
```

`opnbtv()` is the source of all Btrieve file identifiers. `setbtv()` is used to set the Btrieve file for all subsequent database functions.

WARNING: You must be careful not to forget to use `setbtv()`. If you do, your program might seem to work fine when you test it with a single user, but fail insidiously when you try it with multiple users.

Many of the database functions have an explicit "recptr" parameter for specifying where to get or put a record for writing or reading. If you use NULL for this value then you may use a default buffer specified in the `btvblk` "data" field.

There are three flavors of database record read procedures. All specify a record according to a "key" or according to the order by a key.

`get` Read the record. If missing, bomb (with "catastro" message)

query	Find out if the record is in the database
acquire	Find out if the record is in the database, and if so, read it

Here are synopses of the routines in PLBTVSTF.C:

omdbtv(mode);	set mode for next opnbtv() call
int mode;	see codes below

This routine sets the Btrieve file mode for subsequent database files opened by opnbtv(). The following mutually exclusive values for the "mode" parameter are defined in BTVSTF.H:

PRIMBV	default, pre-image (data integrity) mode
ACCLBV	accelerated (faster write) mode
RONLBV	read-only mode
VERFBV	write-with-verify mode
EXCLBV	exclusive (non-sharing) mode

bbptr=opnbtv(filnam,reclen);	open a Btrieve file for I/O
BTVFILE *bbptr;	file identifier
char *filnam;	filespec
int reclen;	record length in bytes

If the file is not found, a "catastro()" error message (BTRIEVE OPEN ERROR 12) is generated automatically by opnbtv() -- you never have to check the return value for error conditions.

setbtv(bbptr);	set BTVFILE ptr for subsequent ops
BTVFILE *bbptr;	file to be used hereafter

This important utility specifies the Btrieve database for all other -- btv utility functions (except opnbtv() and clsbtv()). See about Btrieve File Identifiers, above.

rstbtv();	restore the current BTVFILE to what it was before the corresponding recent setbtv()
------------------	---

Calls to setbtv()/rstbtv() make use of a "stack" so they can be nested up to 10 levels deep.

is=qrybtv(key,keynum,qryopt);	query whether a record exists
int is;	1 if record exists, else 0
char *key;	key to be used for lookup
int keynum;	key position number to use
int qryopt;	search option (used via macro)

getbtv(recptr,key,keynum,getopt);	get a record (bomb if not there)
char *recptr;	destination record buffer ptr (NULL to use bbptr->data)
char *key;	key to be used for lookup
int keynum;	key position number to use
int getopt;	search option (used via macro)

```

is=obtbv(recptr,key,keynum,obtopt); acquire a record (if you can)
int is;                               1 if record exists, else 0
char *recptr;                          destination record buffer ptr
                                       (NULL to use bbptr->data)
char *key;                              key to be used for lookup
int keynum;                             key position number to use
int optopt;                             search option (used via macro)

```

The above three routines are almost exclusively called out in the source code only by using macros that are defined in BTVSTF.H. For example, all the q-btv() "functions" are actually macros that generate special-purpose calls to qrybtv().

```

abspos=absbtv();                        find current "absolute" position
long abspos;                            "absolute" (direct) file position

gabtv(recptr,abspos,keynum);           get a record by "absolute" position
char *recptr;                          destination record buffer ptr
                                       (NULL to use bbptr->data)
long abspos;                            "absolute" (direct) file position
int keynum;                             key number to establish there

is=aabtv(recptr,abspos,keynum);        acquire a record by "absolute" position
int is;                                  1 if record could be read, else 0
char *recptr;                          destination record buffer ptr
                                       (NULL to use bbptr->data)
long abspos;                            "absolute" (direct) file position
int keynum;                             key number to establish there

```

The return value of absbtv() may be used to identify the "physical" position of a record in a database. The record may be accessed using gabtv() or aabtv() with that position. This type of access is much faster than any of the keyed access methods. We have determined that this absolute position value is never zero for a legitimate record. Therefore, we sometimes use 0L as a special value to represent a pointer to no record at all.

```

is=slobtv(recptr);                     Read the physically first record in
int is;                                the database
                                       1=there was one 0=database empty

is=snxbtv(recptr);                     Read the physically next record in
int is;                                the database
                                       1=there was one 0=already read last

is=sprbtv(recptr);                     Read the physically previous record
int is;                                in the database
                                       1=there was one 0=already read first

is=shibtv(recptr);                     Read the physically last record in
int is;                                the database
                                       1=there was one 0=database empty

```

These routines search the database in the physical order in which records are stored. The sequence defined by the database keys usually doesn't matter in this case, and neither does chronology -- records could appear in

any order. The advantage of the `snxbtv()/sprbtv()` routines over the `qnxbtv()/qprbtv()` routines (which are keyed-sequential -- see page 153) is their speed: physical access can be about eight times as fast as keyed-sequential.

Database Update Routines

<code>updbtv(recptr);</code> <code>char *recptr;</code>	update current record replacement record buffer ptr (NULL to use <code>bbptr->data</code>)
<code>ok=dupdbtv(recptr);</code> <code>int ok;</code> <code>char *recptr;</code>	(more tolerant) update current record 1=updated 0=duplicate collision replacement record buffer ptr

These functions must be called immediately following a get-record call of some kind (queries are not enough, but `gcrbtv()` will suffice after a query). `updbtv()` and `dupdbtv()` are the same except that when the new record contents produce an illegally duplicate key, `updbtv()` bombs with a `catastro()` error, while `dupdbtv()` simply returns a 0. These routines cannot be called on a database with variable length records. Instead, use `upvbtv()`:

<code>upvbtv(recptr,length);</code> <code>char *recptr;</code> <code>int length;</code>	update variable length record replacement record buffer ptr (NULL to use <code>bbptr->data</code>) number of bytes for new record contents
---	---

Database Insert Routines

<code>insbtv(recptr);</code> <code>char *recptr;</code>	insert new fixed-length record new record buffer ptr (NULL to use <code>bbptr->data</code>)
<code>ok=dinsbtv(recptr);</code> <code>int ok;</code> <code>char *recptr;</code>	(more tolerant) insert new record 1=inserted 0=duplicate collision new record buffer ptr (NULL to use <code>bbptr->data</code>)

`insbtv()` will automatically generate a fatal error (BTRIEVE INSERT ERROR 5) if you try to insert a record with the same key as another record in a database (if that key does not allow duplicates). `dinsbtv()` will simply return a 0 in that case. Otherwise, the routines have identical effects.

<code>invbtv(recptr,length);</code> <code>char *recptr;</code> <code>int length;</code>	insert variable length record new record buffer ptr (NULL to use <code>bbptr->data</code>) number of bytes for new record
---	--

Deleting a Database Record

<code>delbtv();</code>	delete current record
------------------------	-----------------------

This function must be called immediately following a get-record call of some kind (queries are not enough, but `gcrbtv()` will suffice after a query).

Variable Record Length -- Just how long was it?

```
reclen=llnbtv();           find the record length of the most
                           recently read record
```

This function is handy after reading a variable length record to find out how many bytes are actually in the record. In that case, this is the same value that was passed to `invbtv()` or `upvbtv()` as the length parameter when the record was put into the database.

Closing a Database File

```
clsbtv(bbptr);             close a Btrieve file when finished
BTVFILE *bbptr;           file identifier (from opnbtv())
```

Database Query Routines

The following database utilities are implemented as macros (defined in `BTVSTF.H`). They actually generate calls to functions `qrybtv()`, `getbtv()`, and others.

```
is=qegbtv(key,keynum);    query for "equal to" spec'd key
int is;                   1 if record exists, else 0
char *key;                key specification
int keynum;               key number involved

is=qnxbtv();              query for "next" record in seq
int is;                   1 if record exists, else 0

is=qprbtv();              query for "previous" record
int is;                   1 if record exists, else 0

exists=qgtbtv(key,keynum); query for "greater than" key
int exists;               1 if record exists, else 0
char *key;                key specification
int keynum;               key number involved

exists=qgebtv(key,keynum); query for "greater/eq (>=)" key
int exists;               1 if record exists, else 0
char *key;                key specification
int keynum;               key number involved

exists=qltbtv(key,keynum); query for "less than" key
int exists;               1 if record exists, else 0
char *key;                key specification
int keynum;               key number involved

exists=qlebtv(key,keynum); query for "less/equal (<=)" key
int exists;               1 if record exists, else 0
char *key;                key specification
int keynum;               key number involved

exists=qlobtv(keynum);    query for lowest record present
int exists;               1 if record exists, else 0
int keynum;               key number involved
```

exists=qhibtv(keynum);	query for highest record present
int exists;	1 if record exists, else 0
int keynum;	key number involved

The above query routines set the "key" buffer reserved for the database. For example, the following code might be used to find out if there are any users in the Registry database whose User-ID starts with the letter "Q" (see REGISTRY.C for the variables and structure of this database -- the "regrec" structure).

```

setbtv(regbb);
if (qgebtv("Q",0) && regbb->key[0] == 'Q') {
    prf("Warning! Someone named \"%s\" is in the registry!",regbb->key);
}

```

Database Get Routines

geqbtv(recp,key,keynum);	get record "equal to" spec'd key
char *recp;	destination record buffer ptr
char *key;	key specification
int keynum;	key number involved
gnxbtv(recp);	get "next" record in sequence
char *recp;	destination record buffer ptr
gprbtv(recp);	get "previous" record in seq
char *recp;	destination record buffer ptr
ggtbtv(recp,key,keynum);	get first record > spec'd key
char *recp;	destination record buffer ptr
char *key;	key specification
int keynum;	key number involved
ggebtv(recp,key,keynum);	get first record >= spec'd key
char *recp;	destination record buffer ptr
char *key;	key specification
int keynum;	key number involved
glbtv(recp,key,keynum);	get highest record < spec'd key
char *recp;	destination record buffer ptr
char *key;	key specification
int keynum;	key number involved
glebtv(recp,key,keynum);	get highest record <= spec'd key
char *recp;	destination record buffer ptr
char *key;	key specification
int keynum;	key number involved
globtv(recp,keynum);	get lowest record present
char *recp;	destination record buffer ptr
int keynum;	key number involved
ghibtv(recp,keynum);	get highest record present
char *recp;	destination record buffer ptr
int keynum;	key number involved

```

gcrbtv(recp,keynum);           get (or re-get) "current" record
char *recp;                   destination record buffer ptr
int keynum;                   key number to establish

```

The above "get" routines read in a full record from a database. By contrast, the query routines simply tell you if the record is there, and read in the key fields.

In all of these routines you may specify where to put the data, using the "recp" parameter. You may also pass NULL for this parameter, and the data record will go into the standard data buffer for the database. Expanding upon the query example, gcrbtv() can be used to read in a database record that passed the query test:

```

setbtv(regbb);
if (qgebtv("Q",0) && regbb->key[0] == 'Q') {
    gcrbtv(NULL,0);
    prf("Warning! Someone named \"%s\" is in the registry!",regbb->data);
    prf("\nAnd he has this to say about himself:\n\"%s\"\n",
        ((struct regrec *)regbb->data)->sumlin);
}

```

This technique of "casting" the data buffer to a special purpose structure is usually required to use this buffer, because the data field of the btvblk structure (see page 149) is just a general purpose character pointer -- you must overlay the structure of the actual database record.

Database Acquire Routines

```

is=acqbtv(recptr,key,keynum);  "acquire" record with spec'd key
int is;                        1 if record exists, else 0
char *recptr;                 destination record buffer ptr
char *key;                    key value to search for
int keynum;                   key number

is=agtbvtv(recptr,key,keynum); acquire first record > key
int is;                        1 if record exists, else 0
char *recptr;                 destination record buffer ptr
char *key;                    key specification
int keynum;                   key number involved

is=agebtv(recptr,key,keynum);  acquire first record >= key
int is;                        1 if record exists, else 0
char *recptr;                 destination record buffer ptr
char *key;                    key specification
int keynum;                   key number involved

is=altbtv(recptr,key,keynum);  acquire highest record < key
int is;                        1 if record exists, else 0
char *key;                    key specification
int keynum;                   key number involved

is=alebvtv(recptr,key,keynum); acquire highest record <= key
int is;                        1 if record exists, else 0
char *recptr;                 destination record buffer ptr
char *key;                    key specification
int keynum;                   key number involved

```

```
is=alobtv(recptr,keynum);      acquire lowest record in database
int is;                        1 if record exists, else 0
char *recptr;                 destination record buffer ptr
int keynum;                    key number involved
```

```
is=ahibtv(recptr,keynum);     acquire highest record in database
int is;                        1 if record exists, else 0
char *recptr;                 destination record buffer ptr
int keynum;                    key number involved
```

These routines combine a query and a get into the useful combination where you want to see if a record is in a database, and if it is, to read it. Using these routines we could code:

```
setbtv(regbb);
if (agebtv(NULL,"Q",0) && regbb->data[0] == 'Q') {
    prf("Warning! Someone named \"%s\" is in the registry!",regbb->data);
    prf("\nAnd he has this to say about himself:\n\"%s\"\n",
        ((struct regrec *)regbb->data)->sumlin);
}
```

Here are two special-purpose acquire routines:

```
is=agmbtv(recptr);           "acquire next" record in sequence
int is;                       1 if another record exists, else 0
char *recptr;                 destination record buffer ptr
```

```
is=agpbtv(recptr);          "acquire previous" record in sequence
int is;                       1 if previous record exists, else 0
char *recptr;                 destination record buffer ptr
```

Use these routines only for databases with a single non-unique key that's a NUL-terminated string. Each of these routines returns false if the two records (the "current" one and the "next/previous" one) compare unequal (case-sensitive) when treated as strings.

Creating your own Databases

If you purchase the Btrieve development kit from Novell, you can use the following command to create new databases:

```
BUTIL -CREATE <filename>.VIR <filename>.BCR
```

The .BCR file is an editable text file that you will define that specifies the format of your database. See the Btrieve manual. (Tip: use the "zstring" format for NUL-terminated string fields.) The .VIR is an empty "virgin" form of the database that you'll always keep online. During the installation process, an empty .VIR file is copied to a .DAT file if no .DAT file exists.

System Variables Database

The Major BBS maintains several variables on disk in BBSVBL.DAT. These are available at runtime, are changed as necessary, and are automatically saved back to disk every 300 seconds (default value of SVRATE). The following code from MAJORBBS.H shows the fields of the system variables in sv, sv2 and sv3:

```
extern
struct sysvbl {
    char key[4]; /* system-variable btrieve record layout*/
    char dspopt[6]; /* 4-character dummy key of "key" */
    long calls[8]; /* display options by position number*/
    char lonmsg[MTXSI2]; /* number of calls this month/ baud rt*/
    long dwnlds; /* log-on message in effect */
    long uplds; /* total downloads to date */
    long msgtot; /* total uploads to date */
    unsigned emlopn; /* msg (E-mail/Forums) total to date */
    unsigned sigopn; /* E-Mail open at the moment */
    int hisign; /* Forum messages open at the moment */
    char monmal; /* highest Forum number used to date */
    char savmin; /* Aux. CRT display selector */
    long oldsec[8][24]; /* Minutes to save screen */
    char spare[1300-1230]; /* old sec/grp/hr (now in sv3.secghr)*/
} sv;

extern
struct sysvb2 {
    char ky2[4]; /* second system variable btrieve layout*/
    unsigned matrix[NCOMTY][NAGEBK]; /* 4-character dummy key of "ky2" */
    long oldcrd[8][24]; /* matrix of accts (computer/age)*/
    int nliniu[48]; /* old crd/grp/hr (now in sv3.crdghr)*/
    int lstzer; /* number of lines in use per hlf/hr */
    long x25kps; /* date of last zeroing of stats */
    unsigned x25ps; /* X.25 kilopackets sent or received */
    long x25mbs; /* fractional X.25 kilopackets */
    long x25bs; /* X.25 megabytes sent or received */
    unsigned numact; /* fractional X.25 megabytes */
    unsigned numfem; /* total number of user accounts */
    unsigned numcor; /* number of female users */
    unsigned numans; /* number of corporate users */
    unsigned long paidpst; /* number of ANSI users */
    unsigned long freepst; /* credits paid-for so far */
    long totcalls; /* credits given away free so far */
    int lastmcb; /* total calls-to-date */
    char spare[1300-986]; /* date of last midnight cleanup */
} sv2;

extern
struct sysvb3 {
    char ky3[4]; /* third system variable btrieve layout */
    long secghr[NGROUPS-1][24]; /* 4-character dummy key of "ky3" */
    long crdghr[NGROUPS-1][24]; /* seconds used (channel grp/hr) */
    /* credits consumed (channel grp/hr) */
} sv3;
```

User Account Database

The following code from USRACC.H shows the fields of the user accounting database, BBSUSR.DAT. This same structure is used for the dynamically allocated usracc[] array, which stores the information in memory for users who are online. (Note: that array may be bigger than 64K. Use uacoff() to get information on online users -- see page 82).

```
struct usracc {
    char userid[UIDSIZ]; /* user-id */
    char psword[PSWSIZ]; /* password */
    char usrn timer[NADSIZ]; /* user name */
    char usrad1[NADSIZ]; /* address line 1 (company) */
    char usrad2[NADSIZ]; /* address line 2 */
    char usrad3[NADSIZ]; /* address line 3 */
    char usrad4[NADSIZ]; /* address line 4 */
    char usrpho[PHOSIZ]; /* phone number */
    char systyp; /* system type code */
    char usrprf; /* user preference flags */
    char ansifl; /* ANSI flags */
    char scnwid; /* screen width in columns */
    char scnbrk; /* screen length for page breaks */
    char scnfs; /* screen length for FSE stuff */
    char age; /* user's age */
    char sex; /* user's sex ('M' or 'F') */
    unsigned int credat; /* account creation date */
    unsigned int usedat; /* date of last use of account */
    int csicnt; /* classified-ad counts used so far */
    int flags; /* various saved bit flags */
    int access[AXSSIZ]; /* array of remote sysop access bits */
    long emllim; /* e-mail limit reached so far (new/old bdy) */
    char prmcls[KEYSIZ]; /* class to return user to if necessary */
    char curcls[KEYSIZ]; /* current class of this user */
    long timtdy; /* time user has been online today (in secs) */
    unsigned int daystt; /* days left in this class (if applicable) */
    unsigned int fgvdys; /* days since debt was last "forgiven" */
    long creds; /* credits available or debt (if negative) */
    long totcreds; /* total credits ever posted (paid & free) */
    long totpaid; /* total credits ever posted (paid only) */
    char birthd[DATSIZ]; /* this user's birthday date */
    char spare[USRACCSPPARE]; /* spare space, for graceful upgrades */
};

#define ANSON 1 /* ansifl bit definitions */
#define ANSMAN 2 /* ANSI on=1; off=0 */
/* ANSI manual override (0=auto sensing) */

#define HASMST 1 /* flags bit definitions */
#define UNDAXS 2 /* user has the "MASTER" key for the BBS */
#define SUSPEN 4 /* this account cannot be deleted */
#define DELTAG 8 /* this account is "suspended" */
#define GOINVS 16 /* this account is tagged for deletion */
/* this account is "invisible" upon logon */

/* usrprf bit definitions */
#define PRLFLIN 1 /* always use line editor? yes=1 */
```

User Class Database

This database records information on the user classes. Classes are defined by the Sysop using the Remote Sysop ACCOUNT menu, CLASS command.

```
extern
struct acclass {          /* accounting class structure          */
    char cname[KEYSIZ];  /* class name                          */
    char nxtcls[4][KEYSIZ]; /* class to return to when expires      */
    int limcal;          /* limit per call (-1=no limit)         */
    int limday;          /* limit per day (-1=no limit)          */
    int dftday;          /* default days before expiring (-1=never) */
    long dbtlimt;        /* debt limit (0=none)                  */
    int fgvdlay;         /* wait how many days before "forgiving" */
    int idlday;          /* inactive days before delete (-1=never) */
    int flags;           /* general bit flags                    */
    long seconds;        /* seconds used so far this month       */
    unsigned users;      /* total number of users in this class  */
    char msgsz[2][XMSGSZ]; /* exiting class messages               */
    char spare[2032-2022]; /* spare space - decrease when needed   */
};

/* indexes for nxtcls[] when a user... */
#define DOUTTIM 0        /* is out of time for the day           */
#define DLOAFER 1       /* hasn't logged in for x number of days */
#define DEXPIRE 2       /* has been around x number of days     */
#define DCREDIT 3       /* has/doesn't have credits             */

/* struct acclass bit flag definitions */
#define KCKOFF 1        /* out of time: knock the user offline   */
#define CLSCHG 2       /* out of time: temporarily change class */
#define NOCRED 4        /* expire when: credits < 1             */
#define DBTLMT 8        /* expire when: user reaches a debt limit */
#define HASCRD 16       /* expire when: credits > 0             */
#define DAYEXP 32       /* expire when: x number of days passes  */
#define IDLEXP 64       /* expire when: no log on for x # of days */
#define MONDAY 128      /* forgive: every Monday                 */
#define FSTMTH 256     /* forgive: on the first of each month   */
#define NUMDAY 512     /* forgive: every x number of days      */
#define HITLMT 1024    /* forgive: when they hit their debt limit */
#define REPDBT 2048    /* report debt when forgiven?           */
#define CRDXMT 4096    /* this class exempt from credit charges? */
```

Using Spare Space in Galacticom Databases

If you're developing your own Add-on Option for The Major BBS, you should make your own databases rather than add onto the spare spaces in Galacticom databases. Otherwise you'll run into conflicts with other developers trying to use the same space.

On the other hand, if you're customizing your own BBS, and you want to add fields to a database, there might be a way. Add your fields after the spare[] field.

When customizing the database on your BBS, add fields at the end of the structure -- between the spare[] field and the "}" -- and decrease the spare size accordingly, so the size of the overall structure is unchanged.

In new versions of the BBS, Galacticom will try to add new fields before the spare[] field.

Be conservative and use as few bytes as possible. If you use a lot of bytes in a database, and Galacticom eventually uses them too, you're going to be in for some complex conversion activity to be able to update to a new release of The Major BBS.

Generic User Database

This database, BBSGEN.DAT, may be used by any application to store information about users. To create your own records in BBSGEN.DAT, first define a structure. The first two fields should be User-ID and module name. Say you wanted to store a user's score in a game:

```
struct bgame {          /* generic user data records for my game */
    char userid[UIDSIZ]; /* User-ID */
    char modnam[MNMSIZ]; /* Module Name ("My Game") */
    int score;          /* score */
};
```

This sure beats making a whole separate .DAT file for one measly integer.

To store a record for the current user, with a score of 50, you could code:

```
struct bgame bgbuff;
:
:
strcpy(bgbuff.userid, usaptr->userid);
strcpy(bgbuff.modnam, "My Game");
bgbuff.score=50;
setbtv(genbb);
invbtv(&bgbuff, sizeof(bgbuff));
```

Notice how the module name was not obtained from the "descr" field of your module structure (see page 30 about gmdnam()). That field is a copy of the module name in your .MDF file. If the Sysop innocently edits the .MDF file to change the module name, you probably don't want him to suddenly be missing all of your records in BBSGEN.DAT. That's why it might be a good idea to hard-code your module name in your records of BBSGEN.DAT.

The global variable `genbb` is declared in `MAJORBBS.H`.

To read the current user's score, you could code:

```
struct bggame bgbuff;
:
:
strcpy(bgbuff.userid,usaptr->userid);
strcpy(bgbuff.modnam,"My Game");
setbtv(genbb);
if (acqbtv(&bgbuff,&bgbuff,0)) {
    prf("Score: %d",bgbuff.score);
}
else {
    prf("No score recorded");
}
```

9. OFFLINE UTILITIES

This section covers routines that are used in the offline utilities and nowhere else. The offline utilities also make use of several routines that are used for the online operator interface, starting on page 137.

Most offline utilities have a basic background screen design which stays in view whenever the Operator is using that utility. These screens can be designed using TheDraw, and saved as 4000 byte .BIN files. For a utility to read a screen into memory at runtime, it uses iniscn():

<code>iniscn(filspc,where)</code>	Read an 80x25 character color screen from a .BIN file
<code>char *filspc;</code>	DOS path for the file
<code>void *where;</code>	buffer or video memory

The "where" parameter can be either an in-memory buffer (allocated by `alcmem(SCNSIZ)`, where `SCNSIZ` is defined as 4000 in `GCOMM.H`), or the actual video RAM address (see page 139 about `frzseg()`).

See page 164 for what `iniscn()` does on monochrome screens.

Window output (explode())

To make a window "pop up" on the screen, we use `explode()`:

<code>explode(sctptr,wulx,wuly,wlrx,wlry)</code>	Pop up a window on the CRT screen image (from <code>iniscn()</code>)
<code>char *sctptr;</code>	window upper left corner
<code>int wulx,wuly;</code>	window lower right corner
<code>int wlrx,wlry;</code>	(inclusive)

You can use TheDraw to design a pop-up window background in a .BIN file, then read it in with `iniscn()`, possibly modify it with `setwin()` and `printf()`, and display it with `explode()`.

The four windowing parameters in the `explode()` function define both where to read the image (relative to `sctptr`) and where to display it (on the CRT). So you really design where to put the pop-up window in the .BIN file with TheDraw, and just tell `explode()` what you came up with.

Or, you can pack many window backgrounds on a single .BIN screen and use `explodeto()` to put them anywhere on the CRT:

```
explodeto(sctptr, fux, fuy, flx, fly, tux, tuy)
char *sctptr;          Pop up a window on the CRT
int fux, fuy;         screen image (from iniscn())
int flx, fly;         source window upper left corner
                     source window lower right corner
                     (inclusive)
int tux, tuy;         dest. window upper left corner
```

If you don't like shadows, use `nsexploto()` instead of `explodeto()`, with the same parameters.

A call to any of the `explode()` family of routines automatically calls `proff()` with the `tux, tuy` parameters, so that you can use calls to `prat()` relative to where you put the window on the screen (see page 140 about `proff()` and `prat()`). This affects future operation of the `choose()` and `edtnval()` routines (see below).

All X coordinates range from 0 to 79, left to right, and Y coordinates range from 0 to 24, top to bottom.

The global variable `explodem` defaults to 1 for an animated exploding effect, but may be set to 0 to make the windows pop up instantly.

```
int explodem;          animate the exploding window?
                       1=animate, 0=instant
```

A "shadow" of one cell vertically and two cells horizontally is automatically applied to the bottom and right edges of each pop-up window.

When you're ready to pop up a window, first save the current screen image so you can restore it when you make the window disappear. For example:

```
char *scnsav;
:
movmem(frzseg(), scnsav=alcmem(SCNSIZ), SCNSIZ);
```

This allocates 4000 bytes and moves the current screen image to it. When done with the window, just restore the saved image back, as in:

```
movmem(scnsav, frzseg(), SCNSIZ);
free(scnsav);
```

Pop-up windows can be popped on top of one another. This means you'll need to have a series of saves and restores nested in one another, like this:

```
save background
pop-up #1
save window #1
pop-up #2
save window #2
pop-up #3
restore window #2
restore window #1
restore background
```

To be compatible with both monochrome and color screens, you can call `monorcol()`:

<code>monorcol()</code>	Determine monochrome versus color, based on the offline Hardware Setup option CRT (which is set to COLOR, MONO or AUTO)
<code>imonorcol()</code>	Determine monochrome versus color based on BIOS settings only. This is equivalent to <code>monorcol()</code> when CRT is set to AUTO. We use this routine in cases where we don't want to be depending on the BBSMAJOR.MCV file.
<code>int color;</code>	1=operator's screen is color 0=monochrome

If you don't want to depend on the CRT offline option, you could compute color automatically by some other method as in:

```
color=(FP_SEG(frzseg()) != 0xB000);
```

Don't define your own "color" variable -- use the global variable from `GCOMM.LIB`.

The color variable has a global effect on `iniscn()` and `setatr()` -- if color is set to zero, those routines will translate color values into reasonable monochrome values:

Automatic translation of color to monochrome

Any attribute with a white background becomes black on white (inverse video).

Otherwise, the attribute becomes white on black, preserving blinking and/or brightness, if they are present.

Window input (`edtval()`, `choose()`)

If you want the operator to enter something, you can use `explode()` to pop up a window (see the previous section), and then `edtval()` to handle his entry session.

<code>save=edtval(sx,sy,maxlen,sval,valor,flags)</code>	edit a string field on the screen
<code>int save;</code>	0=ESC hit, 1=Enter, Tab, Shift-Tab, cursor up or cursor down was hit
<code>int sx,sy;</code>	starting point for the field on the screen (sx is 0-79, sy 0-24)
<code>int maxlen;</code>	maximum size of string (including NUL -- maxlen-1 is field width)
<code>char *sval;</code>	default value / return value
<code>int (*valor)();</code>	validation routine
<code>int flags;</code>	options

The `sx,sy` coordinates are relative to the `tux,tuy` coordinates of the most recent call to the `explode()` family of routines (page 162). This is done via the `proff()` and `prat()` routines (page 140).

Put a default value in the sval buffer if you want one (the cursor will start at the right end of the value), or fill sval with a zero-length string to start from scratch. Either way, you have to have maxlen bytes available at sval. Here are the bit flag options for the last parameter of edtv():

```
#define MCHOICE    1    /* multiple choice question, hide cursor */
#define ALLCAPS   2    /* convert all chars to capital letters */
#define USEPOFF   4    /* use proff() x,y base coord offsets */
#define MULTTEX   8    /* allow multiple field-exit conditions */
```

The operator can type in a new value, move the cursor right or left, insert or delete characters, hit home or end, and when finally done, hit <Enter>, <Tab>, <Shift-Tab>, <up> or <down> to save or <Esc> to abort. Actually it's up to you how you handle the difference between all these exit methods. You can tell whether it was <Esc> or not by edtv()'s return value. You can distinguish among the other cases using the edtv() global variable:

```
int edtv();                Keystroke that ended edtv()
```

After edtv() returns, you can get the entry results in the buffer that the sval parameter pointed to.

While edtv() is running, the entry field will use setatr() attribute of 0x0F (bright white on black). For this reason, your pop-up window should probably have a background color other than black. When edtv() completes normally it restores the original attribute. If it completes with the <Esc> keystroke, the entry field stays visible.

The valrou parameter is the address of a keystroke validation function. It will be called each time the operator hits a key other than one of the exit keys (see the return value "save" above). The function is passed the code for the key pressed (see about key codes on page 144) and the buffer contents so far (NUL-terminated, without that keystroke). Your function should return a 1 to accept the keystroke or a 0 to reject it. Here are a few validation functions from GCOMM.LIB that you can use:

```
isok=validig(c,sval);      digit validation routine
int isok;                  1=it's a digit, 0=reject
int c;                     key code (ala getchc())
char *sval;                string entered so far

isok=validyn(c,sval);     yes/no validation routine
int isok;                  1=it's a digit, 0=reject
int c;                     key code (ala getchc())
char *sval;                string entered so far
```

Here's the source code for these routines:

```
int
validig(c,sval)          /* is c a valid decimal digit?      */
int c;
char *sval;
{
    return(c >= '0' && c <= '9');
}

int
validyn(c,stg)          /* validate c as yes/no (for edtval()) */
int c;
char *stg;
{
    if (tolower(c) == 'y') {
        strcpy(stg,"Yes");
    }
    else if (tolower(c) == 'n') {
        strcpy(stg,"No");
    }
    return(0);
}
```

Notice how you can allow a single character to change the entire entry string -- just write to the buffer pointed to by the sval parameter. The routine calling your validation routine adds the character to the buffer if your routine "accepts" it, and doesn't otherwise. Either way, the entire string is redisplayed after each keystroke.

When using validyn(), maxlen must be at least 4.

The following routine allows an offline operator to make a multiple-choice selection using a scrolling window, with up and down arrow keys highlighting the different choices, and <Enter> making the final choice:

```
choice=choose(nchoices,choices,upx,upy,lox,loy,escok);
int choice;
int nchoices;
char *choices[];
int upx,upy;
int lox,loy;
int escok;
Pop up a window of choices
index of choice 0..nchoices-1
or -ESC if operator escaped
number of choices
array of choice
upper left corner of window
lower right corner of window
allow ESC? 1=yes 0=no
```

The upx,upy coordinates are relative to the tux,tuy coordinates of the most recent call to the explode() family of routines (page 162). This is done via the proff() and prat() routines (page 140).

The window boundaries are inclusive. The window does not have to be big enough to hold all your choices, and if it isn't, choose() will show "(more)" at the bottom and scroll when the operator moves the cursor down. A few global variables are controlling the display attributes:

```
int selatr;          Attribute for scrolling choice bar
int nslatr;          Attribute for the other choices
```

There are some alternatives and variations to choose(). The first is choowd():

```
choice=choowd(choices,first,upx,upy,lox,loy,escok);
int choice;
char *choices[];
int first;
int upx,upy;
int lox,loy;
int escok;
```

Pop up a window of choices
index of choice 0..nchoices-1
or -ESC if operator escaped
array of choice, after the last of
which is a NULL
index of the "default" choice
upper left corner of window
lower right corner of window
allow ESC? 1=yes 0=no

Here too, upx,upy are relative to the window established by explode().

The two differences between choowd() and choose() are: choowd() uses a NULL to terminate the choices[] array while choose() passes the quantity nchoices; and choowd() allows you to specify a default starting point in the choice array, while choose() always starts you at index 0.

The third alternative is to break the choosing up into two pieces:

```
supchc(nchoices,choices,upx,upy,lox,loy,escok);
choice=choout();
```

This does exactly what choose() does, with the same parameters and return value, but you get the chance to sneak some processing in between the startup and the choosing session. You would only do this if you had knowledge of some of global variables in CHOOSE.C from the Extended C Source Suite.

The fourth alternative is to break the choosing up into many pieces. You'd need to do this in a multitasking environment so that you could be working on other tasks while waiting for keystrokes from the operator. Or you'd need this if you wanted to take some special action on certain keystrokes. Here you get the best of choose() and choowd() in kit form, with some assembly required. Here's the equivalent of choose() (with an optional starting point like choowd()):

```
supchc(nchoices,choices,upx,upy,lox,loy,escok);
jmp2chc(first);
dspchc();
cursiz(NOCURS);
do {
    choice=hdlchc(getchc());
while (choice == nchoices);
rstcur();
```

<- this line is optional

The jmp2chc() routine establishes the default or starting point, as does the "first" parameter in choowd(). The new routine dspchc() displays the background of the choice window after startup. Notice it's polite to turn the cursor off for the choosing session. The hdlchc() routine handles operator keystrokes. Of course, you could set things up to be doing other things while kbhit() is false, and only call getchc() when kbhit() is true. The hdlchc() routine has the same return value as choose() and choout(), except it may return nchoices to indicate that the choosing session isn't over yet.

Large Model Programming

Most offline utilities from Galacticom use the Large memory model of Borland C++ and do not make use of the Phar Lap DOS-Extender. This is a less complicated development environment than what we use to make MAJORBBS.EXE and all the .DLL files. These .MAK files come with The Major BBS Developer's C Source Kit:

BBSRPT.MAK	Offline reports (option 9 from the introductory menu, includes all eight reports)
GALP&QR.MAK	Offline polls & questionnaires analysis (GALP&QR selection from option 7 of the introductory menu)

Many more .MAK files come with The Major BBS Extended C Source Suite. These make files call out linker response files too. Remember to use the large model libraries for offline utilities.

Here are the most important differences in large model programming versus protected mode programming:

- o Smaller memory limits on the program (640K or so total, up to 64K static data)
- o Object files reside in \BBSV6\LOBJ instead of \BBSV6\PHOBJ.

To do the compiling and linking steps piece by piece:

To compile a <filename>.C source file that contributes to an .EXE offline utility program:

```
CD \BBSV6\SRC      or      CD \BBSV6\DDD  (as appropriate)
CTL <filename>    CTL <filename>
```

(It's not a good idea to do "CTL *" because different source files need to be compiled with different CTXXX.BAT files.)

To relink a <utility name>.EXE file:

```
CD \BBSV6\LOBJ
LNK <utility name> <other file 1> <other file 2>
```

or, as appropriate

```
CD \BBSV6\DDD
LNK <utility name> <other file 1> <other file 2>
```

Look in the corresponding .MAK file for the utility for the exact way to link it.

Language Editor DLLs

When you define a language, you can also define a custom editor program for editing text or other information in that language. Usually a custom editor will be associated with the protocol portion of the language, for example BBSDRAW for all languages that end in "/ANSI", or RIPaint for all "/RIP" languages.

Language Editors are used by CNF to edit text blocks and by Menu Tree to create and edit custom menus. Language Editor DLLs will run in protected mode, and they must behave appropriately. See page 186 for more on running in protected mode.

To create your own language editor DLL:

1. In your language .MDF file (page 26), use the name of your .DLL file in the language editor command line, as in:

```
Language Editor: TESTIT.DLL %s
```

This one directive can do up to three different things. First it declares that this editor is a DLL editor (as opposed to an editor that's an .EXE file or a .BAT file). Second, it specifies TESTIT.DLL as the DLL that should be loaded in order to run the editor. And third, your language editor handler routine may be able to use it to recognize when text should be edited in your language. When it comes time to edit something, the language editor command will be passed to all editor handler routines, and your editor handler routine will need to decide between "Hey, I'm supposed to edit the text," or "Nah, some other editor is supposed to edit the text, not me." More on this later.

By the way, for consistent selection of the proper editor under Menu Tree, a unique language editor should be associated with a unique language file extension. For example, the language editor command line "RIPaint.DLL %s" should be associated with, and only with, the language file extension ".RIP". This comes up when you're trying to define multiple RIP languages (English/RIP, Spanish/RIP, German/RIP, etc.).

2. Create an editor handler routine. A simple example:

```
int
tstedt(
char *command,
char *txtbuf,
unsigned sizbuf)
{
    if (!strcmp("TESTIT.DLL",command)) {
        return(EONOTME);
    }
    return(edit(txtbuf,sizbuf) ? EOSAVE : EONOCNG);
}
```

In this example, edit() is your function for editing the text, and it returns 1 if it wants to change the text or 0 if it doesn't.

As mentioned, whenever the Sysop wants to edit some text (when he types <F2>=EDIT in CNF, or he chooses to "Edit the way this menu looks" in Menu Tree), the language editor command line from the appropriate language .MDF file is formatted and passed to all registered editor handler routines. Each editor handler has the responsibility to look at the command and either (A) get to work (in the above example, to call edit(), and return either EOSAVE or EONCHG) or (B) pass the buck (to return EONOTME -- effectively saying "it's not my job").

Here are the meanings of the parameters that will be passed to your editor handler routine:

command This is the formatted language editor command line from the appropriate language .MDF file. You would typically look at the first word of this command to find out if your editor should be working on this text. If not, you need to return EONOTME. (See below for all possible return values.)

The "%s" from the language editor command line has been replaced by a file name by the time it gets to you in the form of this command parameter. You probably don't care about that file name unless txtbuf is NULL.

txtbuf If non-NULL, this is the address of the text in memory, and also where you should put the text after it has been edited. This is how CNF will call your language editor.

If NULL, you should get the text from the file identified in the command, and write the edited text back there too. This is how Menu Tree will call your language editor.

sizbuf This is the maximum size the text should attain. In no event should your editor handler write outside of the inclusive memory range txtbuf[0] to txtbuf[sizbuf-1]. (sizbuf does include the room for the terminating NUL.)

The possible return values of your editor handler routine are:

EONOTME	This command is for another editor
EOERROR	Error occurred (details in edterr[])
EOABORT	Operator aborted, recover old data
EONCHG	No change to data, don't update
EOTRUNC	Data truncated (ibsize=original size)
EOSAVE	Done editing, save data
EOSAVE+EOPGUP	Save data, skip to option above
EOSAVE+EOPGDN	Save data, skip to option below
EONCHG+EOPGUP	No change to data, skip to option above
EONCHG+EOPGDN	No change to data, skip to option below

These constants are defined in EDTOFF.H. If you can't decide which return value to use among EOSAVE, EOTRUNC, and EONCHG, use EOSAVE.

Here are some global variables associated with editors:

```
char edterr[];   where to put an error message
                  (used only when you return EOERROR)

long ibsize;     size of original text before it was
                  truncated (used only when you return
                  EOTRUNC)
```

The wording of the edterr[] message should be such that it fits well into a message like "This CNF Editor command <edterr>: <editor command from .MDF file>". For example, some appropriate wordings of edterr[] might be "cannot create CNF00000.FRC", or "requires more real-mode memory", or "erased the SAVE.TXT file". We suggest you test each of your error messages with CNF to make sure they look right.

Your editor handler routine and associated code will need to be in a C source file that includes:

```
#include "gcomm.h"
#include "edtoff.h"
```

3. Register your editor handler routine. Make a function whose name starts with "init_". It gets passed the address of the routine to register editor handler routines, and should be declared EXPORT. The function doesn't return anything. Here's an example:

```
void EXPORT
init_testit(regrou)
void (*regrou)(EDTHANDLER *edthdl);
{
    (*regrou)(tstted);
}
```

This function will get called the first time someone tries to use your editor, so you may want to include some more initialization code here.

A key strategy with each language editor DLL is not to load any DLL until and unless it's actually needed. So the first time the Sysop edits some text that's associated with a given language editor DLL is when that DLL is loaded and initialized.

You may be thinking there's a paradox here ("How can the Sysop pick my editor before I've even registered it?!?"). The resolution is in the multiple purposes of the language editor command line in the .MDF file. The first time the Sysop tries to edit something associated with your language editor DLL, your editor handler routine has not even been registered. But a special editor handler routine (dlload() in EDTOFF.C) will (1) recognize this condition, (2) notice that your command line "TESTIT.DLL <temp file>" calls out a DLL, (3) load and initialize your DLL, and (4) allow you to get to work editing the text. From then on, your registered editor handler will respond to all text edit sessions where your language is in effect.

4. Compile your program using CTPH.BAT, for example:

```
CTPH TESTIT
```

5. Make a linker response file:

```
TESTIT.LNK
\run286\bc3\lib\c0phdll +
\bbsv6\phobj\testit
\bbsv6\testit
nul
plhide phapi cnfimp /Twd /s /n
\bbsv6\dlib\nodef
```

And use it to make your .DLL file:

```
TLINK @TESTIT.LNK
```

If you get undefined symbols for Borland Library routines, you may have to find alternatives to using those routines. See page 20 about the perils of linking BCH286.LIB in a DLL.

6. Install your .DLL file, and the language .MDF file, on the BBS computers where you want the editor to be available.

.MSG File Reading and Writing

If your offline utility needs to examine the value of a CNF option direct from the .MSG file, as opposed to the compiled .MCV file, you can use msgscan().

value=msgscan(filnam,name);	Read a CNF option from an .MSG file
char *value;	value of option (or NULL=can't find)
char *filnam;	file (include .MSG extension)
char *name;	name of option

To read the option from the .MCV file, use the getmsg() and xxxopt() routines, described starting on page 65. It's usually more convenient to use getmsg() or the xxxopt() routines if you are reading a .MSG file that's part of the same product release. That is, if your .MSG file and your offline utility are sold and updated as a package.

On the other hand, if your product's utility is reading *another* product's .MSG file, msgscan() should be used. An example would be any offline Add-on utility that needs to know the values in BBSMAJOR.MSG. Using msgscan() allows your utility to continue to work even after the .MSG file is updated to a new version (as long as the option you're changing has not been obsoleted of course).

If you're writing an offline utility and you need to change the value of CNF options, you could use the setcnf() and applyem() functions.

setcnf(name,value);	Specify a CNF option change
char *name;	name of the option
char *value;	new value for the option

applyem(filnam);	Set the CNF options in this file
char *filnam;	file (include .MSG extension)

Here's an example of using these routines to set the values of several CNF options in different .MSG files.

```
setcnf("GROUP3","MODEM");
setcnf("BAUD3","19200");
setcnf("LOCK3","YES");
setcnf("INIT3","AT&FE0S0=0S2=255X6&R0B1");
setcnf("SUPCLS","PROSPECT");
applyem("BBSMAJOR.MSG");
applyem("BBSUP.MSG");
```

When you bring the BBS up again, new .MCV files will automatically be generated by BBSMSX.

Up to 100 setcnf() changes can be accumulated before you call applyem(). If you want to change more than 100 options, you can specify them in lots of 100 or fewer. (Calling applyem() sets things up so that the next call to setcnf() starts with a clean slate of specified changes.)

As you can see, you can specify changes to several different options in several different files. Then you can call applyem() on the file(s) where the options are stored. If you accumulate option changes for multiple files, beware of options with the same names in two different .MSG files (applyem() would change them both to the same value).

You can change the value of any type of CNF option with setcnf() and applyem(). But if you change the value of a type 'T' text option, only the language 0 version will be affected.

10. MORE ROUTINES AND VARIABLES

Character and String Routines

```
match=sameas(stg1,stg2);
int match;
char *stg1,*stg2;
```

```
case-ignoring string match
return code: true if strings are same
strings to test for matching
```

sameas() returns true if the two strings are the same, ignoring letter case, for example sameas("Fred W. Jones","FRED W. JONES") == 1.

```
match=sameto(shorts,longs);
int match;
char *shorts;
char *longs;
```

```
case-ignoring substring match
true if shorts = first part of longs
"short string": entirety must match
"long string": may have excess on end
```

The sameto() function is like sameas(), but it allows the first parameter to be just a portion of the second. For example:

```
sameto("good","gooder") == 1
sameto("good","good enough") == 1
sameto("best","best") == 1

sameto("badder","bad") == 0
sameto("women","womanhood") == 0
```

Another variation:

```
match=samend(longs,ends);
```

The samend() routine compares endings of strings, asking if the first string ends with the second string, ignoring case. Examples:

```
samend("the end","end") == 1
samend("dogs","s") == 1
samend("gooder","ER") == 1

samend("end","the end") == 0
samend("sheep","s") == 0
```

We remember the parameter order for sameto() and samend() by thinking of the prefix (shorts) sitting next to the prefix side of the long string (longs), therefore to the left of it in sameto().

In `samein()`, we think of the suffix (ends) sitting next to the suffix side of the long string (longs), therefore to the right of it. This way `samein("beg","beginning")` and `samein("ending","ing")` are both true.

```
found=samein(subs,string);      Search a string for a substring
                                (case-ignoring)
int found;                      1=found 0=not
char *subs;                     substring
char *string;                   string
```

This function searches the string for any occurrence of the substring, and returns 1 if it finds any. Examples:

```
samein("good","gooder") == 1
samein("s","UNITED STATES") == 1

samein("can't","cannot") == 0
samein("badder","bad") == 0
```

```
sprstg=spr(ctlstg,p1,p2,...,pn);  sprintf-like string formatter utility
char *sprstg;                    result string ptr (max 120 bytes)
char *ctlstg;                   control string (%l, etc. allowed)
TYPE p1,p2,...,pn;              sprintf-type parameters (max 12 bytes)
```

This routine is frequently used itself as an argument to `prf()`, `prfmsg()`, `catastro()`, or other "printf" format functions. Those functions do not support long integer or floating point conversions ("%ld" or "%f"), but `spr()` does. Warning: the string created by `spr()` must not exceed 120 bytes, including the terminating '\0'. Violation of this rule may have insidious results.

```
stzcpy(dest,source,nbytes);      Copy a string, with limit
char *dest;                      where to put it
char *source;                    the string
int nbytes;                      size of dest, including '\0'
```

if the source string takes up more than `nbytes` of space (including its NUL), then a truncated version is copied to `dest`. If the source takes up less, then the remaining bytes of `dest` are filled with NUL's. The `dest` buffer always gets at least one NUL at the end (assuming `nbytes > 0` -- if `nbytes <= 0`, then `stzcpy()` does nothing). Exactly `nbytes` bytes are written to `dest` (if `nbytes > 0`).

Use the following routines for parsing character strings of one or more "words" separated by whitespace characters:

```
pastwhite=skpwht(string);        Skip past whitespace
char *pastwhite;                pointer to the first NUL or non-
                                whitespace character in the string
char *string;                   NUL-terminated character string

pastword=skpwrld(string);        Skip past non-whitespace
char *pastword;                pointer to the first NUL or
                                whitespace character in the string
char *string;                   NUL-terminated character string
```

```

stg=l2as(lnum);          convert a long integer to an ASCII
                        decimal string of digits
char *stg;              where to store the result
long lnum;              input long integer

```

This function converts a 32-bit signed integer to a decimal character string. Note that:

```

spr("%ld",lnum)        .. is the same as ..      l2as(lnum)

```

Both spr() and l2as() use a 4-stage rotating-buffer technique to avoid the problem of multiple calls pointing to the same physical location. This means you can have up to 4 calls in a single parameter list before overlap will cause difficulties. For example:

```

prf("Shares traded today: %s %s %s",l2as(nyex),l2as(amex),l2as(otc));

```

Each of the three calls to l2as() will return the address of a different buffer -- not overlap three conversions into the same buffer. (Would you have thought of this problem? If not, be careful! Myself, I cannot answer whether this feature first came about by shrewd foresight or humbling debug.)

```

ptr=lastwd(stg);        get the last word of a string
char *ptr;              pointer to the last word
char *stg;              the input string

```

lastwd() finds the last "word" in a string. The return value points to the last nonblank character preceded by a blank (or it points to the beginning of the string if it can't find this).

```

xltctls(txtbuf);       translate ASCII control characters
char *txtbuf;          text buffer (input and output)

```

This routine translates ^ preceded letters into control characters. For example, the two-character sequence "^G" will be translated, in-place, to the single CTRL-G (ASCII BEL) character.

```

valid=isselc(c);        is c a valid menu-select character?
valid=istxvc(c);        is c a valid text-variable character?
valid=isuidc(c);        is c a valid user-ID character?
int valid;              1=yes 0=no
int c;                  character (all routines will return
                        false for non-ASCII values of c)

```

These are the routines we use for checking input strings for valid characters. International characters in the extended ASCII set are supported here.

Here are some more character-handling and string-handling routines:

```

yes=alldgs(string);    Is this string all decimal digits?
int yes;                1=yes 0=no, has other characters
char *string;           NUL-terminated string

```

<pre> oddparity=odd(somebyte); char somebyte; char oddparity; </pre>	<pre> Compute odd parity a byte a byte with odd parity and with the same lower 7 bits as somebyte </pre>
<pre> rmvwht(string); char *string; </pre>	<pre> Remove all whitespace characters string (conversion is in-place) </pre>
<pre> nremoved=depad(string); int nremoved; char *string; </pre>	<pre> Remove trailing blanks from string number of blanks removed NUL-terminated string </pre>
<pre> bufptr=unpad(string); char *bufptr; char *string; </pre>	<pre> Remove trailing blanks from string copy of string pointer NUL-terminated string </pre>
<pre> sortstgs(strings,nstrings); char *strings[]; int nstrings; </pre>	<pre> Sorts a bunch of strings by re- arranging an array of pointers array of pointers to the strings number of strings </pre>

See also the memory handling routines `movmem()`, `setmem()`, and `repmem()` on page 45.

Real-Time Routines (`rtkick()`, `rtihdlr()`, `interrupts`)

<pre> rtkick(time,rouptr) int time; void *rouptr(); </pre>	<pre> "kick off" routine after time delay number of seconds before "kickoff" pointer to routine to be kicked off </pre>
--	---

Naturally the time delay here is not wasted. Control returns to your calling routine almost instantly, and the specified number of seconds later (plus or minus a few), the specified routine is invoked. The invocation actually takes place at the call to `prcrtk()` found near the bottom of the main loop in `main()` in `MAJORBBS.C`.

To make a routine get called at regular intervals, your initialization code could call `rtkick()` on the routine, and then the routine could call `rtkick()` on itself. Use `time=60` for the routine to run once a minute, or `time=1` for once a second.

<pre> rtihdlr(rouptr); void (*rouptr)(void); </pre>	<pre> Register a real-time routine routine to call at 18 Hz </pre>
---	--

If you have a routine that you need to execute more often than once a second, you could register it with `rtihdlr()`. Once you start this, it runs the entire time the BBS is up (don't keep calling `rtihdlr()` on the routine like you could do with `rtkick()`). This routine will be running at interrupt level, so don't try any DOS or GSBL calls except the `chixxx()` routines (see the GSBL manual). And be sure the routine uses as short a time as possible, or the BBS will lose its real-time responsiveness.

<code>dsairp()</code>	Disable interrupts
<code>enairp()</code>	Enable interrupts

These routines can be used to disable interrupts for very brief sequences of code. Warning: disabling interrupts for too long can cause you to lose incoming characters at high baud rates. For example, 300 microseconds is too long for 38,400 baud.

<code>tix64k=hrtval();</code>	Read the free-running 65KHz timer
<code>unsigned long tix64k;</code>	upper word counts seconds, lower word counts 1/65536 seconds

This routine reads the GSBL long integer variable `btuhr` while interrupts are disabled, in order to avoid skew. You should almost always use `hrtval()` in place of referencing `btuhr`. To measure the time between two events you can call `hrtval()` at each event and compute the difference between the values (later value minus earlier value). The result will be a count, in 1/65536 second units, of the time between the two events. Of course, this won't work if more than 65536 seconds elapse between calls (about 18 hours).

Time and Date Routines

<code>date=today();</code>	Find out today's date
<code>int date;</code>	YYYYYYMMDD coding for today

This returns today's date in the format that DOS uses for dates:

YYYYYY-----	(Year-1980) * 512	Representing 1980 through 2107
-----MMMM----	Month * 32	Representing 1 through 12
-----DDDDD	Day of month	Representing 1 through 31

YYYYYYMMDDDDDD	Code for date
----------------	---------------

<code>day=daytoday();</code>	Find out what day of the week it is
<code>int day;</code>	0=Sunday 6=Saturday

<code>time=now();</code>	Find out what time of day it is
<code>int time;</code>	HHHHMMMMSSSS coding for time

This returns the time of day in the format that DOS uses for time:

HHHH-----	Hour * 2048	Representing 0 through 23
-----MMMM----	Minute * 32	Representing 0 through 59
-----SSSS	2-second intervals	Representing the even numbers 0 through 58

HHHHMMMMMMMMSSSS	Code for time
------------------	---------------

<code>ascdat=ncdate(date);</code>	Encode date into "MM/DD/YY"
<code>char *ascdat;</code>	local buffer for date
<code>int date;</code>	YYYYYYMMDDDD coding (see above)

<pre> asctim=nctime(time); char *asctime; int time; </pre>	<pre> Encode time into "HH:MM:SS" local buffer for time HHHHMMMMMMSSSSS coding (above) </pre>
<pre> ascdat=ncedat(date); char *ascdat; int date; </pre>	<pre> Encode date into "DD-MMM-YY" (European style) local buffer for date YYYYYYMMDDDD coding (see page 178) </pre>
<pre> date=dcdate(ascdat); int date; char *ascdat; </pre>	<pre> Decode date from "MM/DD/YY" YYYYYYMMDDDD coding (see page 178) or -1=invalid date format date string </pre>
<pre> time=dctime(asctim); int time; char *asctime; </pre>	<pre> Decode time from "HH:MM:SS" HHHHMMMMMMSSSSS coding (see page 178) or -1=invalid time format time string </pre>
<pre> count=cofdat(date); int count; int date; </pre>	<pre> Count of days since 1/1/80 number of days since 1/1/80 YYYYYYMMDDDD coding (see page 178) </pre>
<pre> date=datofc(count); int date; int count; </pre>	<pre> Compute DOS date YYYYYYMMDDDD coding (see page 178) number of days since 1/1/80 </pre>

See also page 182 for reading and setting a file's time and date.

Numeric Routines

<pre> smaller=min(a,b); bigger=max(a,b); </pre>	<pre> Find the smaller of two numbers Find the larger of two numbers </pre> <p style="margin-left: 40px;">Since these are implemented as macros, the numbers can be all int, all unsigned, or all long.</p>
<pre> absval=abs(signednum); </pre>	<pre> Find the absolute value of a signed integer (int or long) </pre>
<pre> newcrc=calcrc(oldcrc,ch); int newcrc; int oldcrc; char ch; </pre>	<pre> Iteratively calculate a 16-bit CRC CRC on N+1 bytes CRC on N bytes N+1'th byte </pre>

This routine calculates a 16-bit CRC based upon the polynomial $x^{16}+x^{12}+x^5+1$. This routine is used to support the flash protocol for games like Flash Attack! (tm). (Flash game capability is available with the Games and Entertainment Collection.) This is the same CRC as is computed for the XMODEM-CRC protocol.

Text File Scanning

This suite of routines can be handy for scanning one or more text files, especially if you're looking for lines that start with some prefix string. We use these routines to scan all .MDF files for all the lines we're interested in. (For a really thorough demo of the tfsxxx() routines, see how this is done in INTEGROU.C.)

This is not a good thing to be doing for online user processing however. For one thing, the work is unbounded (how many files? how many lines?) so it could hold up the BBS for an unacceptably long time. For another, you can't have multiple text file scanning operations going on at once -- they work off of global variables in TFSCAN.C (available with the Extended C Source Suite).

But the text file scanning routines can be real handy for offline processing, for initializing things before the BBS comes up, or for nightly cleanup operations.

nfiles=tfsopn(filespec);	Prepare to scan 1 or more text files
char *filespec;	file spec (may contain wildcards)
int nfiles;	number of files matching file spec
tfs=tfsrdl();	Read next line from file(s)
int tfs;	returns latest value of tfstate
int tfstate;	Scanning state, with these values:
TFSBGN	tfsopn() was just called, identifying 1 or more files
TFSBOF	preparing to scan a file (name can be found in tfsfb.name)
TFSLIN	scanning lines of a file (line is in tfsbuf)
TFSEOF	done scanning a file
TFSDUN	all files have been scanned
char *tfsbuf;	Line read in by tfsrdl(), if tfsrdl() returned TFSLIN

When you call tfsrdl(), the state value can be found either from its return value, or from the global tfstate variable. You should keep calling tfsrdl() until it returns TFSDUN. Other than that, the most interesting state/return value is TFSLIN. This means tfsrdl() has done it's duty and retrieved a line of text from the file or set of files. That line is available in tfsbuf. The return values TFSBOF and TFSEOF could be useful if you needed to do know when the scanning reached file boundaries.

If you're looking for a specific prefix, or set of prefixes, you can use tfspx() to find it and to isolate what follows the prefix:

found=tfspx(prefix);	Is the current line prefixed with this?
char *prefix;	Prefix string
int found;	1=yes, 0=no
	if yes, tfspx points to what follows the prefix
char *tfspx;	pointer to string following the prefix, with preceding white space removed

For example, if the line you read in was "DLL=GALBLAST", then tfspx("DLL=") would return 1 and tfspx would point to "GALBLAST".

Suppose you needed to scan all .MDF files on the BBS and pass all module names through a routine called modnam() and all developer names through another routine called develn().

```
if (tfsopn("*.MDF") > 0) {
    while (tfsrdl() != TFSUN) {
        switch(tfstate) {
            case TFSLIN:
                if (tfspx("Module Name:")) {
                    modnam(tfspst);
                }
                else if (tfspx("Developer:")) {
                    develn(tfspst);
                }
                break;
        }
    }
}
```

Notice how one or more spaces may follow the colons in the .MDF files, but they are skipped by tfspx().

If you have multiple levels of prefixes, this routine might be handy:

```
tfsdpr();                strip the prefix off of tfsbuf and
                        prepare for sub-prefixes (the dpr
                        stands for "deeper")
```

This routine is used in INTEGROU.C to isolate all of the "Language" lines in the .MDF files with one tfspx() call, and then handle them individually with more tfspx() calls on the sub-prefixes.

If you don't need to exhaustively scan all of the file(s) (say you're only interested in one occurrence of line starting with a certain prefix), you can abort scanning with tfsabt():

```
tfsabt();                abort text file scanning
```

This just makes sure that if a file is opened that it gets closed, which is usually a good idea. Otherwise, you need to keep calling tfsrdl() until it returns TFSUN.

Disk I/O

```
bufptr=mdfgets(buffer,size,fp); Read a line of text from a file
char *bufptr;                  copy of buffer
char *buffer;                  where to store the line
int size;                      maximum size, including '\0'
FILE *fp;                      file (from fopen())
```

This is just like the standard fgets(), except it uses '\r' as a line terminator (a hard carriage return on The Major BBS), and it won't have a line terminator on the last line if the file doesn't have it.

<pre>got=fgetstg(buffer,size,fp); int got; char *buffer; int size; FILE *fp;</pre>	<pre>Read a NUL-terminated string from file 1=got it OK, 0=error or EOF where to store the line maximum size, including '\0' file (from fopen())</pre>
--	--

This reads a NUL-terminated string from what's obviously not strictly an ASCII text file, and stores it in the buffer. The NUL is stored too, but the string will never take up more than size bytes. If it's too big, it will be truncated with a forced NUL at buffer[size-1].

<pre>nbytes=getdfre(diskno); long nbytes; int diskno;</pre>	<pre>Find out how many bytes are free on disk number of bytes disk number (0=default 1=A: 2=B:)</pre>
--	---

<pre>cntdir(path); char *path; long numfiles; long numbytes; long numbytes;</pre>	<pre>Count a set of files path specification (wildcards ok) total number of files counted total number of bytes counted greater number of bytes occupied by the files, considering cluster size</pre>
---	---

<pre>drive=drvnum(path); int drive; char *path;</pre>	<pre>Determine drive number from the path 0=current, 1=A:, 2=B:, 3=C:,... any DOS file specification</pre>
---	--

<pre>clbytes=clsize(drive); unsigned clbytes; int drive;</pre>	<pre>How big are the clusters on disk? number of bytes per cluster 0=current, 1=A:, 2=B:, 3=C:,...</pre>
--	--

<pre>realsiz=clfit(siz,clbytes); long realsiz; long siz; unsigned clbytes;</pre>	<pre>Total space used by a file siz rounded up to the next cluster useful size of file cluster size (from clsize())</pre>
--	---

<pre>ok=setdtd(fname,time,date); int ok; char *fname; int time; int date;</pre>	<pre>Set the time and date for a file (file must not be open at the time) 1=ok 0=couldn't find file. file path HHHHHHMMMMSSSS coding (see page 178) YYYYYYMMDDDD coding (see page 178)</pre>
--	--

<pre>ok=settnd(fname,gmt70); int ok; char *fname; long gmt70</pre>	<pre>Set the time and date for a file (file must not be open at the time) 1=ok 0=couldn't find file. file path Number of seconds since midnight 1/1/1970, GMT</pre>
---	---

<pre>timendate=getdtd(handle); long timendate; int handle;</pre>	<pre>Get a file's time & date 32-bit number encoding date<<16 and time: HHHHHHMMMMSSSS coding (see page 178) YYYYYYMMDDDD coding (see page 178) file handle from an fopen()'d file (returned by fileno(fp))</pre>
---	---

gmt70=gettnd(handle); long gmt70	Get a file's time & date Number of seconds since midnight 1/1/1970, GMT
int handle;	file handle from an fopen()'d file (returned by fileno(fp))
root=fnroot(filnam); char *root; char *filnam;	Extract 1-8 char file root extension and path prefix removed file name or entire file path
fnam=fnwext(filnam); char *fnam; char *filnam;	Extract file name with extension name after the path prefix file name or entire file path
rsvd=rsvnam(filnam); int rsvd; char *filnam;	Check if a file name is reserved 1=reserved, don't use, 0=OK to use file name or complete file path

This routine checks through the DOS list of device drivers and other reserved names to make sure a proposed file name will not be misinterpreted as a device. It's a great idea to use this routine especially if the file name is something a user typed (even if it's the Sysop user). Did you know that writing to "PRN.TXT" will output to your printer? Calling rsvnam("PRN.TXT") will return 1 and catch this and other disasters.

The following routines can be used to find out if a file exists, or to get help from DOS with breaking down a wildcard file specification (such as "*.EXE") into its component files.

yes=fndlst(&fb,filespec,attr); int yes; struct fndblk fb; char attr;	Any files in this filespec? 1=one or more, 0=none "findblock" structure attribute mask (see below)
yes=fndnxt(&fb); int yes; struct fndblk fb;	Any more files in this filespec? 1=yes 0=no more same structure passed to fndlst()

The fndlst() and fndnxt() functions make use of the following data structure, defined in DOSCALLS.H:

```

struct fndblk {
    char junk[21];          /* used by fnd1st,fndnxt in datntim.c */
    char attr;             /* file attribute (see masks below) */
    unsigned time;         /* time in HHHHHMMMMMSSSSS format */
    unsigned date;        /* date in YYYYYYMMMMDDDD format */
    long size;             /* size in bytes */
    char name[12+1];       /* name of file "FFFFFFFF.EEEE" */
};

```

After either routine returns 1, you can consult the name, size, date and time fields of the fb structure for information on the file.

The attr parameter to fndlst() restricts the search to certain types of files (or more accurately directory entries -- fndlst() and fndnxt() are really directory scanning routines). Here are the possible bit components to the attr parameter of fndlst(), and to the attr field of the fndblk structure. These are also defined in DOSCALLS.H:

```
#define FAMRON 0x01 /* File attribute: read only */
#define FAMHID 0x02 /* File attribute: hidden */
#define FAMSYS 0x04 /* File attribute: system */
#define FAMVID 0x08 /* File attribute: volume id */
#define FAMDIR 0x10 /* File attribute: sub-directory */
#define FAMARC 0x20 /* File attribute: archive */
```

The only useful values we ever found for the attr parameter of fndlst() are 0, meaning roughly to scan for files only, not subdirectories or volume ID's, or FAMDIR, meaning to check if a given directory name exists. You can refer to a DOS technical manual on interrupt 21H, function 4EH, for details on the other bits.

Everything Else

<code>cpu=cputype();</code> <code>int cpu;</code>	Returns 88, 186, 286, 386 depending on what type of processor you run it on (486's return 386)
<code>setcrit();</code>	Set the DOS critical error handler to a routine that pops up a window and is loads more friendly than "Abort, Retry, Fail". For real mode only -- don't use with Phar Lap.
<code>pasrcrit();</code>	Set the DOS critical error handler to a routine that retries three times then pops up a friendly window. Must run with the GSBL (Software Breakthrough).

11. RELIABILITY

Some Philosophy on Debugging

Your best debugging work takes place long before you test-run any of your software. A programming task that's well thought through and meticulously coded has the best chance of long term success.

- o Design with vision and foresight.
- o Code with attention to detail.
- o Test thoroughly.

Debugging work is inevitable. But you can save yourself lots of time and grief if you avoid slipping into the habit of using debugging as a programming tool. Routines should account for all possible conditions and be 100% predictable in the mind of the programmer before they are executed.

You're probably trying to increase your programming speed just like we are. Nothing will help your overall productivity more than an ingrained habit of completing highly reliable foundations before you proceed to use them. There are many techniques for making your foundations highly reliable: code walk-throughs, rigorous testing by the developer, line-by-line testing of source code, branch-by-branch testing, routinely retesting "repaired" code, alpha-testing (in-house), beta-testing (by motivated customers). The goal is to get software done fast, put it to use, and move on to other projects, without your customers having to find the problems and you solving them over the phone.

Programming Tips for The Major BBS

There are several unique things about the programming environment of The Major BBS. The multi-user single-tasking aspects mean that your code can't wait very long for any one thing to happen.

Once you become proficient in programming for The Major BBS, there are a few classes of problems you'll want to be on guard against. If you review these problems and develop an eagle eye for them, then your code can be much more reliable and free from frustrating bugs.

Don't forget to set the message pointer with setmbk():

```
Wrong way
int
hdlstt(void)
{
    prfmsg(NOTAVAIL);
    return(0);
}
```

```
Right way
int
hdlstt(void)
{
    setmbk(lclmb);
    prfmsg(NOTAVAIL);
    return(0);
}
```

(lclmb is the return value of the original call to opnmsg())

Remember to set the database pointer with setbtv():

```
Wrong way
int
lookup(char *name)
{
    return(qeqbtv(name,0));
}
```

```
Right way
int
lookup(char *name)
{
    setbtv(gnfbb);
    return(qeqbtv(name,0));
}
```

(gnfbb is the return value of the original call to opnbtv())

Don't call outprf() multiple times (resulting in double prompts):

```
Wrong way
prfmsg(GREETING);
if (problem) {
    prfmsg(WARNING);
    outprf(usrnum);
}
outprf(usrnum);
```

```
Right way
prfmsg(GREETING);
if (problem) {
    prfmsg(WARNING);
}
outprf(usrnum);
```

General Protection Faults

The best thing and the worst thing about protected mode operation is the "General Protection" fault, or GP. Protected mode is designed to halt at the slightest sign of a program gone awry. The first time you try to unravel and debug a GP will be an adventure. But after all the work of tracking it down, it will probably lead you direct to a bug in your code.

What is a GP?

A General Protection fault is the computer CPU's way of saying "I give up" or "I can't do this". Usually it has to do with an invalid or restricted memory address. (There are other GP causes, like privilege violation, executing the wrong code, or "switching to a busy task", whatever that is. If you're getting one of these, your program has probably gotten pretty out of control.)

GP's don't catch every bug, not by a long shot. But they can catch certain thorny problems way before they get out of hand. The worst kinds of bugs to find are intermittent with inconsistent symptoms: you can't make them happen when you want to, and it's always something different when they go wrong. Or this variation: you make what *should* be an irrelevant modification, perhaps to add some debugging or probing code, and the symptoms change. This is exactly what can happen from random unintentional memory reads or writes, and this is exactly the kind of thing that a GP is likely to catch in the act, with a bull's-eye on the instruction that made the memory reference. With some patience, you can sift through the clues at the scene of the crime and find out exactly what went wrong. This chapter is all about helping you get started with GP detective work.

Kinds of GP's

Illegal memory accesses can happen in hundreds of squirrely little ways, some of which may even be completely harmless in a program running in real mode. In protected mode, however, they cause a GP fault. Here are three common programming errors that can result in GP's.

A. NULL passed to a routine that uses it as a pointer

For example, consider the following code fragments:

```
...    blah(margv[0],NULL);
...
...    blah(stg1,stg2)
char *stg1,*stg2;
{
    if (strcmp(stg1,stg2) == 0) {
        ...
```

This will cause a GP when `strcmp()` is called, because although NULL is not an illegal value for a pointer, to access memory through it is an illegal operation. Right? When you pass NULL as a pointer, you never mean "go to the memory at absolute memory location zero and look at whatever bytes are there." You mean "this here is the absence of a pointer". This is not a problem in real mode ordinarily because absolute memory location zero has some garbage in it that is unlikely to match a normal string. Referring to NULL may be similar to referring to the empty string, "", and it is possible, when writing real mode programs, to form the bad habit of using the two interchangeably. Protected mode is not going to let you get away with this.

A special case of this class of problem worth noting is the use of any of the macros from `STDIO.H`: `getc`, `putc`, `getchar`, `putchar`, `ferror`, `feof`, or `fileno`, with a NULL file pointer. You might think that attempting to do file I/O with a NULL or garbage file pointer would have no result. It turns out that it results in arithmetic being done on some illegal location in memory, which, in protected mode, will cause an immediate GP.

B. Accessing off the front or end of an array

The most common way this happens in practice is if you use invalid array indices. However, not every invalid array

access attempt will cause a GP. It is complicated, because protected mode will only detect an attempt to go outside a whole segment, and one segment may contain several arrays, variables, structures, etc. A classic example of this kind of bug is the innocent looking little binary search routine in none other than Kernighan & Ritchie's revered tome "The C Programming Language", first edition, page 129 (we reformatted the code to match our own formatting standards):

```
1  struct key *binary(word,tab,n)
2  char *word;
3  struct key tab[];
4  int n;
5  {
6      int cond;
7      struct key *low,*mid,*high;
8
9      low=&tab[0];
10     high=&tab[length-1];
11     while (low <= high) {
12         mid=low+(high-low)/2;
13         if ((cond=strcmp(word,mid->keyword)) < 0) {
14             high=mid-1;
15         }
16         else if (cond > 0) {
17             low=mid+1;
18         }
19         else {
20             return(mid);
21         }
22     }
23     return(NULL);
24 }
```

This code begins to come unglued if the sought keyword fits sequentially before all entries in the sorted tab[] array, and the tab[] array just happens to physically reside at the beginning of a segment or selector (that is, its offset is 0x0000). Then the instruction in line 14 eventually does this: high=&tab[-1]. That pointer computation results in an offset of 0xFFFF or thereabouts. This does not generate a GP yet, because the "high" pointer is not dereferenced.

But now things go from bad to worse, because the test in line 11, which is supposed to fail and halt the loop, instead succeeds, and mid is inexorably dragged upward until it points beyond the end of the region allowed for the selector. Finally, a GP occurs in the strcmp() routine invoked in line 13.

To fix this code, you could put the following three lines between lines 13 and 14:

```
13.1         if (mid == low) {
13.2             break;
13.3         }
```

And, to be truly thorough about it (in protected mode this tends to be a good idea), you can guard against the possibility that the top of the table is flush up against the high bound of a

full 64K data segment by inserting the following between lines 16 and 17:

```
16.1         if (mid == high) {
16.2             break;
16.3         }
```

Another example of this is a situation in which you wish to truncate a string to a certain maximum length, without knowing in advance just how much storage has been allocated for the string. Consider the following code fragments:

```
... speak("hi there");
... speak(stg)
char *stg;
{
    char temp;

    temp=stg[20];
    stg[20]='\0';
    printf(stg);
    stg[20]=temp;
}
```

In real mode, this is not a problem because even if the memory location that is 20 beyond the start of stg is in some other segment, you are putting it back to what it was when you are through. In protected mode, though, this type of thing can generate an immediate GP, depending on how close the "hi there" string is to the end of the segment in which it is allocated.

- C. Too few arguments in a call to printf() or prf() or prfmsg()

If your control string has more %s place-holders in it than there are pointers in your argument list, the machine will find itself picking up random stack contents as a pointer, and accessing memory there. Actually, the mere loading of a random stack value into a selector register is likely to generate a GP right off; if not, then accessing memory through a random offset is likely to exceed the length bounds of the selected segment.

Sysop GP Handling

When a Sysop of The Major BBS wants maximum up-time for his BBS, he sets the offline Hardware Setup option GPHDLR to YES:

```
GPHDLR    Continue operation after "GP" errors? ..... YES
```

This causes the BBS to attempt to recover from GP errors after reporting them in the Audit Trail (see the System Operations Manual for details). To be on the safe side, the user being serviced at the time of the GP is logged off immediately, and his channel is reset. But the BBS stays up and running (or tries to).

This also helps with hacker protection if a user discovers a way to generate an otherwise harmless GP: with GPHDLR set to YES, a GP simply logs him off.

Developer GP Handling

But developers should use GP's on their demo or support BBS to maximum advantage for tracking down bugs.

GPHDLR Continue operation after "GP" errors? NO

In this mode, extensive information on the GP is captured in a text file \BBSV6\GP.OUT.

Reading a GP.OUT Report

A GP is a software interrupt 13 (coincidence?). You could also get other software interrupts for other disastrous conditions like interrupt 12 for a stack fault. (One possible cause of interrupt 12 is an infinitely recursive routine.) But only a GP can provide you with a lengthy report.

Keep in mind that GP.OUT accumulates reports, so you'll want to look at the bottom of the file.

Here is an example of one report in a GP.OUT file:

```
BBS GP @ 2d4f:031a EC 0000 (recorded 12/16/93 17:11:17)
GP'ed between 2d4f:0000 _INIT_TELECON and 0000:0000 Unknown
AX=0000 BX=0000 CX=0000 DX=0787 SI=0005 DI=0054 BP=2a88 ES=0000
DS=2d57 Flags=3246
Current CS:IP=>26 83 0f 01 b8 f7 04 8e c0 26 c4 1e 87 08 26 c7
```

```
BBS Version: 6.20
User 3 channel 00 User-ID "Sysop" status 3
Online level 6, state 5, substate 0
MSG:galtlc.mcv/-1
Module "Teleconference"
Input "T"
```

```
0167=C:\BBSV6\MAJORBBS.EXE
0717=C:\BBSV6\GALGSBL.DLL
073f=C:\BBSV6\BBSBTU.DLL
075f=C:\BBSV6\DOSCALLS
2a5f=C:\BBSV6\GALMSG.DLL
2cff=C:\BBSV6\GALRSY.DLL
2d3f=C:\BBSV6\GALTLC.DLL
2d6f=C:\BBSV6\GALT XV.DLL
2d97=C:\BBSV6\GALUIE.DLL
2dbf=C:\BBSV6\GALXIT.DLL
2de7=C:\BBSV6\GALTST.DLL
```

```
Stack:
0707:2a78 04ff 0746 04ff 0054 2f47 04ff 0054 0005
0707:2a88 >2aaa< 0ba3 0197 04ff 0000 098a 0197 04f7
0707:2a98 0000 4f54 0050 2aaa 0029 039f 0074 0065
0707:2aa8 04f7 >2ac0< 529a 018f 0039 28ef 0001 0000
0707:2ab8 04f7 1b32 0003 0000 >2ace< 3579 018f 04f7
0707:2ac8 000e 4f84 018f >2b30< 3099 018f 04f7 022f
0707:2ad8 018f 0707 1b14 0000 0000 0000 0000 0000
0707:2ae8 0000 0000 0000 011f 96c7 0017 0006 0000
0707:2af8 0707 0000 011f 966b 0017 0e29 0000 0e29
0707:2b08 0000 2b36 0707 0707 0001 1330 26cc 0147
0707:2b18 1b32 0707 1b32 1b14 2b36 4c48 016f 0578
0707:2b28 0707 0587 0707 000f >0000< 014f 016f 0000
0707:2b38 0000 0000 fdcc 077f ---- ---- ----
0707:2b48 ---- ---- ---- ---- ---- ---- ----
0707:2b58 ---- ---- ---- ---- ---- ---- ----
0707:2b68 ---- ---- ---- ---- ---- ---- ----
0707:2b78 ---- ---- ---- ---- ---- ---- ----
0707:2b88 ---- ---- ---- ---- ---- ---- ----
```

```
Routines:
0197:07c5 _GOPAGE < 0197:0ba3 < 0197:0e3d _DSPMNU
018f:4e23 _NXTLOF < 018f:529a < 018f:53be _CURUSR
018f:3487 _HDLINP < 018f:3579 < 018f:35e4 _CLR XRF
018f:2502 _KILETC < 018f:3099 < 018f:3145 _CONDEX
0000:0000 Unknown < 016f:014f < 016f:0155 _CLEANUP
```

GP Location

The first line of GP.OUT has the CPU instruction pointer CS:IP contents, which reflect the location (address) of the fault. Usually this points to an instruction that the CPU refused to execute for some reason. This line also tells you when the GP occurred. If you see GPCNT here, then the Sysop has GPHDLR set to YES and the GP reported is not the first (and perhaps not the most informative -- have them set GPHDLR to NO and get you another GP.OUT).

GP Vicinity

The second line of GP.OUT attempts to place the CS:IP between two public symbols that you can search for in your source files. Only symbols declared EXPORT are included here though. The only EXPORT routine in your whole DLL is probably your init_xxx() routine (page 17). If the GP occurred in a source file with no EXPORT routines, you're likely to see some irrelevant addresses. You can still get an idea what file the GP is in from the DLL list. More on that below.

For a reproducible GP (one you can make happen any time you want), you might declare some routines EXPORT just to get more information here. Often, narrowing the problem down to one routine is enough to start studying the routine and to find the problem.

Registers

This section shows the contents of the assembly language registers right when the GP occurred.

Code at the GP Location

Here on the fifth line of GP.OUT are 16 bytes of executable code starting from the GP location. This includes the actual instruction that caused the GP. This object code can come in handy when searching for the exact locations in an assembly listing (more below).

User Conditions

The next section of GP.OUT show information on the most recent BBS activity on a user's channel.

BBS Version: 6.20	BBS software version
User 3	User number (usrnum, see page 39)
channel 00	Channel number (hexadecimal, page 39)
User-ID "Sysop"	User-ID on channel most recently serviced
status 3	Status code from the channel
Online level 6	usrptr->class, 6 is "ACTUSR" -- online and active
state 5	usrptr->state (see page 31)
substate 0	usrptr->substt
MSG:galtlc.mcv	the current .MCV file being read
/-1	the last message read from this file (-1=none)
Module	From the usrptr->state code (5 in this example)
"Teleconference"	
Input "T"	Most recent status-3 input line on the BBS

The `usrptr->state` code represents the module the user was in. Sysops choose the module by name when they create a module page during offline Menu Tree Design. Module names are also listed in the Miscellaneous Statistics screen of the BBS console.

In many cases this information at the top of a GP report can be very helpful. But don't trust it too much. The most recent user activity is not necessarily related to the GP at all. In fact, you should carry a highly skeptical attitude into all of your debugging efforts, always seeking a thorough and precise understanding of what your program is doing, while understanding thoroughly how it is supposed to work.

To put more debugging information into GP.OUT, you can carefully add some code to the lower part of `appgprept()` in `PLBBS.C`. You'll see the `fprintf()` calls that formatted all of this information, along with careful checking in case pointers are not usable. (Using a bad pointer inside the GP report routines is like accidentally exploding a bomb at the crime scene -- you'll lose the evidence from the original crime due to the new one on your hands.)

DLL Selectors

This is a list of the starting selectors for each of the DLL's you have loaded on the BBS. You can find out in which DLL the GP occurred by finding the highest DLL starting selector that is less than the GP selector.

In the example above, the GP occurred at address `2d4f:031a`. That's somewhere in `GALTLC` because `GALTLC.DLL` starts at `2d3f` and the next one, `GALTXV` starts at `2d6f`. There's a little more you can find out here. Selectors are created in increments of 8 hexadecimal.

```
2d3f  GALTLC starting selector
2d47  \run286\bc3\lib\c0phdll.obj
2d4f  \bbsv6\phobj\mjrtlc.obj
2d57  \
2d5f  > (other selectors used by GALTLC)
2d67  /
2d6f  GALTXV starting selector
2d77  :
2d7f  :
```

After the starting selector, selectors are created for each of the `.OBJ` files in the order they are listed in the `.LNK` file. In `GALTLC.LNK` there are two `.OBJ` files. The GP occurred in the code from `MJRTLC.OBJ`, which of course comes from `MJRTLC.C`. The offset starts at `0000` in the code from that file, so the error occurred `031a` bytes into the object code produced from `MJRTLC.C`.

Note: This selector counting falls apart if you see a `.LIB` file called out in the `.LNK` file -- this may link the equivalent of several `.OBJ` files and use up several selectors. There is a `.LIB` file linked among the `.OBJ` files in `LTBBS.LNK`, for example, the linker response file that goes into making `MAJORBBS.EXE`.

Once you get a feel for these things you'll know that `031a` should be pretty near the beginning of a file as big as `MJRTLC.C`.

Stack Dump

The stack dump is very relevant to "the crime" of the GP, and it's rich with information -- too rich. Let's jump ahead to the "Routines" section with the return address chain at the bottom. That's where the most meaningful information is pulled from the stack dump and given symbols from the source code where possible. We'll come back to the stack dump later.

Routines in the Return Address Chain

At the bottom of GP.OUT are the chain of return addresses pulled from the stack. These reflect the nesting of subroutines that the CPU was executing at the moment of the GP. The first one in the list, 0197:0ba3, is pulled from the stack at 0707:2A8A or at SS:BP+2. If you know how C language subroutines are implemented in assembly language, it's not hard to realize that this is the return address to some ancestor routine that called (perhaps indirectly) the routine in MJRTLC.C where the GP was triggered.

The first thing that happens in some subroutines is "PUSH BP" then "MOV BP,SP". This means the 2 bytes at [BP+0] are the old BP, and the 4 bytes at [BP+2] are the return address. But not all routines save and reload BP. Since it's the chain of BP's in the stack that we trace, not actually the return addresses, some routines get "skipped". If a routine doesn't save and reload BP, we won't find the return address to its parent that was pushed when the routine was called. So its parent will appear to be skipped in the list of routines at the bottom of GP.OUT.

So far we know:

1. Somewhere in a routine early in MJRTLC.C a GP was triggered.
2. That routine was called (perhaps indirectly) by a routine between `gopage()` and `dspmnu()`.

A "grep" of source files turns up `gopage()` and `dspmnu()` in MENUING.C. The call to the routine in MJRTLC.C must be somewhere between these points. That is, in one of the routines `gopage()`, `gomodl()`, `gocond()`, `gomenu()`, or `gofile()`. (The call couldn't be in `dspmnu()` itself because 0197:0ba3 is less than 0197:0e3d, the start of `dspmnu()`.)

The call must be in `gomodl()`, because that fires off the `sttrou()` entry point for a module page (page 33), and early in MJRTLC.C is `telecn()`, the `sttrou()` entry point for the Teleconference module.

The next ancestor of `gomodl()` shown in the return address chain is between `nxtlof()` and `curusr()`. This has a different selector than `gomodl()` (018f versus 0197) so it must come from a different file. In fact it comes from MAJORBBS.C. You'll notice that there's no calls to `gomodl()` in all of MAJORBBS.C, but there's one in MENUING.C inside of `gopage()`. Hmm, this is making sense. There are reams of calls to `gopage()` in MAJORBBS.C for menu tree pages in general. And `gopage()` calls `gomodl()` for module pages in particular. `gopage()` was skipped in the return address chain. That's probably because `gomodl()`, after various compiler optimizations and shenanigans, didn't need to save and reload BP.

Stack Dump, Revisited

The notations like >2aaa< in the stack dump are BP save locations. When traversing the chain of saved and loaded BP's to generate the return address chain at the bottom of GP.OUT, the BP save locations are flagged with little ">" <" symbols. The return addresses shown at the bottom immediately follow each of the ">" <" symbols. For example, the ">2aaa<" is followed by 0ba3 and 0197. 0197:0ba3 is the first return address in the list.

Close scrutiny of the stack dump can help us determine the values of parameters passed between routines, and the values of automatic stack variables. In some cases, the values of pushed registers are also of some help in investigating the circumstances of the crime. But to do any of this, we have to look at the assembly language listing of the code where the GP happened and where the ancestor routines did their subroutine calling. And an assembly language listing lets us find out the exact instruction that caused the GP. Let's start with the routine where the GP occurred, probably telecn(), definitely in MJRTLC.C.

Assembly Listing

To get the assembly language listing of MJRTLC.C, we have to generate assembly source code and then assemble it and make a listing:

```
CD \BBSV6\SRC
CTDLL -S MJRTLC
CD \BBSV6\PHOBJ
TASM MJRTLC,NUL,MJRTLC;
```

(Use "CTPH -S" for getting the assembly listing of files that are part of the kernel -- that go to make up MAJORBBS.EXE.) The "-S" parameter (capital S, not small s) asks for an .ASM assembly source file to be output instead of an .OBJ. Then TASM assembles \BBSV6\PHOBJ\MJRTLC.ASM and makes a listing file in \BBSV6\PHOBJ\MJRTLC.LST. (The NUL means don't generate an .OBJ file from the .ASM file.) That .LST file is pretty big. It's where the offsets and object code can be found. To find what we're looking for, there are two clues: The GP location is 2d4f:031a, so we can look in the vicinity of the offset 031a. And the code somewhere near there should be 26 83 0f 01 (from the fifth line of GP.OUT). Turns out, it's somewhere exactly at 031a:

```
Turbo Assembler      Version 3.1      07/13/92 11:19:52      Page 10
mjrtlc.ASM

530                ;
531                ;
532                ;           case 0:
533                ;           tptr->flags|=NOPAGE;
534 0316 C4 1E 0077r  ;           les   bx,dword ptr tptr
535 031A 26: 83 0F 01 ;           or    word ptr es:[bx],1
536                ;
537                ;           usrptr->substt=1;
538                ;
539 031E B8 0000s    ;           mov   ax,seg_usrptr
540 0321 8E C0      ;           mov   es,ax
541 0323 26: C4 1E 0000e ;          les   bx,dword ptr es:usrptr
542 0328 26: C7 47 08 0001 ;          mov   word ptr es:[bx+8],1
```

Notice that the sixteen bytes of code don't match exactly where "0000s" goes with "f7 04"? Those "0000s" and "0000e" symbols means that the exact object code is not known at compile (or assembly) time. It will take linking and loading to find out those exact values. So, the object code does confirm that we're looking at the right instruction (the "or" instruction).

So, here's the crime scene, what was the crime? The instruction was "or word ptr es:[bx],1". That's a bitwise OR of the value 1 into some memory location. Notice that the very helpful assembly comments show the original C source code near the relevant assembly code. Turns out the NOPAGE constant is 1, so "tptr->flags|=NOPAGE" is for sure the guilty instruction. A quick look at the registers in line 3 of GP.OUT shows that es:bx is 0000:0000. That's pretty wild, isn't it?. OR'ing to the location addressed by NULL is definitely GP territory. From MJRTLC.C we see that "flags" is the first field of the tlc structure (tptr is type "struct tlc *"), so tptr itself must be NULL.

A scrutiny of the C source logic reveals that tptr is not expected to be initialized here: it's just a low-level looping pointer. We cheated of course; this is exactly the line we inserted to make this GP happen.

INDEX

- %-symbols, 61
- .ANS files, 26
- .ASC files, 26
- .BCR (Btrieve database creation) files, 156
- .DAT (database) files, 25, 156
- .DEF files, 19
- .DLL files, 17-18
 - creating, 19-21
 - custom language editor, 169-172
 - language editor, 27
 - rebuilding, 21
- .DOC files, 24
- .EXE files
 - language editor, 27
 - rebuilding MAJORBBS.EXE, 17
- .IBM files, 26
- .LIB files
 - created by IMPLIB, 19
- .LNK files, 19, 20, 172, 192
- .MCV files
 - using, 69
- .MDF files, 8, 23-28
 - language definition, 26
 - language editor, 169-172
- .MSG files, 8
 - .MDF directives for, 25
 - adding to, 55-63
 - format, 56
 - language limit, 54
 - levels, 55-56
 - routines for reading and writing, 172
- .RLN files, 8
- .VIR (virgin database) files, 156
- .ZIP file, 125
- 65536 Hz timer, 178
- ^ (caret character), 176

A

- aabbtv(), 151
- Abort uploads, 112
- abs(), 179
- absbtv(), 151
- acclass (structure), 159
- Accounting, 93, 93-96
- acqbtv(), 155
- Add-on Options, 6
- agebtv(), 155
- agtbvt(), 155

- ahibtv(), 156
- alcblok(), 43
- alcdup(), 43
- alcmem(), 42
- alcrsz(), 43
- alctile(), 44
- alczer(), 43
- alebvt(), 155
- alldgs(), 176
- Allocating memory, 42-45
- alobtv(), 156
- altbtv(), 155
- ANSI Graphics, 140-142
- ANSI screen attributes, 141
- ansion(), 140
- applyem(), 172
- aqnbtv(), 156
- aqpbvtv(), 156
- Arguments, user input, 74
- Assembly listing, 194
- Attributes, display, 137
- Attributes, file, 184
- Audit trail, 39, 148
- await (integer), 88
- Auto-cleanup, (see Cleanup)
- Autosensor routines, 86-89
- auxcrt(), 139

B

- baudat(), 148
- BBSes
 - Galacticomm's Demo System, 22, 102, 106, 109, 114, 125
 - Phar Lap's, 5
- BBSGEN.DAT, 160
- BBSMSX utility, 18, 21, 64
- BBSPRV (user class), 31, 85
- BBSRPT.MAK, 168
- BBSV6, 13
- BCH286.LIB, 19, 172
- begin_polling(), 50
- belper(), 140
- bgnenc(), 75
- bgnedt(), 99
- Binary (yes/no) CNF options, 58
- Blank padding, 177
- Block memory allocation, 43
- Btrieve, 6
- Btrieve database engine, 52, 149-156
 - .MDF directive for, 25
 - using BUTIL, 156
- btuhrt (variable), 178
- btvblk (structure), 149
- BTVFILE (structure), 149
- BTVSTF.H, 16
- BUTIL (Btrieve utility), 156
- By reference upload, 111
- BYEBYE flag, 83
- byenow(), 83

C

- C source conventions, 16
- calcrc(), 179
- catastro(), 51-52
- Channel number, 39
- Channel status, 148
- Channel type, testing, 40
- channel[], 40
- Character CNF options, 58
- Character string handling routines, 174-177
- Charging users, 93
- chimove(), 45
- chkdft(), 80
- choose(), 166
- choout(), 167
- choowd(), 167
- chropt(), 68
- Class database, 159
- Cleanup
 - .MDF directive, 25
 - mcurou() entry point, 38
 - statistics screens, 146
- clfit(), 182
- clingo (integer), 53, 54
- Closing database, 153
- clreol(), 139
- clrmlt(), 70-73
- clrprf(), 70, 71
- clrxrf(), 80
- clsbtv(), 153
- clsize(), 182
- clsmsg(), 67
- Cluster size, 182
- cncall(), 77
- cncchr(), 76
- cncdex(), 76
- cncint(), 76
- cncing(), 54, 77
- cnclon(), 76
- cncnum(), 76
- cncsig(), 76
- cncuid(), 76
- cncwrd(), 76
- cncyesno(), 28, 76
- CNF options, 55-68
 - changing, 68
 - compiling, 64
 - creating, 55-63
 - format, 56
 - hinge interdependency, 62
 - languages, 55, 63
 - level numbers, 55-56
 - routines for reading and writing, 172
 - types, 58-61
 - updating to new version, 8
 - using online, 65-68
- cntcand(), 89
- cntdir(), 182

- Code examples
 - beeping the operator, 140
 - catastro, 52
 - choose() equivalent, 167
 - choosing language, 54
 - CNF option writing, 173
 - command concatenation, 78
 - cycle mediating, 49
 - database acquire, 156
 - database get, 155
 - database query, 154
 - downloading, 129-130, 131-135
 - feedback to Sysop, 100-103
 - finish entry point, 38
 - generic user database, 160
 - GP generating, 188
 - handle-connect vector, 84
 - initializing a module, 30
 - language editor handler, 169
 - multilingual, 70
 - pseudo-key handler routine, 92
 - scanning a text file, 181
 - uploading, 114-115, 116-120
 - video output routines, 143
 - window entry validation, 166
- cofdat(), 179
- color (global variable), 164
- Color
 - ANSI coding, 141
 - IBM display attribute, 138
 - versus monochrome, 164
- Command concatenation, 75-79
- Commands, global, 96
- Compiler libraries, 19
- Compiler updates, 22
- Compiling
 - CNF options, 64
 - for a .DLL, 18, 21
 - MAJORBBS.EXE, 17
 - offline utilities, 168
 - online source, 18
- Compressed file, 125
- Computer requirements, 1
- CONCEX flag, 78
- condex(), 78
- Confidence factors (autosensing), 88
- CONNECT string handling vector, 86
- Connect time, intercepting vectors, 84-86
- Context of user, 31
 - state code, 192
- Control characters, 176
- cputype(), 184
- CRC (cyclic redundancy check), 179
- crdrat (usrptr-> field), 96
- Creating databases, 156
- Credit consumption rate, 96
- Credits, 93, 93-96
- Critical error handler, 184
- Cross referencing User-IDs, 79
- CTDLL.BAT, 18, 21

- CTL.BAT, 168
- CTPH.BAT, 18, 194
- curatr (structure), 143
- curcurs(), 145
- curcurx(), 139
- curcury(), 139
- curmbk (pointer), 66
- currsact(), 140
- currsiz(), 145
- cursor
 - positioning
 - ANSI command, 141, 142
 - locate() function, 139
 - size, 145
- curusr(), 73
- Customizing, 1
- Cycle mediating, 48

D

- Data structures, 41-47
 - Btrieve databases, 149
- Databases, 149-161
 - acquire, 155
 - close, 153
 - creating, 156
 - delete, 152
 - file identifiers, 149
 - functions, 149-156
 - generic user, 160
 - get, 154
 - insert, 152
 - opening, 150
 - physical-order scan, 151
 - query, 153
 - spare space, 160
 - system variables, 157
 - update, 152
 - user account, 158
 - user class, 159
 - variable length record, 153
- Date and time
 - files, reading and setting, 182-183
 - routines, 178-179
 - upload handler formats, 110
 - upload handler setting, 112
- datofc(), 179
- daytoday(), 178
- dcddate(), 179
- dclvda(), 46
- dctime(), 179
- Debugging, 185-195
- Decorator analogy (exit points), 108
- dedcrrd(), 94
- Default selection character, 80
- delbtv(), 152
- Delete, database, 152
- depad(), 177

- Developer's C Source Kit, 4
- Developer-ID, 3
 - reserved, 3
- Development, 1
 - directories, 6, 14
 - environment, 11-22
 - your own Add-on Options, 6, 19, 23, 29
- dinsbtv(), 152
- Directories
 - development, 14
 - overall structure, 13
 - runtime, 13
 - size measurement, 182
 - test for existence of, 184
 - your development files, 6
- Disk I/O routines, 181-184
- DISK1.DID, 8
- dlarou() module entry point, 38
- DLL (Dynamic Link Library), 19,
 - (see also about .DLL files at the beginning of the index)
- dload(), 171
- DOS critical error, 184
- DOS device names, 183
- DOS file time and date, 182-183
- DOSCALLS.H, 184
- DOSFACE.H, 16
- Download, 121
 - ASCII, 120
 - download handler routine, 124-128
 - examples, 129-130, 131-135
- Drive number, 182
- drvnum(), 182
- dsairp(), 178
- DSKUTL.H, 16
- dspchc(), 167
- dupdbtv(), 152
- Duplicate string allocation, 43

E

- Echo on/off/secret, 75
- echon(), 75
- echsec(), 75
- Editor DLLs, 169-172
- edterr[], 170
- edtimr, 99
- EDTOFF.C, 171
- EDTOFF.H, 170, 171
- edtval(), 164
- edtvalc (integer), 165
- enairp(), 178
- endcnc(), 76
- English/RIP, installation, 9
- Entering and exiting a module, 34

- Entry point (module)
 - dlarou(), 38
 - finrou(), 38
 - huprou(), 38, 47
 - injrou(), 35
 - lofrou(), 37, 46
 - lonrou(), 32, 46
 - mcurou(), 38
 - ststrou(), 35, 46
 - sttrou(), 33, 46
 - variables, 31
- Enumerated CNF options, 59
- Error message output, 51
- Examples, (see Code examples)
- Exception handling, 51-52
- EXICNC, 78
- explode(), 162
- explodem (integer), 163
- explodeto(), 163
- EXPORT (routine type), 17
- Exported symbols
 - Borland libraries, 19
- Exporting symbols, 17, 19, 24
- Extended C Source Suite, 5
- Extensions on file names, 15
- extoff(), 37, 53, 71, 82
- extptr (pointer), 31, 53, 54, 71

F

- FAMDIR (mask for fndlst()), 184
- Feedback example, 100-103
- fgetstg(), 182
- File names
 - Developer-ID prefix, 3
 - extensions, 15, 26, (see also the beginning of the index)
 - parsing, 183
 - reserved, 183
- File time and date, 182-183
- File transfer
 - ASCII download, 120
 - defining a protocol, 136
 - upload, 107
 - using, 107-135
- fileup(), 107
- FILEXFER.H, 107, 121
- File
 - attributes, 184
 - directory structure, 13
 - finding, 183
 - handles, 50
 - I/O routines, 181-184
- finrou() module entry point, 38
- Flash games, 2
- fndlst(), 183
- fndblk (structure), 183
- fndnxt(), 183
- fnroot(), 183

- fnwext(), 183
- fopen(), 50
- Free disk space, 182
- free()
 - with alcblok() or alctile(), 45
 - with alcmem(), 42
 - with stgopt(), 68
- frzseg(), 139
- fsddan(), 106
- fsdft(), 106
- fsdnan(), 106
- fsdord(), 106
- fsdpan(), 106
- fsdroom(), 103
- fsdxan(), 106
- FSE, (see Full Screen Editor)
- fstcand (integer), 89
- ftfbuf (buffer), 109
- ftfmsp (protocol specifications), 109, 125
- FTFREF flag, 111
- ftfscb (session control block), 109, 125
- ftgnew(), 121
- ftgptr, 121
- ftgptr (tag table entry), 125
- ftgsbm(), 122
- ftplog(), 136
- Full Screen Data Entry, 103-106
- Full Screen Editor, 99-103
 - example, 100-103
- FUPXXX, upload handler exit points, 110-113

G

- gabbtv(), 151
- Galacticomm registering names, 2
- GALDNX, download example, 129-130
- GALDNX2, download example, 131-135
- GALFBK feedback example, 100-103
- GALIMP.LIB, 22
- GALP&QR.MAK, 168
- GALUPX, upload example, 114-115
- GALUPX2, upload example, 116-120
- GCOMM.H, 16
- GCOMM.LIB, 16
- gcrbtv(), 155
- gdedcrd(), 94
- genbb (pointer), 161
- General Protection (GP), 186-195
- Generic user database, 160
- gen_haskey(), 90
- geqbtv(), 154
- getasc(), 67
- getbtv(), 150
- getchc(), 144
- getdfre(), 182
- getdft(), 80
- getdtd(), 182
- getmsg(), 66

- gettnd(), 183
- ggebtv(), 154
- ggtbtv(), 154
- ghibtv(), 154
- glebtv(), 154
- Global commands, 96-98
 - possible vdaptr conflict, 42
- globalcmd(), 97, 98
- globtv(), 154
- glbtbv(), 154
- gmdnam(), 24, 30
- gnxbtv(), 154
- GP.OUT, 190-195
 - customizing, 191
 - example, 190
- GPHDLR (offline Hardware Setup option), 189
- gprbtv(), 154
- Group number (channels), 39, 40
- Group type code, 40
- grpnum[], 40
- grtype[], 40
- gtstcrd(), 95

H

- Handling, user, 81-86
- Hanging up, 83
- haskey(), 91
- hasmkey(), 90
- hdlc25 (handle-X.25-connection vector), 86
- hdlhc(), 167
- hdlcnc (handle-connect-string vector), 86
- hdlcon (handle-connect vector), 84
- hdlnrg (handle-non-RING-string vector), 85
- hdlrng (handle-RING string vector), 85
- hdluid(), 79
- Hexadecimal CNF options, 60
- hexopt(), 68
- Hinge specification (on CNF options), 62
- hrtval(), 178
- huprou() module entry point, 38
 - use of VDA in, 47

I

- IBM display attributes, 137
- ibm2ans(), 142
- ibsize (for language editors), 170
- IMPLIB (Borland's utility), 22
- IMPLIB utility, 19
- Import libraries, 19
- inimsg(), 65
- iniscn(), 162, 164
- Initialization routine, 29
- init_XXX() routine, 17, 19, 29, 97
 - language editor, 171

- injoth(), 72
 - with injrou(), 35
- injrou() module entry point, 35
- inplen (integer), 74
- input[], 74
- Input
 - offline operator window, 164-167
 - operator keystrokes, 144-145
 - user keyboard, 74-80
- insbtv(), 152
- Insert, database, 152
- INSTALL.EXE, 8
- Installation
 - Add-on Options, 6
 - Btrieve, 6
 - Developer's C Source Kit, 4
 - Extended C Source Suite, 5
 - of your Add-on Option, 8
 - .MDF directive, 24
 - testing, 7
- instat(), 81
- Intercepting connect-time vectors, 84-86
- Intercepting user input, 98
- INTERNAL (.MDF directive), 25
- invalid pointer, 61
- invbtv(), 152
- isripo(), 82
- isripu(), 82
- isselc(), 176
- istxvc(), 176
- isuidc(), 176
- ISX25 flag, 40

J

- jmp2chc(), 167

K

- kbhit(), 144
- Keyboard input
 - operator, 144-145
 - scan codes, 144
 - user, 74-80
- Keys (security), 90-92

L

- l2as(), 176
- LAN channel, testing for, 40
- LANGOP, 89
- Language subsets, 57, 70, (see also the System Operations Manual)
- Languages, 53-54
- languages[] array, 53

Languages

- .MDF files, 26-28
- autosensing, 88
- changing CNF options offline, 173
- choosing, 54
- editor DLLs, 169-172
- in .MSG files, 55, 63
- maximum number of, 54
- user input, 77
- user output, 70-73

Large memory allocations, 43

Large model programming, 168

Large numeric CNF options, 60

lastwd(), 176

lclmbk (pointer), 66

ldedcrd(), 94

Levels in .MSG files, 55-56

lingyn(), 53, 77

Linking

- .DLL, 18, 19
- MAJORBBS.EXE, 18
- offline utility, 168
- undefined symbol errors, 19
- your .DLL, 21

listing(), 120

llnbtv(), 153

lngfnd(), 53

lngfoot(), 54

lnglist(), 54

lngopt(), 67

LNK.BAT, 168

locate(), 139

Locks & Keys, 90-92

lofrou() module entry point, 37, 46, 71

Logging off, 83

lonrou() module entry point, 32, 46, 71

ltstcrd(), 95

M

MAJORBBS.DEF, 22

MAJORBBS.EXE, 17-18

margc (integer), 74

margn[], 74

margv[], 74

Matching strings, 174-175

max(), 179

maxcand (integer), 89

MAXTAGS (CNF option), 121, 123

mcurou() module entry point, 38

mdfgets(), 181

memcata(), 52

Memory

- allocation, 41, 42-45
- available, 45
- handling, 45

Menu select character, 176

Menu Tree, editor selection, 169

- min(), 179
- Model of compiler
 - Large model, 168
 - Phar Lap Huge model, 14, 17
- module (structure), 30, (see also the MAJORBBS.H file)
- Module definition files, 23-28, 29
- Module Name, 24
- Monochrome CRT, 164
- Monolingual routines, 71
- monorcol(), 164
- morcnc(), 76
- movmem(), 45
- msgscan(), 172
- Multi-user programming, 48-50
- Multilingual, 53-54, (see also Languages)
 - user output routines, 70-73
- Multiple choice (operator selection), 166

N

- Names, registering, 2
- ncdate(), 178
- ncedat(), 179
- nctime(), 179
- ndedcrd(), 94
- nlingo (integer), 53
- NOGLOB flag, 98
- NOINJO flag, 72
- now(), 178
- nsexploto(), 163
- nslatr (variable), 166
- nterms (integer), 39
- ntstcrd(), 95
- NUL padding, 175
- null pointer, 61
- numbytp (cntdir() output), 182
- numbyts (cntdir() output), 182
- numcand (integer), 89
- Numeric CNF options, 59
- Numeric
 - checking for, 176
 - conversion, 176
 - routines, 179
- numfils (cntdir() output), 182
- numopt(), 67

O

- obtbvt(), 151
- odd(), 177
- odedcrd(), 94
- Offline operations, 162-173, (see also Operator)
- Offline utility programming, 162-173
 - .MDF directive, 25
- omdbvt(), 150
- onbbs(), 82

- Online User Information screen, 148
- Online, user determining, 81
- onsys(), 70, 81
- onsysn(), 82
- Operating environment, 23-68
- Operator
 - cursor, 145
 - input, 144-145
 - interface, 137-145
 - multiple choice, 166
 - offline programming, 162-173
 - output, 137-144
 - services, 146-148
 - string entry, 164
 - window input, 164-167
 - window output, 162-164
- opnbtv(), 150
- opnmsg(), 65
- Options, CNF, 55-68, (see also CNF options)
- othkey(), 91
- othuap (pointer), 81
- othusn (integer), 81
 - use in injrou() module entry point, 36
- othusp (pointer), 81
- otstcrd(), 95
- outmlt(), 70, 70-73
- outprf(), 69, 70
- Output
 - offline operator, 162-164
 - operator, 137-144
 - user, 69-73
- Overview, 1

P

- Padded blanks, 177
- Padded NUL's, 175
- Paradox (language editor loading), 171
- Parity, 177
- parsin(), 74
- Parsing
 - file names, 183
 - general routines, 175, 176
 - user input, 74, 76
- pascrit(), 184
- Password echo mode, 75
- Permission, (see Security)
- PFBSIZ, 69
- PFCEIL, 75
- pfnlvl (integer), 75
- PHAPI.H, 16
- Phar Lap DOS-Extender
 - directories, 14
 - installation, 5
 - updates, 22
 - using, 17
- PHGCOMM.LIB, 50
- PLPLAT2.ZIP, 5

- pmlt(), 70-73
- pointer, null or invalid, 61
- Polling routine, 50, 71
- Pop-up windows, 162-167
- poslng (pointer to 2D array), 88
- prat(), 140, 163
- prf(), 69, 71
- prfbuf (buffer), 70
- prfbuf (internal buffer), 69, 70
- prfbuffers (array), 71
- prfmlt(), 70, 70-73
- prfmsg(), 65, 70
- prfpointers (array), 71
- prfptr (internal pointer), 69, 70
- printf(), 137
- printfat(), 140
- Profanity, 75
- proff(), 140, 163, 165
- Protected mode, 186-195
- Protocol definition, 136
- Protocol validation, file transfer, 108, 122
- Protocols
 - download, 122
 - upload, 107
- Prototypes (C functions), 16
- Pseudo-keys, 92
- ptrblok(), 44
- ptrtile(), 44

Q

- qqqbtv(), 153
- qgebtv(), 153
- qgtbtv(), 153
- qhibtv(), 154
- qlebtv(), 153
- qlobtv(), 153
- qltbtv(), 153
- qnxbtv(), 153
- qprbtv(), 153
- qrybtv(), 150

R

- rawmsg(), 67
- rdedcrd(), 94
- Reading .MSG files (offline), 172
- Reading lines from a text file, 180, 181
- Real-time routines, 177-178
- Reference upload, 111
- regautsns(), 87
- Registering
 - autosensor routines, 87
 - Developer-ID's, 2-3
 - global commands, 98
 - modules, 30

Registering (continued)
 names with Galacticcomm, 2
 offline language editors, 171
 pseudo-keys, 92
 statistics screens, 146
 text variables, 73
register_module(), 30
register_pseudok() (pseudo-keys), 92
register_stascn(), 146
register_textvar(), 73
Release notes (.RLN files), 8
Reliability, 185-195
Remodeling analogy (exit points), 108
Replaces, .MDF directive, 24
repmem(), 45
Requirements, computer, 1
Requires, .MDF directive, 24
Reserved DOS device names, 183
Resizing allocated memory, 43
Resuming an aborted upload, 110
RING string handling vector, 85
ripdfd (integer), 82
ripidx (integer), 82
rmvwht(), 177
rstbtv(), 150
rstcur(), 145
rstloc(), 139
rstmbk(), 66
rstrin(), 74
rstwin(), 138
rsvnam(), 183
rtihdlr(), 177
rtkick(), 177
rtstcrd(), 95
Runtime directory, 13
Runtime environment, 11

S

sameas(), 174
samein(), 175
samend(), 174
sameto(), 174
Scan codes (keyboard), 144
Scanning .MSG files, 172
sclank(), 139
scnoff(), 143
secchr (character), 75
Secret echo mode, 75
Security, 90-92
 uploading files, 110
selatr (variable), 166
Selection character, default, 80
Selectors, 192
Services, user, 90-136
setatr(), 137, 164
setbtv(), 150
setbyprot(), 89

- setcnf(), 172
- setcrit(), 184
- setdtd(), 182
- setmbk(), 66
- setmem(), 45
- settno(), 182
- setwin(), 138
- shibtv(), 151
- shochl(), 148
- shocst(), 148
- Size of a set of files, 182
- Size restrictions on uploads, 111
- sizmem(), 45
- skpwht(), 175
- skpwrn(), 175
- slobtv(), 151
- snxbtv(), 151
- Software updates, 22
- sortstgs(), 177
- Spare space in databases, 160
- Splitting up a big task, 48-50
- spr(), 175
- sprbtv(), 151
- Stack dump, 193
- State code, 192
- STATIC (routine type), 17
- Statistics, 146-147
- status (variable), 35
- Status of users, 81
- Status, user, 81-86
- stgopt(), 68
- stop_polling(), 50
- stranslen(), 106
- String CNF options, 60
- String handling routines, 174-177
- stsrout() module entry point, 35, 46, 71
- sttrou() module entry point, 33, 46, 71
- stzcpy(), 175
- Submitting tagspec for download, 122
- Subsets (language), 57, 70, (see also the System Operations Manual)
- supchc(), 167
- System Variables database, 157

T

- Tag table entry (for downloads), 121
- tagspec, 123
- Tagspec (for downloading), 121
- Task splitting, 48-50
- TASM assembler, 194
- Terminal
 - user input, 74-80
 - user output, 69-73
- Text CNF options, 61
- Text File Scanning (tfsxxx() routines), 180-181

- Text variables
 - defining, 73
 - translating, 67
 - using, (see your System Operations Manual)
 - valid characters in name, 176
- tfsxxx() routines, 180, 180-181
- Tiling large memory regions, 44
- Time and date
 - files, reading and setting, 182-183
 - routines, 178-179
 - upload handler formats, 110
 - upload handler setting, 112
- Timing routines, 177-178
- TLINK linker, 18, 19, 21
- today(), 178
- tokopt(), 68
- tshmsg (buffer), 125
- tstcrd(), 95

U

- uacoff(), 37, 82
- uhskey(), 91
- uinsys(), 81
- UNCONDITIONAL (.MDF directive), 25
- Undefined symbol errors (linker), 19
- unfrez(), 139
- unpad(), 177
- Update, database, 152
- Updates, software, 22
 - .MSG files, 8
- updbtv(), 152
- Upload, 107
 - by reference, 111
 - examples, 114-115, 116-120
 - resume after abort, 110
 - upload handler routine, 108-113
- upvbtv(), 152
- usaptr (pointer), 31
- user (structure), 31, (see also the MAJORBBS.H file)
- User account database, 158
- User class database, 159
- User number, 39
- User-ID
 - cross referencing, 79
 - validity testing, 176
- user[], 37
- User
 - hanging up on, 83
 - input, 74-80
 - intercepting input, 98
 - interface, 69-89
 - output, 69-73
 - services, 90-136
 - status, 81
 - status and handling, 81-86
- usracc (structure), 31, 82, 158

usridx(), 40
usrnum (integer), 39
 changing its value, 73
usrptr (pointer), 31
Utility, (see Offline utility)

V

valdpc(), 122
validig(), 165
validyn(), 165
valupc(), 108
Variable length record, database, 153
Variables for module entry points, 31
vdaoff(), 47
vdaptr (character pointer), 46
vdasiz (integer), 46
vdatmp (character pointer), 47
Vectors, connect-time handling, 84-86
Versions (software), 21
vidkey(), 91
Viewing compressed files, 125
Volatile Data Area, 32, 33, 41, 42, 46-47
Voting confidence factors, 88

W

Whitespace characters, 175, 176, 177
Wildcards
 breaking down with fndlst(), 183
 downloading application, 131
 file/directory counting, 182
 text file scanning, 180
Window (pop-up)
 input, 164-167
 output, 162, 162-164
Writing .MSG files, 172

X

X.25 channel, testing for, 40
X.25 connection handling vector, 86
xltctls(), 176

Y

Yes/No (binary) CNF options, 58
ynopt(), 68

Z

ZMODEM resume after abort, 110

