Server Developer's Guide

for The Major BBS v10

Galacticomm, Inc.

Document version 1.0

Overview

The Major BBS communicates with its users in Terminal mode, the traditional centralized processing approach.

Note: The Client/Server mode available in Worldgroup v3.x is no longer supported in The Major BBS v10 nor is ActiveHTML.

This Server Developer's Guide describes the development environment for The Major BBS. It explains the internal workings of The Major BBS server and the terminal-mode user interface. You should read this guide first.

The GSBL Guide describes low-level communications functions.

Requirements

On the server side of your development environment, you need:

- A PC capable of running Microsoft Visual Studio Community Edition
- The Major BBS v10
- Microsoft Visual Studio Community Edition
- The Major BBS v10 module SDK or The Major BBS v10 core SDK

If you need to create new databases for your application, please contact one of the development team for further details.

This Server Developer's Guide assumes:

- You have installed The Major BBS.
- You have read *Sysop's Guide* for The Major BBS.
- You are proficient in using The Major BBS via modem or other interface.
- You are proficient in managing The Major BBS from the Main Console.
- You are proficient in the C language.

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

You Can Modify The Major BBS in Stages

Changes made to The Major BBS fall into either of two broad categories:

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

The main purpose of *This Server Developer's Guide* is to help developers create new products for The Major BBS, but it will also be helpful if you're running The Major BBS and wish to customize it by making changes or additions to the source code. From simple cosmetic changes to the authoring of new applications, The Major BBS is as flexible as you require.

You can customize the values of several thousand CNF options. These allow you to steer each baseline module through a set of pre-defined courses: charging for messages sent vs. not charging, requiring keys for various activities, even fine-tuning the frequency of various events.

More importantly, the Text Block type of CNF option allows you to customize completely the language of the text presented to terminal-mode users (and some of the text presented to C/S-mode users).

Basic Utilities let you manage data files and do other tasks offline.

Menu Tree page settings allow you to change how the facilities of your The Major BBS are presented to your users. You can control both the layout and the appearance of your system here.

You can change almost everything else in The Major BBS environment: the CNF program itself, WGSDRAW, the Menu Tree design program, etc.

Installation

To begin, simply clone the SDK from GitHub and open the solution file in Visual Studio.

The MBBS V10 Module SDK solution contains the relevant .LIB, .H and source files for a simple sample module. It also contains example .MDF, .MSG and .VIR/.DAT files. These are covered in greater detail further in this guide.

In the case of the core SDK, the source files that make up the BBS itself are included (such as APIs, apps, etc) – approximately 150 projects in total.

The solution file has the appropriate properties set for linking, structure types and other settings required to make it compile. It is suggested that these not be modified unless there is a very specific reason.

All modules are required to be compiled as x86 as x64 is not supported.

Distributing your own modules

In order to distribute your module, simply zip up the contents of the /Dist folder along with the release version of your DLL. Once unzipped into The Major BBS installation directory (C:\BBSv10 by default) the module will be "installed". Sysops can then use the Configuration Editor to change settings and text blocks as required.

.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 WGSMAIN.RLN to Sysops when they first install The Major 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 .RLN's 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 59 for full details on creating .MSG files, but here's an example of how it works:

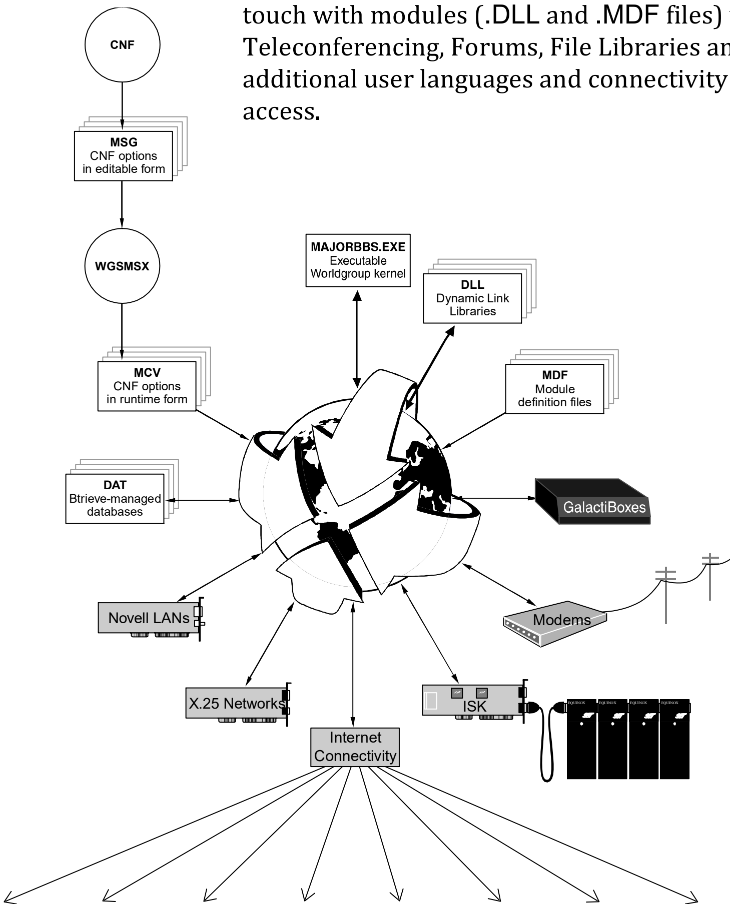
Suppose you supply a DDDSAT.MSG file with an option named HOOPLA{}. If a DDDSAT.MSG file with a HOOPLA{} option already exists on the Sysop's system, then the contents of that option (what's between the curly braces) in the Sysop's old DDDSAT.MSG displace those from your release disks. In all other respects (option type, description, help message), the contents of your DDDSAT.MSG file are updated on the Sysop's system.

The Major BBS's Environment

The Run-time Environment

This is the way a Sysop without Developer's Kits sees The Major BBS:

MAJORBBS.EXE is the kernel of The Major BBS server, putting users in touch with modules (.DLL and .MDF files) which offer activities such as Teleconferencing, Forums, File Libraries and E-mail. Other modules provide additional user languages and connectivity options such as X.25 and Internet access.

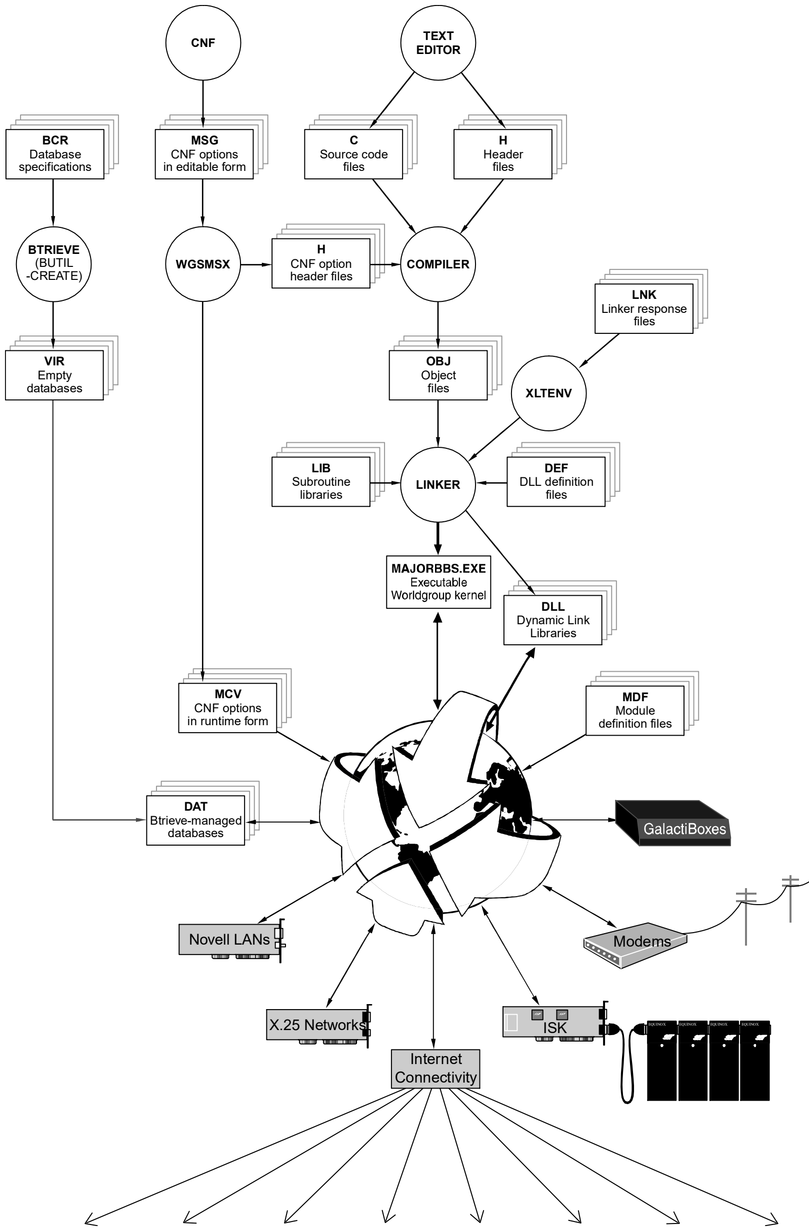


Module developers provide what they feel are a sufficient number of CNF options, giving Sysops some control over modules and users. The CNF program lets Sysops edit .MSG files, which the WGSMSX program then converts into indexed .MCV files for faster performance online. Developer's Kits are necessary to create new CNF options because this involves changing source code.

Btrieve .DAT databases store user account information, messages, files online, etc. To create new databases, or restructure existing ones, you need Developer's Kits plus Btrieve's development software.

The Development Environment

Behind the scenes are Developer's Kits and programs used with them:



CNF option .MSG files are analyzed by WGSMSX, producing header (.H) files telling the source code how to locate CNF options indexed in the .MCV files.

The compiler merges .C source code files with .H header files, producing object files (.OBJ).

The linker, guided by .LNK files, links object files with subroutine libraries (.LIB) and definition files (.DEF) to form executable code, either the kernel, MAJORBBS.EXE, or add-on module .DLLs.

When installing an add-on, if active data files (.DAT) don't exist yet, our INSTALL program copies virgin data files (.VIR) to same-named .DAT files.

You need Btrieve's BUTIL program to make empty .VIR database files from the .BCR specification files.

Galacticomm's development environment supports, and we encourage the use of, optionally compiled debugging code. This means that you can write code into your source files which can be compiled, or not compiled, depending on which batch file you use to make the project. Most of the development environment's batch files come in pairs:

Debugging version, filename ending in D	produces programs with debug code, giving you greater feedback during development and testing phases
Final version	produces programs without debug code, as intended for final release

See page 302 for more details on debugging.

The Run-time Directory

Almost all of the files that The Major BBS needs to run will be located in the \BBSV10 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 **Error! Bookmark not defined.** for conventions on file naming with Developer-IDs. See page 7 for the conventions on file extensions. When you run The Major BBS, don't execute MAJORBBS.EXE directly, use WG.BAT instead.

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 WGSMSX)
.HLP	Help topic text files
.IBM	Text file with ANSI commands and Extended ASCII characters
.IDX	Internal data file
.INS	Instructions
.LNK	Linker response files specify 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 (run-time form)
.MDF	The Major BBS module definition file
.MSG	CNF options (editable form)
.OBJ	Object files (compiler output)
.REF	Internal reference 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 **Error! Bookmark not defined.** on the use of Developer-IDs as file naming prefixes.

A few unique filenames:

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

To avoid conflicts between filenames and directory names wherever possible, filenames should have nonblank extensions and directory names should have blank extensions.

C Source Conventions

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

- 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:

#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 Galacticomm 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
MSGRDR.H	Reading .MSG files (for offline utilities)

#includes Phar Lap's header file PHAPI.H

Defines prototypes for functions in PHGCOMM.LIB and LGCOMM.LIB

Defines constants for the second parameter of the fopen() function (see page 54)

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 getchc(), see page 250)

Defines other constants, data types, and macros

- Galacticomm C Source code uses function prototypes. This means:

  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

  Assembly language functions are prototyped as much 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.

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

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

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

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

- Your init__xxx() routines (page 23) should be coded as EXPORT, as in:

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

Building projects / generating your DLL files

Since moving to MSVC, the process of building The Major BBS and add-on modules has been greatly simplified. There is no longer any requirement for .MAK or.BAT files to help the linking and compilation process as it is all handled through the Visual Studio IDE.

Building is simply a matter of building via Visual Studio as you would for any other project. Right-click the project you wish to build and select Build this project, or select Build from the toolbar at the top.

Again, ensure you are building for x86!

Compiled DLLs will be placed in either a Release or Debug folder under your solution folder. Ensure you distribute the Release version of your DLL when you are ready to distribute your module to Sysops.

Operating Environment

The Major BBS is designed as a central hub (MAJORBBS.EXE) which governs free-standing (and therefore easily interchangeable) *modules*.

Each module has a module definition file (.MDF). Most modules also have their own .MSG files, and program modules have their own .DLL files which themselves may govern various data files.

Once installed, modules can be set to be Available or Disabled via the Basic Utility WGSDMOD (under choice **7** on the Introductory Menu). An Expressly Disabled module will not be allowed to operate until made Available again via WGSDMOD.

Once Available, modules can be further described as Active or Inactive.

A module is Inactive when, although Available, nothing calls it.

A module becomes Active when it is named in a module page in either Menu Tree (as configured by the Sysop in Design Menu Tree, choice **2** on the Introductory Menu).

A module can also become Active when it is named as a requirement in another module's .MDF file. When the other module becomes Active, the first module becomes Active. E-mail, for example, Requires the messaging engine in order to function.

Finally, a module can be set unconditionally Active. Language modules, for example, are set to become active as soon as The Major BBS server comes online, without need of outside prompting from either the Menu Trees or from other modules. Unconditional activation is set within the module's own .MDF file.

Module Definition Files: .MDF

Module Definition Files are how The Major BBS recognizes many aspects of your Add-on Option. They provide information on:

- Special installation programs to be run once then erased
- Once installed, what .DLL files to load, if any
- What .MSG files (CNF options) to include, if any
- Whether the module can be used in Menu Tree module pages
- Whether the module requires that other modules be active
- Whether the module replaces other modules
- Whether the module should be loaded regardless of other circumstances
- Document files to be integrated into WGSUSER.DOC, if any
- Btrieve database requirements
- Special processing at auto-cleanup, or at timed events
- Add-on utilities, if any
- Language information, if appropriate
- Custom editors for CNF text blocks and Menu Tree custom menus
- Whether other modules should be involved when downshifting from C/S mode to terminal mode
- Version Control for both client app and server agent files
- .EXE and support files involved in the client app
- The default icon file to use if the Sysop doesn't specify one in Menu Tree

For purposes of this book, we'll concentrate on the terminal-mode items. **Error! Reference source not found.** and **Error! Reference source not found.** cover the C/S-mode items.

For example, 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: GALREGTM.DOC

DLLs: GALREG
MSGs: GALREGIS

Btrieve page size: 1024
Btrieve files:    1

Cleanup:

Event-1:
Event-2:
Event-3:
Event-4:

Add-On Utility:

I need:

Agent version: 1.01
Client app version: 1.00
Client app EXE: GALRG100.EXE
Client app support files: GALRG100.HLP
Default Icon: GALREG.ICO
```

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

Comment Header

This is the name of the .MDF file. Note that a semicolon ; at the beginning of any line labels that line as a comment.

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 and in Basic Utility WGSDMOD. Use gmdnam() to read in this description — see page 25. The name may be up to 24 characters 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 your module replaces another, name the other module here. For example, Galacticomm's Entertainment Collection has an Entertainment Teleconference that replaces the Teleconference that comes with the standard version of The Major BBS. Note: the Requires and Replaces 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 filename, 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 is automatically deleted. The presence of this file acts as a flag to find out whether installation is needed or not.

Online user manual

You should write a brief but thorough help topic in a normal text file, aimed at the audience of terminal-mode users, and name this file here. The convention is to use the same name as the .MDF file with a .DOC extension.

It will be automatically combined with others to form WGSUSER.DOC. That file is (by default) attached to the welcoming e-mail message sent to new users. WGSUSER.DOC is also available through one of the default file pages in the Information Center menu page.

To provide help text geared to the C/S-mode side of your application, you'll create a Windows .HLP file referenced by the client app (the Windows .EXE file) you create. See *Client App Guide* for more details.

DLLs

Name the .DLL Dynamic Link Library (or Libraries, up to five of them) for your module. See page **Error! Bookmark not defined.**.

MSGs

Name the .MSG files that contain CNF options (for the Sysop to manage in Hardware Setup, Security & Accounting, Configuration Options, and Edit Text Blocks). More about CNF options starting on page 59.

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. If you have purchased Btrieve in order to create your own 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 Major BBS server is up.

Dynamic Btrieve Files

The Btrieve files line in your module's .MDF file is intended for the "core" (fixed) Btrieve files required by your module. If your module needs a variable number of additional Btrieve files, you can count them in the Dynamic Btrieve files line in your .MDF file.

If your module makes use of dynamic Btrieve files, it should police itself when online, making sure it doesn't use up more Btrieve file handles than it's entitled to. Call gnumdb() to get your .MDF file's current number of dynamic Btrieve files:

```
nbfalw=gnumdb("DDDXXX.MDF");
```

You'll need to supply an offline utility to update this line when adding or removing data files used by your module. You can use snumdb() to set this number:

```
snumdb("DDDXXX.MDF",nbfalw);
```

The server will pick up changes in the .MDF file, and rewrite BBSBTR.BAT as appropriate (combining the count of your module's core data files with its dynamic ones).

As it does with module names, INSTALL will maintain the current Dynamic Btrieve files lines when installing updates to .MDF files, while updating the standard Btrieve files line.

Cleanup

Name an .EXE or .BAT file to run when The Major BBS server shuts down for auto-cleanup. You can have multiple Cleanup: lines with multiple DOS commands. These commands are executed *before* The Major BBS runs the Sysop-configurable BBSCLEAN.BAT file.

Event-1 through Event-4

Name an .EXE or .BAT file to run when The Major BBS server shuts down for any of the timed events. These commands are executed *before* The Major BBS runs the Sysop-configurable BBSEVT?.BAT files.

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 parentheses. 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 usually want to make it also an UNCONDITIONAL module. If you don't, the only way this module can become active is if another (active) module Requires it.

I need, Needs me

If a system has terminal-mode module pages in its C/S-mode Menu Tree, C/S-mode users who select these pages will temporarily suspend C/S activities and drop into terminal mode. Common uses of this include the Menuing System (terminal mode's Top page), the Remote Operator Menu, and Doors.

The I need field in .MDF files lists which other modules are required in order to properly operate in this circumstance. The Needs me field activates this module whenever any of the modules named in this field are accessed in terminal mode by a user dropping down from C/S mode. See page 102 for more details.

Agent version, Client app..., and Default Icon

These C/S constructs are covered in ***Error! Reference source not found.*** and ***Error! Reference source not found.***. They have nothing to do with terminal mode.

Language .MDF Files

Here is an example of an .MDF file for the German/RIP Language:

```
; GERMRIP.MDF (language file created by WGSLANG.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
```

Notice that it does not declare a .DLL file. Most of a language module's purpose in life is to tell the text-block handling system that each **T** type CNF option may have a version in this language.

Language module .MDF files have a few additional constructs:

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:

.ASC	Standard ASCII
.ANS	ANSI colors and cursor control with standard ASCII
.IBM	ANSI colors and cursor control with IBM extended 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)
    else
        Use the .ANS file (or .IBM or .ASC as required)
else
    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 RIP*aint*.

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

Edit INFOMENU.IBM using WGSDRAW

Edit INFOMENU.ANS using WGSDRAW

Edit INFOMENU.ASC using WGSDRAW

Edit INFOMENU.RIP using RIP*aint*

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. WGSDRAW is the standard in-memory editor and is specified like this:

```
Language Editor:  WGSDRAW %s
```

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

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

When it comes time to edit an option or a menu, the %s is replaced with a filename and the above command is executed as a DOS command. We use a system() call on this command line, so DOS will look for ZOGEDIT.COM, ZOGEDIT.EXE, or ZOGEDIT.BAT 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 280 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 that 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 other universal meanings within The Major BBS (Help and Exit). Also, a yes-or-no word that starts with R will conflict with the highly visible 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 A, then cncyesno() returns Y. This way, your code can always check cncyesno() for Y or N return values. See page 86.

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* under_scores) as an initialization routine, and call that routine when The Major BBS server comes up. In your initialization routine, be sure to:

There must be exactly one init__ routine in every .DLL. The one routine will initialize the module for both terminal mode and C/S mode as necessary. See **Error! Reference source not found.** for more information on C/S-mode aspects of the initialization routine.

- Declare it as EXPORT (page 10).
- Register your module with register_module() (page 24).
- Open the CNF options file (.MCV in run-time) using opnmsg() (page 70). You can read in options using routines like numopt() and ynopt().
- Open any Btrieve databases you're using with opnbtv() (page 256).
- Declare how much if any of the Volatile Data Area that you'll need to service users, using dclvda() (page 49).
- Allocate memory if needed. Use the Volatile Data Area if at all possible.
- Register global commands with globalcmd() (page 114).
- Prepare enough VDA memory for any Full Screen Data Entry sessions using fsdroom() (page 122) and dclvda().
- Register text variables with register_textvar() (page 82).
- Kick off any rtkick() routines (page 291).

Modules

A module is the main mechanism for making your code run on The Major BBS while 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 Trees. Remember that the Sysop ultimately decides who uses each service and how the menus leading to it are structured.

Note: modules service *terminal-mode* users. **Error! Reference source not found.** describes how to create *agents* to service C/S-mode users (actually, to service client apps running on users' PCs). Before you get into that, though, you need to understand the concepts in this guide first.

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

- Put your initialization code in a routine whose name starts with init__ (see above).
- In that initialization code, register your module using the register_module() function and a unique module structure.
- Compile and link your module code into a .DLL file (page **Error! Bookmark not defined.**).
- Identify the .DLL file in your .MDF file (page 13).

For example:

```
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 {                      /* module interface block           */
     char descrp[MNMSIZ];            /*    description for main menu     */
     int (*lonrou)();                /*    user logon supplemental routine */
     int (*sttrou)();                /*    input routine if selected     */
     void (*stsrou)();               /*    status-input routine if selected */
     int (*injrou)();                /*    "injoth" routine for this module */
     int (*lofrou)();                /*    user logoff supplemental routine */
     void (*huprou)();               /*    hangup (lost carrier) routine  */
     void (*mcurou)();               /*    midnight cleanup routine      */
     void (*dlarou)();               /*    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            */
     "",                    /* name used to refer to this module */
     NULL,                  /* user logon supplemental routine   */
     colorinp,              /* 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            */
     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.

Note: if the Module name field in one .MDF file matches the Module name field in another .MDF file, The Major BBS will complain to the Sysop at startup, asking which module to delete or rename.

Naturally, DOS will not allow two identical filename.exts to reside in the same directory, so there will never be two .MDFs with the same filename installed on the same The Major BBS server. The INSTALL program will overwrite the existing file with the new file of the same name. If this is an update, this is desirable. Strict adherence to Developer-ID naming conventions will prevent accidental sharing of a single name between products of different developers.

Any of the entry points can be NULL if you don't need them, except: stsrou() (make it dfsthn 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 Major BBS server shuts down.

Variables Available to Many of the Entry Points

These global variables are set up by the executive before calling any of the entry points lonrou(), sttrou(), stsrou(), lofrou(), or huprou():

int usrnum;	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 extendible 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 268). For example, usaptr->userid is his User-ID, usaptr->usrpho is his phone number. This data is stored in a database.

Each user's session on The Major BBS server 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 logon activity in progress (using the lonrou() entry point)
	SUPLOF = 5	supplemental logoff activity in progress (using the lofrou() entry point)
	ACTUSR = 6	logged on using entry point sttrou()
	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 BBSPRV, SUPLON, SUPLOF, or ACTUSR.	
usrptr->substt	Substate 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.	

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 logon or logoff (more on that below).

int margc;	the number of separate arguments (words) in the user's input line (see page 82)
char *margv[];	table of pointers to the "words" of the user's input line
char *margn[];	table of pointers to the ends of the input words
int inplen;	total length of the input line in bytes
int pfnlvl;	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 50).

char *vdaptr;	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.

Logon Input Service Routine: lonrou()

You can use this entry point to give a user some kind of notice when he is logging on, such as "Our stock price rose 4 points yesterday". 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 logon message versus a series of logon prompts and responses) lies in the lonrou() return value:

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

To send one logon 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 logon 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 stsrou() entry point.

Note: to intercept the moment when a user first connects to The Major BBS server, use the handle-connect vector (*hdlcon)() as described further on page 94.

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.

Module Input Service Routine: sttrou()

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.

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 69, text blocks are identified first by their .MCV file (specified by setmbk()), and second 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 WGSMSX-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;
```

So 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 types the single letter **x** followed by ENTER, that he exit out of whatever he's doing and return to a previous menu. If your module has its own local menu (many do), then he returns to that menu. If he's already there, he should return to the parent menu page which originally brought him to your module.

We also recommend that whenever the user just presses ENTER (so that margc == 0), that you re-transmit the last prompt. See page 32 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 82) is a combination of three things:

- 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)
- The command string for the corresponding module page, if any (the Sysop specified this in offline Menu Tree Design)
- 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 Major BBS server executive) that the user is exiting your module back to the parent menu. See page 88 for more on whether to exit to a module's local menu or to the parent menu page.

Status Handler Routine: stsrou()

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 logon or logoff supplemental entry point), and the channel detects a status condition. Here are some of those status conditions (for more details, see *GSBL Guide*):

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 btuoes(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 dfsthn()). After processing, turn off status 5's with btuoes(usrnum,0). If you don't turn them off, 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 51.
251	Data input overflow (mostly harmless)
252	Echo buffer overflow
253	Data output overflow
254	Status input overflow (rather serious)
255	Command output overflow (rare)

Call dfsthn() (the default status handler) for status conditions your module is not specifically expecting. If there is no User-ID logged onto the current channel, dfsthn() may call you back, resulting in an infinite loop. See the status handling in FILEXFER.C for an example of the proper way to test against this condition.

Reprompting Routine: injrou()

In many cases, The Major BBS server 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 (page 80) is used to send asynchronous messages to a user's terminal, such as SYSTEM GOING DOWN 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 prfbuf 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,prfbuf);
btuoes(othusn,1);
user[othusn].flags¦=INJOIP;
return(1);
```

In fact, this is exactly the code in dftinj() that gets 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's 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 btuoes(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 prfbuf buffer contents. Those contents may be needed for output to other channels.

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

Note the use of othusn instead of usrnum. 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 user's 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:

- Don't use usrnum, use othusn
- Don't use usrptr->substt, use user[othusn].substt
- Don't use usaptr->userid, use uacoff(othusn)->userid
- 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 80).

Logoff Input Service Routine: lofrou()

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 logoff message versus a series of logoff 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 logoff 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 logoff 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.

User Disconnect Routine: huprou()

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 49). If NULL, no action is taken. This applies to all modules, not just the current one.

Auto-Cleanup Routine: mcurou()

The mcurou() entry point of each module is invoked once per day, (default time: 3:00 AM). 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 17.

The absdtdy() function exists to determine whether or not there's been an abnormal shutdown (non-catastro() crash) today:

```
if (absdtdy()) {
    longcup();
}
```

If the server locks up, GPs, or otherwise crashes without the system shutdown routines being called, the absdtdy() routine will return TRUE for the remainder of the day. After a successful cleanup, it will begin returning FALSE again.

You may want to make use of absdtdy() in order to do certain crash recovery in your cleanup code. This should be of particular interest to any modules that keep important data in memory and only save it at shutdown time. If things get out of sync due to crashes (message counts, for example), cleanup can get them back in sync again.

Delete User Account Routine: dlarou()

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 Major BBS server is deleted, the corresponding Registry entry should be deleted from the Registry database as well 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.

System Shutdown Routine: finrou()

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 54), 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, which for ease of management are grouped into up to 16 groups. Channels are numbered in hexadecimal from 00 to FF, while groups are numbered in decimal from 1 to 16.

Groups are defined in Hardware Setup through CNF options GROUP1 to GROUP16. The first channel number in each group is assigned under each GROUPn with the STARTn CNF option, and the number of channels in each group is assigned by the NUMBRn CNF option. See *Sysop's Guide* for The Major BBS. Channel numbers need not be assigned sequentially. For example, a Major BBS server 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 81.

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 GTOTHER  6       /* group type code:  GCDI 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

Structure	Lifetime	Access	See page
CNF options	Unlimited	Disk (direct)	59
Online User status and session info (`usrptr`)	User session	Memory	92
Online user detail (`usaptr`)	Unlimited	Memory	268
Offline user detail (BBSUSR.DAT)	Unlimited	Disk (indexed)	268
System variables (sv,sv2,sv3)	Unlimited	Memory	267

Data Structures For Use By A Specific Module

Structure	Lifetime	Access	See page
Volatile Data Area	Module active	Memory	49
`alcmem()` memory	From server up to server down	Memory	41
File opened using `fopen()`	Unlimited	Disk (sequential)	53
File opened using `opnbtv()`	Unlimited	Disk (indexed)	256

Examples Of Data Used By A Specific Module

Structure	Lifetime	Access
Electronic Mail message	21 days (default)	Disk (indexed)
Uploaded attachment to an e-mail message	14 days (default)	Disk (sequential)
Online user's Forum quickscan configuration	Unlimited	Memory
User logoff record	From server up to server down	Memory

Lifetime	Explanation
Unlimited	For the life of your hard disk (or until someone explicitly changes the information)
From server up to server down	While The Major BBS server is on-the-air: from `5 Go` to `Kill System`
User Session	While a user is online
Module active	Between the time a user selects a module from the Menu Tree, and exits that module

Access	Explanation
Memory	Access is fastest
Disk (direct)	Access requires reading only a sector or two
Disk (sequential)	Access as a serial stream of bytes
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:

- 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 93) for online users, or the BBSUSR.DAT database record for offline users (these have the same structure, see page 268). You can see how the module addcrd() in ACCOUNT.C makes this distinction when it adds credits to a user's account.
- 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 49 for more details.
- If you are developing a global command handler (page 114), 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 50).
- 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 Major 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 shutdown routine memcata() (see page 56).

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 x 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 x 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 times 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():

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

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:

```
bigregion=alctile(qty,sizblock);
                                Allocate a very large memory region,
                                qty by sizblock bytes
void *bigregion;                return value, for ptrtile() only
unsigned qty;          number of tiles
unsigned sizblock;     size of each tile

block=ptrtile(bigregion,unum);
                                Dereference an alctile() region
void *block;           pointer to a tile (offset always 0)
void *bigregion;                return value from original alctile()
unsigned unum;                  index, 0 to qty-1
```

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.

If you want to create an array that will dynamically increase in size, consider using the dynamic array API. It can manage arrays up to 64K in size, expanding them as needed. To create a dynamic array, just call newarr():

```
arrhdl=newarr(incsiz,elemsiz);
                                    create new dynamic array
int arrhdl;              handle to new dynamic array
int incsiz;              number of elements to grow by at a time
unsigned elemsiz;        size of each array element
```

To minimize memory fragmentation, the incsiz parameter specifies the number of elements to grow by each time the array needs to be resized. To add a new element to an array, call add2arr():

```
elemptr=add2arr(arrhdl,newelem);
                                    add element to dynamic array
void *elemptr;                      pointer to the new element
int arrhdl;              dynamic array handle returned by newarr()
void *newelem;                      data for new element (or NULL to init to 0's)
```

To get a pointer to an array element, call arrelem():

```
elemptr=arrelem(arrhdl,index);
                                    get pointer to element in array
void *elemptr;                      pointer to the requested element
int arrhdl;              dynamic array handle returned by newarr()
int index;               index of requested element
```

After calling add2arr(), an element may relocate in memory. So, you can't get a pointer to an element once and refer to it forever. Instead, call arrelem() to get the latest pointer whenever an add2arr() might have taken place.

To find out the number of elements currently in an array, call ninarr():

```
nelems=ninarr(arrhdl);          get # of elements in array
int nelems;                     number of elements in the array
int arrhdl;                     dynamic array handle returned by newarr()
```

Another API exists for dealing with pools of memory areas. Each pool contains a variable number of memory areas, each up to about 64K in size. To conserve memory, the pool API allows *x* total memory areas and *y* memory areas in memory at once. The API will handle swapping memory areas in and out of memory as needed. The Sysop can set the location of the swap file used with the ASFILE CNF option.

You can create a memory pool by calling newpool() at initialization time:

```
poolhdl=newpool(areasiz,nareas,ninmem);
                                create a new memory pool
int poolhdl;                    handle to new memory pool
unsigned areasiz;               size of memory areas within memory pool
int nareas;                     total number of memory areas within memory pool
int ninmem;                     number of memory areas to be in memory at once
```

The messaging engine (GME) uses the pool API for buffers to handle the writing of messages. By default, there's one per channel but only one for every eight channels can be in memory at once. That ratio is editable by the Sysop with the CHNPBUF CNF option.

The number of areas in memory relative to the total number is something that you will want to default differently based on the nature of what's stored in the areas and how often they'll be used. You should make the ratio configurable by the Sysop since every system makes use of software in a different way.

When you want to make use of a memory area, you need to reserve it. As long as it's reserved, its data is secure. You maintain the actual data, and the API handles swapping it in and out of memory as needed. When you're done with an area, you need to unreserve it so that it can be used again.

To reserve an area, call rsvarea():

```
areahdl=rsvarea(poolhdl);
                                        reserve a memory area
int areahdl;          handle to the area (or -1 if none available)
int poolhdl;          pool handle returned by newpool()
```

To get the pointer to a memory area you've previously reserved, call areaptr():

```
area=areaptr(poolhdl,areahdl);
                                        get pointer to memory area
int poolhdl;          pool handle returned by newpool()
int areahdl;          area handle returned by rsvarea()
```

The areaptr() routine will swap areas in and out of memory as necessary. Every time you call areaptr(), it invalidates anything previously returned by areaptr(). That's because, in getting the pointer to one area, it may very well have been necessary to swap out another area. So, you can't rely on pointers to areas remaining valid very long: just until the next areaptr() call. So, call areaptr() each time you need to get the pointer to an area.

To unreserve a memory area, call unrarea():

```
unrarea(poolhdl,areahdl);
                                        unreserve memory area
int poolhdl;          pool handle returned by newpool()
int areahdl;          area handle returned by rsvarea()
```

Once you unreserve a memory area, the data in it is lost as it becomes free to be reserved again. The ability to reserve and unreserve memory areas allows a pool of memory areas to be used in a very dynamic fashion. If your needs are more static, like one area per channel, then you can reserve all nterms areas at initialization time and never unreserve them.

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

```
movmem(source,destination,nbytes);
                                Move a block of memory
char *source;           source block
char *destination;      where to put it
unsigned nbytes;                number of bytes, 1 to 65535

setmem(destination,nbytes,value);
                                Set a block of memory to a value
char *destination;      pointer to the block
unsigned nbytes;                number of bytes, 1 to 65535
char value;             1-byte value or character

repmem(destination,pattern,nbyt);
                                Replicate a pattern in memory
void *destination;      where to put it
char *pattern;          NUL-terminated string
int nbyt;                       total number of bytes at destination
```

The repmem() function will replicate [nbyt/strlen(pattern)] 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.

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

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

```
memavl=sizmem();                Find out how much memory is available
long memavl;            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 23). 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()	logon message / line input during logon
lofrou()	logoff message / line input during logoff
stsrou()	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 29) 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 24) of the module whose huprou() entry point has been called. That is, huprou() may work with the VDA 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 server 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 in your module (using the above test), then huprou() may take appropriate steps to clean up any unfinished business in the VDA. 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 49) 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. 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. 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 vdaoff() 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. 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 response time in two ways:

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

Cycle Mediating

The trick here is to 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 *GSBL Guide* that the status code we use for cycle mediating, status 240, is reserved for application program use. In the source file MAJORBBS.H, the constant CYCLE is defined as 240.

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

line 1
line 2
line 3
line 4
Press ENTER (wait until user presses ENTER)
back to menu...

In practice, you would never split up such a small task, but it serves well as an example.

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

```
STATIC int
sttexm(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\nPress ENTER ");
               usrptr->substt=4;
               break;
          default:
               dfsthn();
               return;
          }
          outprf(usrnum);
     }
     else {
          dfsthn();
     }
}
```

Notes:

- 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.
- prf() is like printf(), except that the converted text goes into a buffer. outprf() transmits the contents of that buffer to a specific user.
- stsexm() calls dfsthn() (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 which is called regularly. The actual polling rate depends on system loading, but it can be very rapid.

begin_polling(unum,rouptr);	
	Turn on polling for this channel
int unum;	User number for the channel
void (*rouptr)(void));	Polling routine (no parameters, no return value)
stop_polling(unum);	Turn off polling for this channel
int unum;	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
FOPRWA	Read & Write in ASCII mode
FOPRWB	Read & 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
                              16 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 of CNF options (see page 56 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. Phrase Sysop-causable errors in terms comprehensible to Sysops and phrase errors that only a programming error could cause in terms most useful for programmers.

Often, rather than calling catastro(), you can just allow something unpleasant but isolated to 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 gets something predictable and not a wild memory write followed by a computer lock-up. Nothing is worse than an intermittent lock-up.

A non-Sysop user should never be able to trigger a catastro() — your customers’ The Major BBS servers would be vulnerable to hackers bent on sabotaging a system rather than in pillaging it.

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, opnbtv() (page 257), 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 the l2as() and spr() functions in the section beginning on page 287.

All catastro() messages are written to the text file CATASTRO.TXT with a time and date stamp... assuming, of course, that 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(), but instead 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/ANSIGerman/RIP	Tutorial/RIP	

The multilingual feature primarily allows different versions of user output to be defined for different languages. The Major BBS won't translate user input (e.g. menu selections and commands), however.

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 Major BBS expects different strings for **YES** and **NO**. This affects the operation of the cncyesno() 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 87).

When The Major BBS server comes up, it builds a list of the user languages that are defined and sets a few global variables:

nlingo	number of languages defined, always at least 1
clingo	language index, 0 to nlingo-1, for the current user
extptr->lingo	usually the same as clingo
extoff(n)->lingo	language index of user number n (where n is 0 to nterms-1)
languages[clingo]->name	name of the current user's language
languages[clingo]->desc	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 69.

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

ilingo=lngfnd(lngnam);	look up a language by its 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?");
lnglist(1);
lngfoot(1);
```

You'd be better to use prfmsg() (page 69) 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 101). 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 WGSMSX 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 *Sysop's Guide*. 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:

- A non-programmer Sysop can change numerous values, names, options, prompts, and messages that affect the operation of his The Major BBS system.
- 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 run-time — the .MCV files. These sections will help you create new CNF options and use them in The Major BBS.

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

CNF type TEXT options (text blocks) 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
LEVEL3	Security & Accounting
LEVEL4	Configuration Options
LEVEL6	Edit 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
LEVEL10	Internet Connectivity Option (ICO)
LEVEL30 : LEVEL45	Reserved for configuring the 16 databases of The Major Database
LEVEL95	Debugging Options
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
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 WGSMSX 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 Dialects (language subsets) in *Sysop's Guide*.

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 run-time. This, and the option name is the only information that WGSMSX takes from the .MSG file to create the run-time .MCV file. In the .MSG files:

} is represented as ~}

~ is represented as ~~

The number of characters in each version is limited by CNF option OUTBSZ (located in Configuration Options, choice **4** on the Introductory Menu). OUTBSZ 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 67 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 68. 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 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 show up as softkey options 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.

WGSDRAW, 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 located in Edit Text Blocks (choice **6** on the Introductory Menu), 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 server 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 server 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 server 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 option GEOSYNC 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. Its 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	language2
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 second example omits 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 WGSMSX with no arguments (this happens in WG.BAT). That checks all .MSG files and makes .MCV files out of them if their time and date disagree. (After WGSMSX 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 **Error! Bookmark not defined.**).

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

```
CD \WGSERV
WGSMSX <filename> -OSRC
```

where <filename>.MSG is the name of your editable .MSG file. This puts the .MCV file into \WGSERV and the .H file in \WGSERV\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.

WGSMSX has these alternative command syntaxes:

```
WGSMSX [-O<source directory prefix>]
WGSMSX @<list file> [-O<source directory prefix>]
WGSMSX <root filename> [-O<source directory prefix>]
WGSMSX <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 run-time. The Major BBS reads from these files at run-time 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 61. 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 28 for more on this.

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

This function is just like prf() (page 76), 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 57).

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 Edit Text Blocks. 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 78 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.

inimsg(maxsiz)	initialize the message buffer
unsigned maxsiz;	maximum number of bytes in any option

You'll only have to use inimsg() 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.

mbkptr=opnmsg(mcvfil);	open a new .MCV file
FILE *mbkptr;	.MCV file identifier
char *mcvfil;	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 clsmsg(). The same type of value is also stored in the global variable curmbk (see below).

setmbk(mbkptr);	set current .MCV file block pointer
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.

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 pointer 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 \WGSERV\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 WGSMSX utility.

getmsg() is called indirectly by prfmsg() (see page 69) 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 82).

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 82).

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 carriage return or paragraph boundary, and LF = soft carriage return).

clsmsg(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(). There's no absolute need to call this: the shutdown will close it if it has not already been explicitly closed.

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 value started with Y, 0 if not
int msgnum; message number (use option name)

This function gets the value of a type **B** CNF option (YES or NO).

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;             message number (use option name)
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 WGSMSX utility, page 69), and recompile all the source code that #includes this header file.

User Interface

This chapter deals primarily with the *terminal-mode* side of The Major BBS modules. Terminal mode by definition means that all processing occurs on the server PC, none on the user's PC. In C/S mode, user I/O issues are handled by client apps written in Visual Basic. A C/S agent merely offers an API to the client app. In terminal mode, however, there is no client app, so the code running on the server PC is expected to provide a UI, a user's interface, as well.

Even if you never intend to write terminal-mode access for your projects, it is worthwhile to at least skim this chapter, and to carefully read the final section beginning on page 102.

User Output: prf(), prfmsg()

prf(ctlstg,p1,p2,...,pn);
prfbuf-directed printf()-lookalike

char *ctlstg;	printf()-like control string
p1,p2,...,pn;	just like printf()'s parameters (note: no longs or floats)

Function prf() 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) that 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 CNF option OUTBSZ in Configuration Options (Level **4**) and is set by iniprf() at initialization time.

See page 246 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 69 about prfmsg(), the variation of prf() that reads text from an .MCV file. prfmsg() is used far more often in The Major BBS's code.

outprf(unum);	send prfbuf to a channel & clear
int unum;	user number

This function transmits the contents of the prfbuf buffer to a specific user. When unum is anything other than usrnum, you should probably use outmlt() (page 78). 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 independent 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 fallback alternative — see Dialects in *Sysop's Guide* for more details on language subsets).

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:

Monolingual	Multilingual
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(), stsrou(), lonrou(), or lofrou() entry point, and also before every polling routine (page 53).

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 allocated for each language. Here's how to get each buffer's address and pointer:

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

The language 0 version goes in the first of the prfbufs, which is prfbuf itself. If anyone is online with language 1 selected, then the language 1 version goes in the prfbuf 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, you might try this:

```
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 server:

got=injoth();	inject a message to another user implicit inputs: othusn ... channel # to inject to prfbuf ... message to be injected
int got;	1=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:

- the Teleconference page feature and the global **/P** command
- the Sysop send-message function (F2)
- notifying online users that they have received Electronic Mail
- 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:

Right way	Wrong way
`int unsave;`	`int unsave;`
:	:
`unsave=usrnum;`	`unsave=usrnum;`
`curusr(userno);`	`usrnum=userno;`
:	:
`curusr(unsave);`	`usrnum=unsave;`

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.

curusr(newunum); Change to a different user number

int newunum; new user number, 0 to nterms-1

Defining Text Variables

See the Editing Text Blocks chapter in *Sysop's Guide* 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 WGSDRAW.

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 *Sysop's Guide*.

User Input

On The Major BBS, each time a user types a string of characters and presses 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 29).

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 NULs)

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 NULs 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[]	points to RAIN IN SPAIN

You can still use margv[] pointers after rstrin(), but don't expect the same results as before. Although each margv[] still points to the same initial character in each word, the terminating NUL has been removed from all but the last. For example,

margv[0]	now points to RAIN IN SPAIN
margv[1]	now points to IN SPAIN
margv[2]	now points to 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 CNF option PFCEIL located in Configuration Options (level **4** on the Introductory Menu).

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 CNF option SECCHR located in Configuration Options (level **4** on the Introductory Menu). SECCHR defaults to * (asterisk).

Command Concatenation

This feature has two purposes on The Major BBS.

1. It allows the Sysop to define detailed subcommands within his online service. This comes up during Menu Tree design when the Sysop types in command strings for module pages that give users access to your module. *Sysop's Guide* has examples of these strings for The Major BBS's baseline modules.
2. Command concatenation allows an experienced user to type several commands at once. For example, **ERF** from a menu that offers E for E-mail means: E-mail / Read messages / starting at the first 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=morcnc(); is there any more command?	
char ch;	next character ('\0' if none)

The morcnc() 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;	optional '-' followed by decimal digits (no conversion — returns ASCII string)
wrd=cncwrd();	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;	always returns name with /
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:

user inputs:	**`cncyesno()` returns**
`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 cncchr() 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():

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

Source code of user input handler entry point:

```
int
sttqiz(void)
{
     int retcode=1;
     do {
          bgncnc();
          switch (usrptr->substt) {
          case 0:
               cncchr();           /* gobble the module select character */
               prf("\nQUIZ!\n");
               prf("What is the first letter of the alphabet? ");
               usrptr->substt=1;
               break;
          case 1:
               if (cncchr() == 'A') {
                    prf("How many fingers do you see?\n");
                    usrptr->substt=2;
               }
               else {
                    prf("\nThat's wrong!  You lose!\n");
                    cncall();
                    retcode=0;
               }
               break;
          case 2:
               if (morcnc() && cncint() == 0) {
                    prf("\nEND OF QUIZ!  You won!\n\n");
                    cncall();
                    retcode=0;
               }
               else {
                    prf("\nThat's wrong!  You lose!\n");
                    cncall();
                    retcode=0;
               }
               break;
          }
     } while (!endcnc());
     outprf(usrnum);
     return(retcode);
}
```

We've used prf() here instead of prfmsg() for simplicity. In practice we'd use prfmsg()s and put all text into a .MSG file (see page 59).

Exiting to the Parent Menu, or to your Module's Menu

condex();	conditional exit to parent menu for after handling concatenated commands

This routine can be used to return the user to the parent menu after the servicing of a string of concatenated commands that either came from the module's command string or from what the user typed.

To help handle these kind of situations, you may be able to make use of the CONCEX flag to give you fair warning of what condex() will do. Whenever a user enters a module from a Menu Tree menu, the (usrptr->flags&CONCEX) flag is:

- Set if the EXICNC CNF option is set to **YES** and the user concatenated two or more command characters together; or
- Cleared if EXICNC is **NO** or if he typed a single character.

It remains set (or cleared) throughout the user's activities in the module.

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.

User-ID Cross Referencing

When writing an Electronic Mail message, users can type in part of a User-ID and the server 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;                         return Code (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 uidxrf.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 clrxrf():

```
clrxrf();                       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 (press ENTER 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 (press ENTER for "E"): E} T Prompt asking for a vowel
```

You want your code to use that final character before the **}** curly brace to fill in for a user who doesn't pick any character and just presses ENTER. Here are the tools:

dftchr=getdft();	Get the default character and remove it from prfbuf
char dftchr;	
chkdft(dftchr);	Put the default character in the input buffer, if user just pressed 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 WGSDRAW 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 pressed 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

ison=uinsys(usrid);	determine if a user is online
int ison;	true if user anywhere online
char *usrid;	User-ID to be tested for
int uisusn;	global variable, set to user number when uinsys() returns 1

ison=onsys(usrid);	determine if a user is online
int ison;	true if user online & logged on
char *usrid;	User-ID to be tested for

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

- 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.
- uinsys() sets the global variable uisusn.
  onsys() sets othusn, othusp, and othuap (see below).

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

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

int othusn;	the user number of the other user
struct user *othusp;	pointer to structure for that user in the user[] array (see MAJORBBS.H)
struct extusr *othexp;	pointer to extendible in-memory structure for that user (see extoff() below)
struct usracc *othuap;	pointer to structure for that user in the usracc structure (see uacoff() below)

These variables are analogous to usrnum, usrptr, and usaptr, see page 26. To get the other user's 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 account 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 extendible in-memory info
int unum;	user number

For dealing with RIP support, the following exist:

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 Major BBS server, call byenow():

byenow(msgnum,p1,p2,...,pn);

	say goodbye 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 71)
TYPE p1,p2,...,pn;	parameters if any (max 12 bytes)

This routine will make reasonably sure that your goodbye 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;
curusr(othusn);
byenow(LASERCEPT);
curusr(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 Major BBS server using the (*hdlcon)() handle-connect vector:

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

When any channel — modem, serial, X.25 or LAN — establishes a 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 Major BBS server, 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 26?):

```
:
usrptr->class=BBSPRV;
usrptr->state=mystate;
usrptr->substt=CONTSTEP1;
usrptr->flags¦=NOGLOBS;
:
```

If you do this, then your sttrou() and stsrou() vectors will get called for future events (such as the user entering a line of text or a status condition on that channel). BBSPRV 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 a channel is in the BBSPRV class, it has the equivalent of MASTER status (also known as carrying the MASTER key), the ability to pass through any lock. It is absolutely mandatory to set the NOGLOBS flag, else the user can execute privileged global commands (**`/L SYSOP`**, for example) while you handle connect-time processing.

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 restore usrptr->class (and remember that this use of the word class is in no way related to user security Classes).

Whatever another 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 (*hdlnrg)(); 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 following the RING string. 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 The Major 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 *GSBL 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 Major BBS server 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 Major BBS server 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(usrnum,"Z");
     }
     else {
          while (nbytes--) {
               if (*incbuf++ == '!') {
                    zebra[usrnum]=1;
                    setbyprot("/ZEBRA",2);
                    return(1);
               }
          }
          if (snccon >= 16) {
               zebra[usrnum]=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 Major BBS server, 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, usrnum 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 snccon 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 (snccon == 0) case to initialize things if you need to. In every call after that, snccon 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 (zero 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 checking for a timeout. This way, if The Major BBS server 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 CNF option located in Configuration Options:

LANGOP=**ASK**	Display a numbered list of the languages that have the highest conf
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, more than one language is 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

*Terminal mode module logon routines are otherwise not called for C/S users.

Client/Server User Support

Client/server users are allowed to drop into terminal mode in order to use terminal-mode modules. When a user drops down to the terminal-mode main menu, all terminal-mode modules' logon routines are invoked as if an ordinary terminal-mode user had just logged on.*

The user then uses terminal mode as if he had made contact with the server through any other ASCII/ANSI terminal program. When he returns to C/S mode, all terminal-mode modules' hangup routines are called, as if an ordinary terminal-mode user had hung up.

The Sysop can specify that a C/S menu option should lead directly to a specific terminal mode module. Assuming the terminal mode module hasn't had its .MDF file updated for C/S support, all terminal mode logon routines (and eventually hangup routines) are called, as with the main menu above. After all of the logon routines are finished, the user is automatically put into the specific module.

If the server knows exactly what modules' logon/hangup routines need to be called in order to drop straight to a specific terminal-mode module, it can skip calling unnecessary logon/hangup routines, providing the user with a much smoother journey into the module. You can specify which other modules need to be initialized in order to go straight into a module with an I need line in its .MDF file:

```
I need: BBSFTF BBSFSD BBSFSE
```

This tells the server exactly which other modules to call logon/hangup routines for when dropping straight into your module. To specify that it doesn't need any other modules, specify a blank I need line:

```
I need:
```

As long as your module has no I need line, the server will be forced to assume it needs everyone, and treat your module as described above. You can also explicitly claim to need everyone with the following line:

```
I need: Everyone
```

To force your module to be needed by someone else (they don't know it, but they really do need you), use the Needs me line:

```
Needs me: GALTLC
```

You can also force yourself to be needed by everyone with the following line:

```
Needs me: Everyone
```

By default, your module should start out with a blank I need line. But, as you call upon services in other modules, you will need to include those other modules. The most popular modules to need are:

- BBSFSE for the FSE (full screen editor) services;
- BBSFTF for the FTF (file transfer facility) services; and
- BBSFSD for the FSD (full screen data entry) services.

Note: Of course, all of the standard Replaces logic is figured into arriving at what modules are really needed by what other ones.

You can always tell a C/S-mode user in disguise as a terminal-mode user by the WSGCSU flag:

```
if (!(usrptr->flags&WSGCSU)) {
    prfmsg(ANNOUN);
    outprf(usrnum);
}
```

You might want to make use of that knowledge in your logon routine to not output unnecessary announcements to C/S users dropping into terminal mode. To go even further, you might want to check the entstt field for such C/S users to see if they're dropping straight into a module or not:

```
if (!(usrptr->flags&WSGCSU) || usrptr->entstt == 0) {
    prfmsg(ANNOUN);
    outprf(usrnum);
}
```

With the above code, the logon message won't be output to C/S-mode users dropping straight into a module. Depending on the importance of your particular announcement message, this might be a very friendly thing for your module to do.

Although it's ok to use the entstt in your logon routine to consider whether or not to display a logon message, it's *not* ok to use it to decide whether or not to initialize your module for terminal-mode use. If entstt isn't 0 and it's not your module, it means that the user is going straight into another module. But, your module's logon routine is still being called for a reason. It's still fair game to be entered by the user or otherwise called upon. Otherwise, your logon routine wouldn't have been called at all.

User Services

Security (Locks & Keys)

As explained in *Sysop's Guide*, security on The Major BBS is controlled at the foundation level by Locks and Keys.

Keys are simply strings of from 3 to 15 ASCII characters, in most cases Sysop-editable. Each user has a class keyring and an individual keyring. Keyrings record the list of keys which the Sysop has assigned to that user, either by assigning the user to a particular class or by assigning the user a key on an individual basis.

Locks, meanwhile, are decision points written into the code. A lock examines each user who encounters it, testing to see if the user carries the key which it has been assigned to look for. If the user carries that key, the lock allows him to pass. If the user does not carry that key, the lock keeps the user from accessing whatever feature it is guarding. A feature can have one or zero locks.

Locks can be built into modules such as File Libraries and Forums, but most locks appear as CNF options located in Security & Accounting (choice **3** on the Introductory Menu).

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 Security & Accounting CNF 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 a Security & Accounting CNF option. See page 59 for creating CNF options. You could make a Security & Accounting CNF 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 {
     sorrynotachance();
}
```

By convention, all security-related CNF options are stored in level 3 — Security & Accounting. If you specify any locks of your own here, we recommend that you 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 Major BBS server

This routine checks if the current user has the specified key.

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

For quicker response, store the string in memory with stgopt() and use haskey() instead of reading it each time you need it with hasmkey().

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 othusn (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 92).

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 channel number he uses for his 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,rouputr);	
	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 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 under_score 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?	Subtract credits if user is *exempt* from credit charges?	Automatically borrow credits if user can go into debt?	User
`dedcrd()`	Yes	No	Yes	Current
`rdedcrd()`	Yes	Yes	No	Current
`odedcrd()`	Yes	Optional	Optional	Any Online
`ndedcrd()`	Yes	Optional	Optional	Any Offline
`ldedcrd()`	Yes	Optional	Optional	Any
`gdedcrd()`	Yes	Optional	Optional	Any
`tstcrd()`	No	No	Yes	Current
`rtstcrd()`	No	Yes	No	Current
`otstcrd()`	No	Optional	Optional	Any Online
`ntstcrd()`	No	Optional	Optional	Any Offline
`ltstcrd()`	No	Optional	Optional	Any
`gtstcrd()`	No	Optional	Optional	Any

The bolded routines are those you'll likely put to the most 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 account
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 account
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);
                        Deduct credits from an online account
int enuf;               1=had enough, 0=didn't
int unum;               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);
                        Deduct credits from an offline account
int enuf;               1=had enough, 0=didn't
char *userid;           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);
                                  Deduct credits from an "active" user account
                                  structure residing in memory
int enuf;                         1=had enough, 0=didn't
struct usracc *uptr;     pointer to active user structure
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=gdedcrd(userid,amount,real,asmuch);
                                  Deduct credits from any user's account
int enuf;                         1=had enough, 0=didn't
char *userid;            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=tstcrd(amount);
                                  Test if user has enough credits
int enuf;                         1=had enough, 0=didn't
long amount;             number of credits (don't deduct)

enuf=rtstcrd(amount);
                                  Test if user has enough real credits
int enuf;                         1=had enough, 0=didn't
long amount;             number of credits (don't deduct)
                                  (won't take debt or exemptions into account)

enuf=otstcrd(unum,amount,real);
                                  Test if user has enough credits
int enuf;                         1=had enough, 0=didn't
int unum;                         user number
long amount;             number of credits (don't deduct)
int real;                         1=don't take debt or exemptions into account
```

```
enuf=ntstcrd(userid,amount,real);
                              Test if offline user has enough credits
int enuf;                     1=had enough, 0=didn't
char *userid;           User-ID
long amount;            number of credits (don't deduct)
int real;                     1=don't take debt or exemptions into account

enuf=ltstcrd(uptr,amount,real);
                              Test if user account structure has enough credits
int enuf;                     1=had enough, 0=didn't
struct usracc *uptr;    pointer to active user structure
long amount;            number of credits (don't deduct)
int real;                     1=don't take debt or exemptions into account

enuf=gtstcrd(userid,amount,real);
                              Test if any user has enough credits
int enuf;                     1=had enough, 0=didn't
char *userid;           User-ID
long amount;            number of credits (don't deduct)
int real;                     1=don't take debt or exemptions into account
```

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 CNF option located in Security & Accounting).

Global Commands

Global commands are commands that users can enter from almost any prompt on The Major BBS. One exception: 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 82), 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);
}
```

Some check like margc == 1 is necessary because margv[0] is undefined when margc == 0. More generally, margv[n] is undefined when margc <= n. The sameas() check is case *in*sensitive 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). *Important: 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 can 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 23):

```
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 50 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:

Command	Purpose	Source code
/R <userid>	registry report	REGISTRY.C
/P <userid> <message>	page	MJRTLC.C or ENTTLC.C
/#	who's online	MAJORBBS.C or ENTTLC.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	MAJORBBS.C

The following commands are not registered global commands. These are special commands available from all menu pages:

Command	Purpose	Source code
FIND	Search menu pages	MENUING.C and MAJORBBS.C
DISABLE (Sysop only)	Disable a page	MENUING.C and MAJORBBS.C
ENABLE (Sysop only)	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 *Sysop's Guide* 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 developers, this distinction is transparent. The bgnedt() routine will make use of what The Major BBS server 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;                     max size of topic (including 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:

```
                            /* flags that can be passed to bgnedt()    */
#define ED_READON      2    /* "read only" mode                        */
#define ED_CLRTOP      4    /* clear topic buffer upon entry           */
#define ED_CLRTXT      8    /* 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. 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 VDA).

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.

edtimr(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 presses CTRL+N for New in the editor, he can import another message. 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).

An example of a Sysop Feedback Forum using the Full Screen Editor:

```
/***************************************************************************
*    GALFBK.C                                                              *
*                                                                          *
*    Copyright  (C) 1989-1995 GALACTICOMM, Inc.    All Rights Reserved.    *
*                                                                          *
*    Feedback to Sysop (sample module discussed in the                     *
*    Developer's Guide for The Major BBS)                                     *
*                                              - RNStein                   *
*                                                                          *
***************************************************************************/

#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            */
     "",                     /* name used to refer to this module */
     NULL,                   /* user logon supplemental routine   */
     fbkinp,                 /* 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            */
     fbkfin                  /* finish-up (sys shutdown) routine  */
};

#define TPCSIZ 40       /* maximum characters in topic       */
#define FBKSIZ 1921     /* maxchars 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 *)vdaptr)
```

```
void EXPORT
init__feedback()        /* initialize feedback stuff         */
{
      stzcpy(fbkmodule.descrp,gmdnam("GALFBK.MDF"),MNMSIZ);
      fbkstt=register_module(&fbkmodule);
      fbkmb=opnmsg("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->substt) {
          case 0:
               cncchr();
               prfmsg(HELLO);
               outprf(usrnum);
               bgnedt(FBKSIZ,fbkptr->text,
                      TPCSIZ,fbkptr->topic,fbkdun,ED_CLRTOP+ED_CLRTXT);
               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\n",
               usaptr->userid,ncdate(today()),nctime(now()),
               fbkptr->topic,fbkptr->text);
          fclose(fbkfp);
          prfmsg(THANKS,usaptr->userid);
          outprf(usrnum);
     }
     return(0);
}
```

```
STATIC void
fbkfin(void)                    /* feedback shutdown             */
{
     clsmsg(fbkmb);
}
```

Here are the Text Block CNF options which go with this example:

ESC represents the Escape character, ASCII 27.

```
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%sESC[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.

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 presses 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 standard system default status handler dfsthn(), 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 vdaptr 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 that 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 that 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:

- Display data
- Enter data, full-screen mode
- 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:

- Template (in .MSG file, level 99)
- Field specification string (usually in memory)
- Memory for the session's variable-length data structures
- Default answer string (usually created on the fly)
- Field-verification routine (optional)
- When-done routine (process answers, restore state/substate)
- Calls to FSDBBS.C routines

Procedure:

1. Create a Template in an .MSG file. (See UEDANSI{} in BBSSUP.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 if the template is for an entry session

```
nbytes=fsdroom(tmpmsg,fldspc,0);
```

but make a call like this if the template is for displaying:

```
nbytes=fsdroom(tmpmsg,fldspc,-1);
```

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 prfbuf. 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 prfbuf. This stuff must be untouched between the 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->substt codes, as well as handle the end of the session.

The title in fsdrhd() is only for smooth operation for RIP*scrip* 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(tmpmsg,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.

stg=fsdnan(fldno);	Get a field's answer
char *stg;	pointer to answer
int fldno;	field number 0 to N-1

fsdfxt(fldno,buffer,maxlen);	Store answer for field into buffer
int fldno;	field number 0 to N-1
char *buffer;	store the answer here
int maxlen;	don't use more than this many bytes

index=fsdord(fldno);	Find index of multiple choice answer. Returns -1 if the answer was not one of the ALT='s.
int index;	the index, 0 to N-1, for the answer according to the N possible ALT= alternate values for the field
int fldno;	field number 0 to N-1

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):

length=stranslen(answers);	Find length of an answer string
int length;	length including final double \0.
char *answers;	answer string

value=fsdxan(answers,name);

Get the value of a field in an answer string, returning "" if not found.

char *value; pointer to answer string value

char *answers; answer string

char *name; name of field

fsdpan(answers,name,value);

Put a new field and/or value into an answer string.

char *answers; answer string

char *name; name of field

char *value; pointer to answer string's new value

fsddan(); Delete the answer just found by fsdxan()

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 23), then here are the steps 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 filename in ftfscb->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 later).

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 upload protocols: **?** or ""

To validate an upload protocol code, you could use valupc():

```
ok=valupc(prot);            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 filename 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 vdaptr. 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 things, 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 filename, 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 filename and size, if the protocol has supplied them. The file time and date may be in one of three forms:

Protocol provides:	`ftfscb->dosdat,dostim`	`ftfscb->unxtim`
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 293 about time and date formats and handling routines. See page 297 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:

- Does the filename have the proper syntax?
- Does the filename conflict with a reserved name? (For example, CON.TXT is an alias for the main console.
- Does this user have permission to upload this file?
- Does this user have permission to overwrite an existing file?
- Are too many users opening files at once (thereby using up all file handles)?
- Will users be able to use up all available disk space?
- Will users be able to upload a very large number of small files, making directory access very slow?
- If charges are associated with upload, can the user afford to pay?
- Could this filename possibly conflict with one of the other online users who are also using your application?
- 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 Major BBS server'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 CNF option FIMLOCK located in Security & Accounting, which requires the SYSOP key by default.

Here's the situation if I ever happen to get around to calling the FUPREF exit point: The user chose 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 1, 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.

In your scheme for referencing a file where it stands, you'll have to decide what you want to do about the standard upload file. The standard upload file is where you would have stored the file's contents if it had been uploaded by more conventional means, such as ZMODEM. When referencing a file where it stands, the contents are really stored somewhere else on disk. You may or may not want to store something in the standard upload file. For example, when e-mail attachments are by reference, the special FILIND flag is set in the message record (see GME.H), and the path of the referenced file is stored in the standard upload file.

So your FUPBEG exit point will always specify the file path name for the standard upload file (whether the file is uploaded conventionally or referenced where it stands), and that exit point must return 1 for the upload to complete. I will create the standard upload file and leave it open while I call your FUPREF exit point.

If you have a use for the standard upload file in your file-reference scheme, your FUPREF exit point is a good time to write to it: it's already open and you don't have to close it (you can use ftfrwr()). If you don't want this file, then unlink() it in your FUPEND exit point. Don't unlink() it in the FUPREF() exit point because the file is still open at that time, and will be closed later.

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 filename 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	**You want to return to the parent menu**
Restore your `usrptr->state`	Restore your `usrptr->state`
Restore your `usrptr->substt`	Say bye to the user if you wish (you don't need to call `outprf()`)
Prompt the user (you don't need to call `outprf()`)	Return 0
Return 1	

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 128 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
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 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 condex(), and possibly wind up returning control to your parent Menu Tree menu after all. 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 88 about this whole condex() 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.

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 35) 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 onto a Major BBS server. 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. Its 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) 1995 GALACTICOMM, Inc.    All Rights Reserved.       *
*                                                                          *
*       Uploading example.                                                 *
*                                                                          *
*                                               - R. Stein                 *
*                                                                          *
***************************************************************************/

#include "gcomm.h"
#include "majorbbs.h"
#include "filexfer.h"

STATIC int uplinp(void);
STATIC int fupupl(int fupcod);

int uplstt;                           /* Uploading module state number   */
struct module uplmodule={"",NULL,uplinp,dfsthn};

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(strcpy(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 Major BBS server computer. 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 filename. 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, dfsthn() is the stsrou() 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 filename. If the user just presses ENTER, he is returned to the parent menu without uploading. If he types in a filename, 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 128. 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 restores 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:

- Limits on file size and quantity in the upload directory
- Checking for conflicts between the filename and DOS devices
- Deleting the fragment left behind from an aborted upload
- Multiple-file uploads
- Sysop-configurable prompts in an .MSG file
- Sysop-configurable upload directory
- Formal declaration and limitation on VDA usage
- Command concatenation
- Automatic reprompt after `/p` page command, etc.
- 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) 1995 GALACTICOMM, Inc.    All Rights Reserved.       *
 *                                                                         *
 *      Uploading example II                                               *
 *                                                                         *
 *                                          - R. Stein                     *
 *                                                                         *
 ***************************************************************************/


#include "gcomm.h"
#include "majorbbs.h"
#include "filexfer.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;                        /* Enhanded 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 uplfmax;                      /* 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             */
     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                */
     uplfin                        /* finish-up (sys shutdown) routine      */
};

void EXPORT
init__uploader(void)               /* Uploader initialization               */
{
     stzcpy(uplmodule.descrp,gmdnam("GALUPX2.MDF"),MNMSIZ);
     uplstt=register_module(&uplmodule);
     uplmb=opnmsg("GALUPX2.MCV");
     mkdir(spr("%s.",upldir=stgopt(UPLDIR)));
     uplbmax=lngopt(UPLBMAX,0,2147483647L);
     uplfmax=numopt(UPLFMAX,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 {
          bgncnc();
          switch (usrptr->substt) {
          case 0:
               cncchr();
               prfmsg(usrptr->substt=UPLNAME);
               break;
          case UPLNAME:
               if (usrptr->flags&INJOIP) {
                    prfmsg(UPLNAME);
                    break;
               }
               cncall();
               parsin();
               switch (margc) {
               case 0:
                    uplfil("","?");
                    break;
               case 1:
                    uplfil(margv[0],"?");
                    break;
               default:
                    uplfil(margv[0],margv[1]);
               }
               break;
          }
     } while (!endcnc());
     outprf(usrnum);y
     return(1);
}

STATIC void
uplfil(                             /* upload file(s)                        */
char *filnam,                       /* filename, 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,"Filename is a reserved DOS device name.");
               break;
          }
          cntdir(spr("%s*.*",upldir));
          if (numfils >= uplfmax) {
               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              */
{
     clsmsg(uplmb);
}
```

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)

UPLDIR {UPLDIR\} S 0 Upload directory:

 This is the maximum number of files allowed in the
 upload directory.

UPLFMAX {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 {}
```

ESC represents the Escape character, ASCII 27.

```
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 filename, please choose another name.
} T Upload example II filename 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 Galacticomm Demo system in the file GALUPX2.ZIP.

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 filename.

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 filename prompt. The user can respond by pressing `*` or ENTER 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 85). These are passed to an internal function, uplfil(), with default values for protocol and filename 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 filename for dangerous characters like colon (`:`), backslash (`\`) and twin periods (`..`). There are many other (less dangerous) illegal characters for filenames, but most of these are caught eventually when FTF tries to create the file. If the filename is safe, it's stored in the VDA and fileup() is called. Notice that the filename 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 meg 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 Major BBS server shuts down.

ASCII Downloads

To dump an ASCII file to the user's terminal, you can use:

listing(path,whndun);	list an ASCII text file to the user's terminal
char *path;	DOS path of the file (must be a permanent storage location)
void (*whndun)(all);	restore state & substate, prompt the user for what to do next
int all;	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 filename, 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 147).

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 tagspecs 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 CNF option MAXTAGS in Configuration Options. 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:

usurp=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 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 filename 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 filename 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 128 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(ftfscb->fname,"<filename 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 in file TSHXXX.C 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:

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
struct ftg *ftgptr;	current tag table entry
ftgptr->tagspc	current tagspec
char *tshmsg;	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 CNF option SRETRYV in Edit Text Blocks).

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 tagspecs 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 tagspecs (which are not now checked for visibility, but may be in future versions) as well as single-file tagspecs. 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 tagspecs 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 160 for a way to do TSHSCN and TSHNXT with fnd1st() and fndnxt(). See page 300 about the routines themselves. For your convenience, ftuptr->fb is a user-specific fndblk structure that you can use with fnd1st() and fndnxt() 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:

- Does the filename have the proper syntax?
- Does the filename conflict with a reserved name?
- Does the user have permission to download this file?
- Are too many users opening files at once (thereby using up all file handles)?
- Can this user afford the download charges, if any?
- 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 filenames, put the filename into ftfscb->fname. For example, if downloading with ZMODEM, this filename will be the one used on the user's terminal. It doesn't have to be the same as the one used on The Major BBS server, but it usually is.

There is another special option in the TSHBEG exit point. You can return -1 to indicate that 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. Two alternatives:

You want control back	**You want to return to the parent menu**
Restore your `usrptr->state`	Restore your `usrptr->state`
Restore your `usrptr->substt`	Say goodbye to the user if you wish (you don't need to call `outprf()`)
Prompt the user (you don't need to call `outprf()`)	Return 0
Return 1	

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 151 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 after all. 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 88 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 136 about making sure your module's cleanup code executes exactly once for users in your module.

Downloading Example #1

```
/***************************************************************************
 *                                                                         *
 *   GALDNX.C                                                              *
 *                                                                         *
 *   Copyright (C) 1995 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,dnlinp,dfsthn,NULL,NULL,NULL,NULL,NULL,NULL};

void EXPORT
init__downloader(void)               /* Downloader initialization       */
{
     stzcpy(dnlmodule.descrp,gmdnam("GALDNX.MDF"),MNMSIZ);
     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],TSLENG);
               ftgptr->tshndl=tshdnl;
               ftgsbm("?");
          }
     }
     outprf(usrnum);
     return(rc);
}
```

```
int
tshdnl(                              /* Handle the application-specific */
int tshcod)                          /* 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 module lets users download files from subdirectory DNLDIR on The Major BBS server computer. Users of this service type in a filename (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 filename. If he presses ENTER, 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 filename, 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-taggable 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:

- Viewing the contents of compressed files, such as .ZIP files
- Multiple-file downloads
- Checking for conflicts between the filename and DOS devices
- Checking for attempts to download files like DNLLIB\..\WG.BAT
- File tagging
- Concatenated commands
- Allowing X to exit
- Telling the user what files are available in the directory
- Sysop-configurable prompts an .MSG file
- Sysop-configurable download directory
- 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) 1995 GALACTICOMM, Inc.    All Rights Reserved.          *
 *                                                                         *
 *   Downloading example.                                                  *
 *                                                                         *
 *                                            - R. Stein  12/6/93          *
 *                                                                         *
 ***************************************************************************/

#include "gcomm.h"
#include "majorbbs.h"
#include "filexfer.h"
#include "galdnx2.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 *dnldir;                      /* 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,gmdnam("GALDNX2.MDF"),MNMSIZ);
     dnlstt=register_module(&dnlmodule);
     dnlmb=opnmsg("GALDNX2.MCV");
     dnldir=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->substt) {
          case 0:
               cncchr();
               if (!fllist()) {
                    cncall();
                    rc=0;
               }
               prfmsg(usrptr->substt=FLNAME);
               break;
```

```
          case FLNAME:
               cncall();
               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],TSLENG);
                    ftgptr->tshndl=tshdnl;
                    ftgptr->flags=FTGABL;
                    if (strchr(margv[0],'?') != NULL
                     || strchr(margv[0],'*') != NULL) {
                         ftgptr->flags|=FTGWLD;
                    }
                    rc=ftgsbm(margc > 1 ? margv[1] : "?");
               }
          }
     } while (!endcnc());
     outprf(usrnum);
     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),ncdate(fb.date),nctime(fb.time));
     } while (fndnxt(&fb));
     return(1);
}
```

```
int
tshdnl(                              /* Handle the application-specific */
int tshcod)                          /* 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=dnlstt;
     }
     return(rc);
}

STATIC void
dnlfin(void)                         /* Finalize downloading example    */
{
     clsmsg(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%-12.12sESC[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 filename prompt

FLRSV {ESC[0;1;35m
That's not a valid filename
} T Download example II filename invalid
```

ESC represents the Escape character, ASCII 27.

This code, plus support files, is available for download on the Galacticomm Demo system in the file GALDNX2.ZIP.

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
**B0000000000000
```

(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 filename.

An entry in the tag table is obtained, if available, by ftgnew(). In this example, the use of wildcard characters **?** or ***** in the filename 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 (CNF option OUTBSZ in Hardware Setup) 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.

Exit points TSHSCN and TSHNXT use the classic technique for breaking down multi-file tagspecs into single files: fnd1st() and fndnxt(). In each case a single-file tagspec (the filename) 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, 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,
     "C",                          /* 1-3 code letters for 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 */
     xyxini,      /* .initze()    Initialize this protocol (recompute scblen) */
     xcxsrt,      /* .start()                               Start a transfer */
     xyxctn,      /* .contin()              Continuously call, 1=more, 0=done */
     xyxinc,      /* .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 */
     ftfinbc,     /* .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}
};
```

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.

Next, register the structure in your init__xxx() routine via ftplog():

```
ftplog(fptr);
```

where fptr is a pointer to the structure from step 1. The new protocol will appear in the appropriate lists with all the other protocols.

The Galacticomm Messaging Engine

The E-mail and Forums messaging system is divided into low-level and user interface subsections. The low-level section handles the details of storing and retrieving messages, forum management, access, credit charges, importing and exporting messages, etc. It is referred to as the Galacticomm Messaging Engine or GME.

The engine provides an API for messaging functions. The purpose of this API is to isolate developers from the details of the low-level architecture, provide a standard method for using the messaging system, increase system efficiency by cycling intensive tasks, and ensure backward compatibility for future messaging system enhancements. All functions and variables in this API are declared in the file GME.H. In order to use this API, #include "gme.h" in your source files, and link to GMEIMP.LIB.

Access and Credit Charges

Most messaging access restrictions and credit charges are applied by GME. It is also possible to circumvent these restrictions in many cases, particularly when writing messages.

Backwards Compatibility

While the vast majority of the old messaging system (for The Major BBS Version 6) can no longer be supported, a small suite of backward compatible functions are provided. These functions mimic the only previously existing semblance of an API: sendmsg(), sigtpc(), saxxok(), uidxst(), and the msgbyts variable.

Fundamental Architecture

GME Requests

Simple processes can usually be handled by a single function call. However, many messaging tasks are complex processes requiring multiple function calls and/or cycles to complete. To achieve this functionality, a request system is used.

A request is a series of function calls, or multiple calls to the same function, that accomplishes a given task (e.g., writing a message). The tie that binds this series of function calls together is the GME work area. This is a fixed size memory buffer that is dedicated for use by the engine to track the state of a request. A pointer to this work area is passed to each function that is part of the request. The size of the work area is declared in GME.H:

```
#define GMEWRKSZ 512              /* memory required for GME work area    */
```

Since a request may span multiple cycles, the work area must reside in a memory area that will not be accessed by other processes (usually VDA for terminal-mode modules and per-request memory for C/S-mode modules).

Note: While it is possible to determine the structure of this work area, its contents should be considered undocumented (or, at the very least, subject to change without notice) and should never be directly examined or altered by any processes outside the messaging engine.

Each request must be opened by initializing the work area. Then the series of function calls that make up the request are made. When the request is finished (either normally or due to an error), some types of requests will automatically close the request, while others require that the request be explicitly closed. If the request must be aborted before completion, it must be explicitly closed. This will free up any GME resources that have been dedicated to the request.

The functions that initialize and close (or abort) a request are:

```
void
inigmerq(                         /* initialize GME request             */
void *workb);                     /*   work area to be used for request */

void
clsgmerq(                         /* close a GME request                */
void *workb);                     /*   work area being used for request */
```

It is important that the work area to be used for a request be opened once and closed once (in the same way that a file should be opened and closed). If a work area is used for one request then used for another request without being closed first, or if an uninitialized work area is used for a request, the results are undefined and possibly damaging. Before initializing a work area for the first time, it is a good idea to setmem() it to zero. After the initial setmem(), it is best to let inigmerq() and clsgmerq() do the work.

There is a function provided to determine whether a work area is in use:

```
BOOL
gmerqopn(                         /* is this request open?              */
void *workb);                     /*   pointer to work area             */
```

However, it is not 100% reliable. In general, you should track the status of your work area(s) yourself to be sure each is opened and closed properly.

The status of a request can be determined after each GME function call by the return code of the function call. A set of standard status codes are used by GME functions. These codes follow a general pattern:

code > 0	successful completion of call to this function
code < 0	error occurred during this function call
code == 0	do a cycle then call this function again

In addition to this general pattern, the status codes specify the nature of any errors or any special conditions of a successful completion. The following status codes are currently defined:

```
                                  /* GME request status codes           */
#define GMEAGAIN 0                /*   GME still processing request     */
#define GMEOK    1                /*   generic "finished OK" status     */
#define GMEAFWD  2                /*   message was auto-forwarded       */
#define GMERRG   3                /*   return receipt generated         */
#define GMEERR  -1                /*   generic "error" status           */
#define GMEDUP  -2                /*   unique name or ID exists         */
#define GME2MFR -3                /*   too many forums error            */
#define GMENFID -4                /*   no available forum IDs error     */
#define GMENFND -5                /*   generic "not found" error        */
#define GMENDEL -6                /*   generic "can't delete" error     */
#define GMENMOD -7                /*   generic "can't modify" error     */
#define GMEFDV  -8                /*   variable part of forum def too long*/
#define GMECRD  -9                /*   not enough credits for operation  */
#define GMEMEM  -10               /*   not enough memory for operation   */
#define GMEIVA  -11               /*   invalid attachment to message     */
#define GMENOAT -12               /*   unable to attach file to message  */
#define GMEACC  -13               /*   user doesn't have access          */
#define GMENRGM -14               /*   couldn't re-get message           */
#define GMENCFL -15               /*   can't copy/fwd to a dist list     */
```

```
#define GMERST  -16              /*   E-mail message pointer reset      */
#define GMEUSE  -17              /*   message in use, can't modify or del*/
#define GMENAPV -18              /*   attachment not approved for dnload */
#define GME2MFL -19              /*   too many forum data files error    */
#define GMENSRC -20              /*   source not found                   */
#define GMENDST -21              /*   destination not found              */
```

The codes that may be returned by any given function will be discussed in later sections.

Many of the key request-oriented functions may need to be called multiple times to complete their task. This somewhat unusual mode of operation is to reduce system loading by breaking up complex tasks across multiple cycles. The GMEAGAIN status code is the key to their use: as long as the function returns the GMEAGAIN status code, you must continue calling it with the same parameters. Since the parameters to such functions must retain their contents across cycles, they must be in non-shared memory areas (such as VDA for terminal-mode modules). Any exceptions to this rule will be noted in subsequent sections.

A typical use of one of these functions might look something like this:

```
void
dosomething(void)                /* process a GME request                */
{
     int status;

     switch (status=gmefunc(mywrkbuf,myparm1,myparm2,...)) {
     case GMEAGAIN:
          /* do a cycle then call dosomething() again */
          break;
     default:
          if (status > GMEAGAIN) {
               /* everything's OK */
          }
          else {
               /* an error occurred */
          }
          break;
     }
}
```

GME Callbacks

Many things can be occurring while a GME function is being cycled (especially when writing a message, see page 181). To allow the caller to track the progress of particularly complex requests, a callback mechanism is provided. A callback handler may be associated with each (initialized) work area using the following function call:

```
void
setgmecb(                          /* set handler for GME status reports   */
void *workb,                       /*   work area being used for request   */
void (*callback)(int evt,int res));/*   pointer to callback handler        */
```

Whenever a significant operation starts or ends, the engine will call the callback handler associated with the current request (if any), passing it an event code and a result code. The event code tells the callback handler what event has taken place. The result code is the GME status code that best describes the result of the operation.

The following event codes are currently defined:

```
                                   /* callback event codes                 */
#define EVTSTRT  0                 /*   starting to send a message         */
#define EVTDONE  1                 /*   done sending a message             */
#define EVTCCST  2                 /*   starting to send carbon copies     */
#define EVTDSTS  3                 /*   starting distribution              */
#define EVTDSTD  4                 /*   done with distribution             */
#define EVTCPYS  5                 /*   starting to copy a file            */
#define EVTCPYD  6                 /*   done copying a file                */
#define EVTNEWF  7                 /*   starting to scan a new forum       */
#define EVTNEWM  8                 /*   starting to scan a new message     */
```

Note that the event codes are generally grouped into “starting an operation” and “done with an operation” pairs. In general, the result code is only meaningful during a done event.

Within the callback handler, you may examine the contents of any buffers you passed to the engine (such as a message header buffer passed to a send-message request). You may also wish to examine the extended information string about the event that can be gotten by calling this function:

```
char *
gmexinf(void);                     /* get extended return information      */
```

The exact contents of this string depend on the request in progress and the event, and will be discussed in later sections.

Since callbacks occur while the engine is processing a request, they should do very little processing of their own — nothing more intensive than a prfmsg() and outprf() or a senddpk() should be done (be sure to do a rstmbk(), of course). In particular, you should never kick off another GME request from within a callback handler.

Addresses

Routing of messages when writing, copying, or forwarding is governed by two parameters: the forum ID and the address. For forum messages, the forum ID is the primary determining factor of where the message goes, and the address is far less important. However, for e-mail messages, the address is *the* determining factor of where the message goes.

Four distinct types of addresses are supported by GME:

Local Address	a User-ID (e.g., Sysop). This sends an e-mail message To: the User-ID specified.
Forum Address	the name of a forum preceded by / (e.g., /Hello). This sends the message To: ** ALL ** in the named forum
Exported Address	an exporter prefix followed by an address in that exporter's format (e.g., IN:TStryker@gcomm.com). This submits the message to the specified exporter which is then responsible for routing the message to the addressee.
Distribution List	an @ or ! followed by the list name (e.g., @STAFF). This will send a copy of the message to each valid address in the list. An @ denotes a Sysop-defined list. Currently the only ! lists are !QUICK and !MASS

The following functions and macros are available to distinguish between different types of addresses:

```
#define FORIDC   '/'              /* E-mail directed to a forum prefix    */
#define isforum(s) (*(s) == FORIDC)/* is this a forum address             */
#define isdlst(s) (*(s) == '!' || *(s) == '@')/* is this distribution list */

BOOL
islocal(                          /* is this a valid local addr (User-ID) */
char *adr);                       /*   address to check                   */
BOOL
isexpa(                           /* is this an exporter-style address    */
char *adr);                       /*   address to check                   */
```

Exporters and distribution lists are discussed in detail in later sections.

Messages

Messages consist of a header part and a text or body part. The message header is a structure that contains all the fields necessary for proper control and routing of the message, and other fields describing the message. The header structure is as follows:

```
#define MAXADR   256              /* message address size              */
#define TPCSIZ   51               /* message (and forum) topic size    */
#define HSTSIZ   57               /* message history size              */
#define FLNSIZ   13               /* DOS filename buffer size          */

struct message {                  /* in-memory message structure       */
     unsigned forum;              /*   ID of forum message belongs to  */
     long msgid;                  /*   unique message identifier       */
     struct globid gmid;          /*   unique global message identifier  */
     long thrid;                  /*   ID of thread message is part of */
     char from[MAXADR];           /*   originator                      */
     char to[MAXADR];             /*   recipient                       */
     char topic[TPCSIZ];          /*   main topic, editable, carried over */
     char history[HSTSIZ];        /*   history/routing (reply to #88888,  */
                                  /*     fw by Aaaaaaaaa, cc: of #99999)  */
     char attname[FLNSIZ];        /*   attached filename               */
     unsigned crdate;             /*   message creation date           */
     unsigned crtime;             /*   message creation time of day    */
     struct globid rplto;         /*   message that this is a reply to */
     int nrpl;                    /*   number of times replied to      */
     long flags;                  /*   message/attachment flag bits    */
};
```

The most important control fields are the forum ID and the message ID. These two identifiers are both required to find any message on the system.

The forum ID (forum) is a number that uniquely identifies each forum on the system. It is assigned by the engine when a forum is created. The forum ID is used to specify which forum a message resides in. A special forum ID is provided for E-mail:

```
#define EMLID    0U               /* E-mail destination ID             */
```

The message ID (msgid) is a number that uniquely identifies each message on the system. It is also assigned by the engine when a message is created.

You can get the highest message ID currently present on the system by calling:

```
long
himsgid(void);                    /* get current highest message ID    */
```

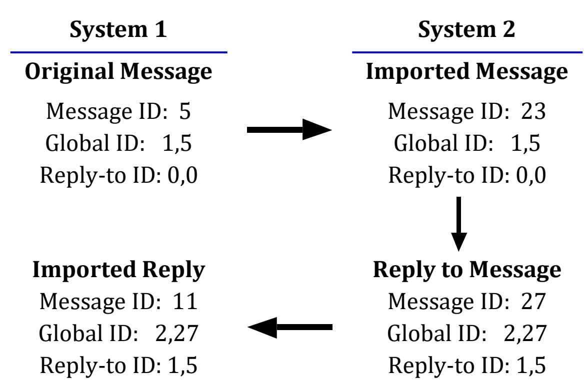
Inter-system threading in forums is supported by three fields: thread ID (thrid), global ID (gmid), and reply-to ID (rplto). The thread ID uniquely identifies a thread within a forum (including all systems participating in an echo). It is assigned by the engine whenever a new message is created, and is currently implemented as a 32-bit CRC of the topic of the message that initiates the thread. Replies to messages in a thread are assigned the same thread ID as the message to which they are a reply. Reading messages in a thread is equivalent to reading messages in a forum with the same thread ID.

In addition to the thread ID, the global ID and reply-to ID are used to maintain the parent-child relationship between messages across systems. In addition to a unique message ID, each message is assigned a unique global ID when it is created. The global ID consists of the system ID (an integer representation of the 4-character C/S system ID) of the system on which the message originated and its message ID on that system:

```
struct globid {                    /* global message identifier          */
     long sysid;                   /*   system ID of originating system   */
     long msgid;                   /*   message ID on originating system  */
};
```

When an echoed forum message is imported into a system, it is assigned a message ID unique to that system, but it retains its original global ID. When a user replies to a message in a forum, the new message is assigned the same thread ID as the original message, *and* the global ID of the original message is used as the reply-to ID of the new message. When a thread-to-parent is requested, the system examines the reply-to ID of the current message and looks for a message with that global ID.

To see how this works, consider the following example:



A message is written on system 1 and echoed to system 2. On system 2 a user replies to this message and the reply is echoed back to system 1. Since the imported reply references the global ID of the original message in its reply-to ID field, a user on system 1 reading the imported reply can correctly thread to its parent.

The from field is the User-ID or address of the person who sent the message. It is also the address to which e-mail replies to the message are sent.

The to field is the User-ID or address of the person to whom the message was or is to be sent.

The attname field is primarily for display purposes. If the message has an attachment, this field should contain a DOS filename by which the attachment will be known. When a user reads the message, this name will be displayed as the filename of the attachment. When a user downloads the attachment, this name will be given to the file transfer protocol as the suggested name for the file.

The history field holds text describing the history of the message (if it is a reply to another message, if it was forwarded by another user, etc.).

The flags field is used to store Boolean information about the message (e.g., is there a file attached, is a return receipt requested, etc.). It is divided into two separate sections. The high order word is used for GME-internal flags. The following flags are currently defined:

```
                                  /* message flag bits                   */
#define PRIMSG  0x00000001L       /*   message is "priority"             */
#define EXEMPT  0x00000002L       /*   message is exempt from auto-delete */
#define RECREQ  0x00000010L       /*   return-receipt requested when read */
#define FILIND  0x00000020L       /*   "indirect" att, direct has filespec*/
#define FILATT  0x00000040L       /*   file is attached to this message   */
#define FILAPV  0x00000080L       /*   file attached is ok to download    */
#define NODEL   0x00000100L       /*   sender can't delete                */
#define NOMOD   0x00000200L       /*   sender can't modify                */
#define ISMPTR  0x00010000L       /*   message header points to other msg */
#define ISTCPY  0x00020000L       /*   message is complete alias of other */
#define FRCLR   0x00040000L       /*   from field of message "cleared"    */
#define TOCLR   0x00080000L       /*   to field of message "cleared"      */
```

Finally, the maximum allowed message text length is Sysop-configurable. A global variable, TXTLEN, is initialized to the proper value when GME is initialized:

```
#define TXTLEN _txtlen            /* message text buffer size            */
unsigned _txtlen;                 /* use TXTLEN, not this                */
```

Since other modules may be initialized before GME, a special function is provided for use in init__ routines that will safely return the proper message text length whether GME has been initialized or not:

```
unsigned                          /*   returns max message text length    */
txtlen(void);                     /* init__ callable get-text-length     */
```

Message Pointers

There is a special type of message that is used strictly within the engine. It consists of just the header; the body is replaced by a special pointer that points to another message on the system. These are referred to as *message pointers*.

When you read a message pointer, the engine examines the flags field of the header. If the ISMPTR flag is set, the engine then reads the message that is pointed to by the first message. There are two distinct types of message pointer, and what happens next depends on which type the first message is. If the first message is a regular message pointer, the text of the second message is used with the header of the first message to form a complete message. If the first message has the ISTCPY (true copy) flag set, the text *and* header of the second message are used in place of the first message. This process is usually completely transparent, but you may need to be aware of it in certain circumstances.

Forum messages directed to a user can appear in the user's E-mail (depending on the user's preferences). This is done by placing a true copy message pointer in the E-mail data file that points to the forum message. Thus, when a user reads a forum message to them from E-mail, they see the message exactly as it appears in the forum. Any changes made to a true copy are reflected in the original message, thus any changes made to the message from E-mail will also appear in the forum.

Distribution lists use regular message pointers. First a normal message is written in the E-mail data file, addressed to the list being used. Then a message pointer is written to each address on the list, pointing to the original message.

Regular message pointers cannot be modified (no error code is returned, but the changes do not take effect), so the sender cannot change the individual messages sent to a distribution list. However, the original message can be modified, and any changes to the text will be reflected in all the individual messages.

Forums

The configuration information for forums is stored in a forum definition structure. This structure exists in two distinct forms: an on-disk format and an in-memory format. In addition to the obvious distinction, the two formats are used for different purposes. The on-disk format is used when creating a forum or getting “all info” about the forum. The in-memory format is used in all other contexts.

These two definition formats are as follows:

```
#define FORNSZ   16              /* forum name size                    */
#define MAXDIR   66              /* maximum DOS directory spec length  */
#define MAXPATH  80              /* maximum DOS path+filename length   */

struct fordef {                  /* in-memory forum definition struct  */
     unsigned forum;             /*   unique forum identifier          */
     char name[FORNSZ];          /*   forum name                       */
     char topic[TPCSIZ];         /*   short description/topic          */
     char forop[UIDSIZ];         /*   User-ID of Forum-Op              */
     int dfnum;                  /*   index of forum's data file in array*/
     char *attpath;              /*   path where attachments are stored */
     unsigned nthrs;             /*   number of threads in forum       */
     unsigned nmsgs;             /*   number of messages in forum      */
     unsigned nfiles;            /*   number of files in forum         */
     unsigned nw4app;            /*   number of files waiting for apprvl */
     char forlok[KEYSIZ];        /*   key required for privileged access */
     unsigned char dfnpv;        /*   default non-priv access setting  */
     unsigned char dfprv;        /*   default priviledged access setting */
     unsigned char mxnpv;        /*   maximum non-priv access setting  */
     int msglif;                 /*   message lifetime (days)          */
     int chgmsg;                 /*   charge per message posted        */
     int chgrdm;                 /*   charge per message read          */
     int chgatt;                 /*   charge per file attachment uploaded*/
     int chgadl;                 /*   charge per file attachment download*/
     int chgupk;                 /*   charge per-kbyte for upload      */
     int chgdpk;                 /*   charge per-kbyte for download    */
     int ccr;                    /*   credit consumption rate          */
     unsigned char pfnlvl;       /*   profanity suppression level      */
     unsigned crdate;            /*   forum creation date              */
     unsigned crtime;            /*   forum creation time of day       */
     unsigned seqid;             /*   number of forum in list of forums */
     int necho;                  /*   number of echo addresses         */
     char *echoes;               /*   pointer to echo address array    */
};
```

```
struct fordsk {                  /* on-disk forum definition structure   */
     unsigned forum;             /*   unique forum identifier            */
     char name[FORNSZ];          /*   forum name                         */
     char topic[TPCSIZ];         /*   short description/topic            */
     char forop[UIDSIZ];         /*   forum-op                           */
     char datfil[MAXPATH];       /*   path+filename of forum data file   */
     char attpath[MAXDIR];       /*   path where attachments are stored  */
     unsigned nthrs;             /*   number of threads in forum         */
     unsigned nmsgs;             /*   number of messages in forum        */
     unsigned nfiles;            /*   number of files in forum           */
     unsigned nw4app;            /*   number of files waiting for apprvl */
     char forlok[KEYSIZ];        /*   key required for privileged access */
     int dfnpv;                  /*   default non-privileged access      */
     int dfprv;                  /*   default privileged access setting  */
     int mxnpv;                  /*   maximum non-privileged access      */
     int msglif;                 /*   message lifetime (days)            */
     int chgmsg;                 /*   charge per message posted          */
     int chgrdm;                 /*   charge per message read            */
     int chgatt;                 /*   charge per file attachment uploaded*/
     int chgadl;                 /*   charge per file attachment download*/
     int chgupk;                 /*   charge per-kbyte for upload        */
     int chgdpk;                 /*   charge per-kbyte for download      */
     int ccr;                    /*   credit consumption rate            */
     int pfnlvl;                 /*   profanity suppression level        */
     unsigned crdate;            /*   forum creation date                */
     unsigned crtime;            /*   forum creation time of day         */
     int necho;                  /*   number of echo addresses           */
     unsigned seqid;             /*   number of forum in list of forums  */
     char spare[8];              /*   spare space, decrease when adding  */
     char info[1];               /*   variable-length forum info         */
                                 /*   echo addresses (if any)            */
                                 /*   description/help message           */
};

#define DFTPFN 4                 /* use pfceil for profanity check       */
#define MAXFDV (16384-sizeof(struct fordsk)-1)
                                 /* max variable part of forum definition*/
```

The forum field is a unique number assigned to each forum when it is created (see page 174).

The name field is a short, memorable name for the forum. It is used by users when selecting the forum, and is typically displayed when reading messages. It can be up to 15 displayable ASCII characters long, excluding spaces, `@` and `;`

Forum messages can be stored in a single file, or distributed across many files. There is one restriction in this regard: all the messages in a single forum must be stored in the same file. The complete path and filename of the data file in which messages in this forum are stored is in the datfil field. Likewise, attachments to forum messages can be stored wherever you like, but all attachments in a single forum will reside in the same directory. The directory in which attachments to messages in this forum are stored is in the attpath field.

The system-wide profanity suppression level can be overridden on a forum-by-forum basis. The pfnlvl field may be set to one of the standard (0-3) profanity levels, or DFTPFN to use the system-wide profanity level.

Since forums are not stored with a set order, the seqid field is provided as the means to create a custom sequence of forums (the order in which they appear in a list of forums for terminal-mode users — C/S users always see them in alphabetical order). The sequence IDs are always continuous from zero up to the number of forums minus one.

Each forum may have multiple echo addresses. The number of echo addresses is stored in the necho field. The addresses themselves are stored in an array of fixed-length strings — each one MAXADR long. This array is stored in the first necho*MAXADR bytes of the info field when stored on disk. When in memory the echo addresses are stored in a separate buffer pointed to by the echoes field of the fordef structure. The number of echoes per forum is limited by the maximum record size.

The rest of the info field is used for a long description/help message. This is a NUL-terminated string for Sysops to provide additional information on a forum that they think would be useful to their users. This text is displayed when a C/S user requests details on a forum and when a terminal-mode user requests help from the main Forums menu. The length of this text is limited to MAXFDV-necho*MAXADR.

GME Initialization

Most initialization of GME is handled internally. However, there is some per-user information maintained by GME that must be initialized when a user logs on and closed when the user hangs up. These functions are as follows:

```
int                              /*   returns GME status code           */
inigmeu(void);                   /* initialize GME/user stuff at logon   */
                                 /*   (must be called by user interface) */

void                             /* (must be called by user interface)   */
clsgmeu(void);                   /* close GME/user stuff at logoff       */
```

The function inigmeu() returns the status code GMERST if the user's E-mail high message pointer has been corrupted (and resets said pointer to zero). Otherwise it returns GMEOK.

To see if a user's quickscan record has been loaded yet, you can use the following function or macro:

```
#define qsonsys(uid)  (onsqsp(uid) != NULL)

struct qscfg *                   /*   returns NULL if not found          */
onsqsp(                          /* get online user's quickscan          */
char *uid);                      /*   given User-ID                      */
```

Note: If the standard E-mail/Forums user interface module (GALMSG.DLL) is being used, you should not call inigmeu() or clsgmeu() as the standard E-mail/Forums module will do so.

Sending Messages

Sending of messages is divided into two fundamentally different types:

- messages sent on behalf of a user (e.g., Joe writes a message to Sysop)
- messages sent by processes (e.g., a Shopping Mall product sending a "Thank you for buying something" message)

Sending a message is a multi-step process involving access and credit checks (when sending from a user), preparation of the message packet itself, and cycling the sending of the message. Both e-mail and forum messages use the same send-message functions.

Validation

To send a message on behalf of a user, you must first initialize a work area that will be used throughout the send-message request. Next, the address to which the message is to be sent and any features (e.g., attachments, return receipts, etc.) must be validated. The validation process checks whether the address to send to exists (or a given feature is available) and whether the user sending the message has enough access and credits to send the message (or use a particular feature).

The functions used to do this validation are:

```
int                              /*   returns VAL code                   */
valadr(                          /* validate address                     */
void *workb,                     /*   GME work space                     */
char *from,                      /*   User-ID writing the message        */
char *to,                        /*   to address                         */
unsigned forum);                 /*   forum message being written in     */
```

```
int                                /*   returns VAL code              */
valatt(                            /* validate file attachments       */
void *workb,                       /*   GME work space                */
char *from,                        /*   User-ID writing the message   */
char *to,                          /*   to address                    */
unsigned forum);                   /*   forum message being written in    */

int                                /*   returns VAL code              */
valrrr(                            /* validate return receipt request */
void *workb,                       /*   GME work space                */
char *from,                        /*   User-ID writing the message   */
char *to,                          /*   to address                    */
unsigned forum);                   /*   forum message being written in    */

int                                /*   returns VAL code              */
valpri(                            /* validate priority messaging     */
void *workb,                       /*   GME work space                */
char *from,                        /*   User-ID writing the message   */
char *to,                          /*   to address                    */
unsigned forum);                   /*   forum message being written in    */
```

The from parameter to these functions is the User-ID of the person sending the message (and it must be a valid User-ID), the to parameter is the address the message will be sent to, and the forum parameter is the forum ID of the destination forum (or EMLID for E-mail).

These functions return special feature validation codes which are defined as follows:

```
                                   /* feature validation codes        */
#define VALYES   1                 /*   feature is available          */
#define VALNO   -1                 /*   feature is not available      */
#define VALACC  -2                 /*   user doesn't have access      */
#define VALCRD  -3                 /*   user doesn't have enough credits   */
```

You must validate each feature used, but it will not hurt anything to validate a feature and *not* use it. The return value from any given function applies only to the feature it checks. So if valatt() returns VALCRD, valrrr() will not necessarily return VALCRD.

A few things to note when using the valxxx() functions: access is always checked before credits. So if one of the functions returns VALCRD, the user *does* have access. The VALNO return code means that the address does not exist or the feature is not available for the given destination (e.g., valrrr() when writing to a forum always returns VALNO since forum messages can't have return receipts).

The valadr() function is a special case among these functions. If valadr() does not return VALYES, the message cannot be sent at all. You should always call valadr() first. Also, the to parameter passed to valadr() must be a writeable buffer at least MAXADR bytes long. This is because valadr() will fix up the to address by, for example, applying proper case to User-IDs. It will also try to auto-sense the appropriate exporter for an address (and add the appropriate prefix) if the address is clearly not a User-ID, forum, or distribution list.

Note: You must use the same parameters for each of these functions, and you must use the same values when forming the message packet. In other words, the same from, to, and forum must be used throughout. If you need to change one of these after validation, you must re-validate.

The return codes from the valxxx() functions can be converted into standard GME status codes using the following function:

```
int                              /*   returns standard GME status code   */
val2gme(                         /* convert a VAL code to status code    */
int valcode);                    /*   VAL code to convert                */
```

which returns the following correspondences:

```
VALYES == GMEOK
VALNO  == GMEERR
VALACC == GMEACC
VALCRD == GMECRD
```

To quickly check the current user's access and credits when writing e-mail, you can use:

```
int                              /*   returns VAL code                   */
wrtany(void);                    /* can current user E-mail anyone       */
                                 /*   (other than Sysop)                 */
```

This function will return VALYES if the user has access and enough credits to write e-mail, VALACC if the user doesn't have access to write e-mail, or VALCRD if the user doesn't have enough credits. As with the valxxx() functions, if the return code is VALCRD, the user *does* have access.

Forming the Message Packet

The message packet consists of the filled-out message header, message body text, file attachment (if any), and cc: address list (if any). This entire packet must be completed before the message can be sent.

The following fields of the message header must be initialized before sending the message: forum, from, to, topic, attname, and flags.

The forum, from, and to fields should be initialized with those values passed to the validation functions (remember that valadr() fixes up the to address, so you should copy the message header to field directly from the buffer passed to valadr()). The topic will typically be supplied by the user, and the attname field should be filled in with a valid DOS filename if there is an attachment. The flags that must be initialized are: PRIMSG, RECREQ, FILATT, and FILIND. All other flags and fields will be filled in by the engine.

Requesting return receipts and priority messages is fairly straightforward — just set the appropriate flag. Attaching a file is a bit more involved. *In*direct file attachments (where the files are already on a drive available to the server) are fairly simple: set the FILATT and FILIND flags, then pass the path and filename of the file to be attached to the send-message function (discussed below). For a direct attachment (where the files are on a user's PC and must be uploaded to the server), you must first fill in all the message header fields discussed above (except attname), set the FILATT flag, clear the FILIND flag, then call:

```
char *                          /*   path & filename to place file in   */
ulname(                         /* path & filename to upload att to     */
struct message *msg);           /*   header of message to attach to     */
```

passing it your filled-out message header. This will return a pointer to a temporary buffer containing the path and filename into which you must upload (or copy) the direct attachment. You must also copy this path and filename into your own buffer so that you can pass it to the send-message function. When you upload the attachment, you can fill in the attname field with the filename reported by the upload protocol (if the protocol supports this) and/or prompt the user for the filename.

With a filled-out message header, you can use the following function to determine what the total cost to send the message will be:

```
long
sendchg(                        /* total charges to send a message      */
void *workb,                    /*   work area in use                   */
struct message *msg,            /*   new message structure              */
long attsiz);                   /*   attachment size                    */
```

The address and any features specified in the message header must have been validated before calling sendchg().

Carbon copies are sent as part of a send-message request. This is done by providing a carbon copy list to the send-message function. This list is just a series of addresses separated by semi`;`colons. These addresses can be any of the four types of addresses supported by GME (User-IDs, forums, exported addresses, and distribution lists).

Individual addresses can be parsed out of this list using the function:

```
char *                              /*   ptr to next entry (NULL if done)   */
parscc(                             /* parse next cc: from list             */
char *addr,                         /*   buffer to put cc: address into     */
char *list);                        /*   ';'-delimited cc: list             */
```

This function is designed for sequential parsing of the cc: list. It takes a pointer to the start of the list (or the beginning of an address), copies the first address into the supplied buffer, and returns a pointer to the start of the next address in the list.

The carbon copy addresses need not be validated — the engine will check each address before sending a copy. It will strip off any unsupported features (e.g., return receipt to a forum). The engine will *not* send the copy if the address is invalid or the user doesn't have enough access or credits to send the copy.

You may use the above validation functions to check the cc: addresses but, if you do, you must not use the same work area to check a cc: address as you are using for the message itself.

Sending the Message

Once the message packet is complete, the message is sent by calling:

```
int                                 /*   returns standard GME status codes  */
sendmsg(                            /* send a new message from a user       */
void *workb,                        /*   GME work space (provided by caller)*/
struct message *msg,                /*   new message structure              */
char *text,                         /*   message body text                  */
char *filatt,                       /*   path+filename of att (if any)      */
char *cclist);                      /*   list of cc: addresses (if any)     */
```

This function sends the primary message (including forum echoes, if any), then starts sending the carbon copies. To indicate no carbon copies, either pass NULL for the cclist parameter or pass an empty string for cclist (i.e., cclist[0] == '\0'). Since this function may require cycling (definitely if the message is to a distribution list or has carbon copies), all parameters must retain their contents across cycles (i.e., don't use vdatmp). The text, filatt, and cclist parameters are considered read-only by the engine, but the work area and the message header buffer must be writeable by the engine.

Upon completion sendmsg() will automatically close the work area passed to it. The following status codes may be returned:

GMEAGAIN	Cycle, then call sendmsg() again.
GMEOK	The message has been sent successfully.
GMEAFWD	The message has been sent successfully and has been auto-forwarded. The original addressee is in the to field of the message header, and the address to which the message was forwarded can be gotten by calling gmexinf().
GMEERR	The message packet was not properly formed or not all the features used were validated.
GMECRD	The user does not have enough credits to send the message.
GMEIVA	The file attachment specified in the filatt parameter could not be found or was not in the right place (if direct).
GMENOAT	The file attachment could not be attached to the message for some reason.
GMEMEM	There was not enough free memory to process a !QUICK list.

Other error status codes may be added in later versions or returned by exporters, so you should always have a default case when handling error codes.

If an error does occur, you cannot just correct it and try again. You will have to start over from scratch — reinitialize the work area, form and validate the message header again, and try uploading the file again.

This function will generate the following callback events (page 171) as appropriate:

EVTSTRT	Starting to send message to primary addressee. If the primary addressee is a distribution list, the EVTDSTS event will precede this event, and this event will be generated for each address in the list. gmexinf() returns the address to which the message is to be sent.
EVTDONE	Done sending message to primary addressee. This event will not be generated if the primary addressee is not a distribution list and there are no carbon copies to be sent. Otherwise, it will be generated for each message sent (the primary message, and each address in a cc: list or distribution list). gmexinf() returns the address to which the message was sent. This would be the forwardee if the message was auto-forwarded and the forum address (the forum name preceded by a **/**) if the message was sent to a forum. When this callback event occurs, the status code passed to your callback handler may be any of the ones listed on page 186.
EVTCCST	Starting to send carbon copies.
EVTDSTS	Starting to process a distribution list. This may be the primary addressee or one of the cc: addresses. gmexinf() returns the list name.
EVTDSTD	Finished processing a distribution list.
EVTCPYS	Starting to copy file attachment. This may occur due to the message being sent to an exported address or to a cc: address. gmexinf() returns the path and filename being copied to.
EVTCPYD	Finished copying file attachment. gmexinf() returns the path and filename copied to.

Sending a Message From a Process

If you need to send a message that is not from a specific user (for example, a game module that sends a "Congratulations! You won last week's lotto drawing..." message), you can use the following function:

```
int                                 /*   returns standard GME status codes  */
gsndmsg(                            /* send message (non-user specific)     */
void *workb,                        /*   GME work space (provided by caller)*/
struct message *msg,                /*   new message structure              */
char *text,                         /*   message body text                  */
char *filatt);                      /*   path+filename of att (if any)      */
```

Using this function is very similar to using sendmsg() — you must form the message packet, initialize a work area, cycle the function until it finishes, and it closes the work area when done. The primary difference is that you do not need to validate the message before sending it.

You must fill in the same message header fields when forming the message packet. However, since the message is not validated, you must make sure that the message header fields are correctly filled in and all necessary flags are set. In general, it is best to setmem() the message header to all-zero first, then fill in the required fields.

Note that you can set any of the non-system message flags. In particular, if you are sending an e-mail message with an attachment (or a forum message for which you want the attachment pre-approved), you must set the FILAPV flag in addition to the FILATT flag.

You must also be sure that you do not include any unsupported features with your message (e.g., don't set the PRIMSG flag for a forum message, or try to send an attachment to an exporter that doesn't support attachments). Finally, this function can not be used to send a message to a distribution list — only User-IDs, forums, or exported addresses are supported.

The following status codes (other than GMEAGAIN) may be returned by gsndmsg():

GMEAGAIN	Cycle, then call gsndmsg() again.
GMEOK	The message has been sent successfully.
GMEERR	The message packet was not properly formed.
GMENOAT	The file attachment could not be attached to the message for some reason.

Other error codes may be added later or returned by exporters.

To determine whether the address you want to send the message to is valid or not, you can use one of the following functions:

```
BOOL
adrxst(                           /* does this address exist?            */
char *adr);                       /*   address to check                  */

BOOL                              /*   returns TRUE if address exists    */
fixadr(                           /* fix up an address                   */
char *from,                       /*   User-ID of sender (NULL for any)  */
char *adr);                       /*   address to fix up                 */
```

The function adrxst() will return TRUE or FALSE depending on whether the address exists or not. The function fixadr() will fix up the address as well (e.g., apply correct capitalization to a User-ID). The from parameter to fixadr() is used for auto-sensing the correct exporter (if a prefix is not supplied) based on the user's access to the exporters. To include all exporters in the auto-sensing, pass NULL for the from parameter.

If for some reason you absolutely cannot cycle the message-sending process, you can use the following function to send a message:

```
int                              /*   returns standard GME status codes  */
simpsnd(                         /* simple send msg (non-user specific)  */
struct message *msg,             /*   message header structure           */
char *text,                      /*   message body text                  */
char *filatt);                   /*   path+filename of att (if any)      */
```

This function requires the same message preparation as gsndmsg(), but it does not require a work area or cycling. The return code may be any of the codes returned by gsndmsg(), including GMEAGAIN. If simpsnd() returns GMEAGAIN you do not need to call it again, it just means that the message has been added to a queue and will be sent by a background task.

You should avoid using simpsnd() whenever possible because it is much less efficient than gsndmsg() and there is a greater possibility that messages sent using simpsnd() will get "lost in transit" due to a system crash.

Reading Messages

Reading messages is done as an ongoing request. You set up a context in which you will read messages, then you can read multiple messages within that context. The work area is initialized once, and does not have to be closed or re-initialized when changing contexts. Both E-mail and Forums use the same function calls to read messages.

Setting the Context

The read context consists of the User-ID for whom the messages are being read, the current forum ID, message ID, thread ID, and message sequence.

The User-ID part of the context is used for access and credit checks, so that the user need not be online while the messages are being read.

The message sequence specifies what type of messages you want to read (e.g., e-mail messages to you, forum messages in a thread, etc.). The following sequence codes are supported:

```
                                  /* message reading sequence codes       */
#define ESQTOU 1                  /*   E-mail to user sequence            */
#define ESQFRU 2                  /*   E-mail from user sequence          */
#define ESQTHR 3                  /*   E-mail thread sequence             */
#define FSQFOR 4                  /*   Forum normal sequence              */
#define FSQTHR 5                  /*   Forum thread sequence              */
#define FSQSCN 6                  /*   Forum one-time scan sequence       */
```

Note that separate sequence codes are used for E-mail and Forums.

After initializing a work area to be used for reading, and before starting to read messages, the context must be initialized. This can be done with one of the following functions calls:

```
void
inictx(                           /* initialize read context              */
void *workb,                      /*   work area to initialize            */
char *userid,                     /*   user ID doing reading              */
int sequence,                     /*   sequence to use                    */
unsigned forum,                   /*   forum to read from                 */
long msgid,                       /*   message to start at                */
long thrid);                      /*   thread to use (if any)             */

void
inormrd(                          /* initialize "normal" read context     */
void *workb,                      /*   work area to initialize            */
char *userid,                     /*   user ID doing reading              */
unsigned forum,                   /*   forum to read from                 */
long msgid);                      /*   message to start at                */
```

The function inictx() gives complete control over what goes into the read context. The function inormrd() is simpler to use, and initializes the sequence based on the forum ID given. For E-mail the sequence is initialized to ESQTOU (messages to the user); for Forums the sequence used is FSQFOR (all messages in the forum).

To read new messages, the message ID part of the read context can be initialized using the following function:

```
long
firstnew(                         /* get first new message #              */
char *userid,                     /*   for this user ID                   */
unsigned forum);                  /*   in this forum                      */
```

This function actually returns the highest message number read in the given forum (which can be EMLID). You can then read the next message to get new messages.

Two constants are also defined to indicated the first message and the last message when setting the read context:

```
#define FIRSTM   0L                 /* "first message" code, for searches   */
#define LASTM    0x7FFFFFFFL        /* "last message" code, for searches    */
```

The read context can be altered without closing the work area. This can be done using the above two functions, or a single element of the read context can be altered using one of the following functions:

```
void
seqctx(                             /* change sequence of read context      */
void *workb,                        /*   work area in use                   */
int sequence);                      /*   sequence to change to              */

void
forctx(                             /* change forum of read context         */
void *workb,                        /*   work area in use                   */
unsigned forum,                     /*   forum to change to                 */
int sequence);                      /*   new sequence to establish          */

void
msgctx(                             /* change cur message of read context   */
void *workb,                        /*   work area in use                   */
long msgid);                        /*   message ID to change to            */

void
thrctx(                             /* change thread of read context        */
void *workb,                        /*   work area in use                   */
long thrid);                        /*   thread ID to change to             */
```

Reading Messages

Once the read context has been initialized, messages can be read using several different functions. All of these functions return GME status codes and may require cycling. A special property of these functions is that the buffer the message is read into is only required on the last cycle. So a temporary buffer (like vdatmp) can be used.

The following functions are the normal message reading functions:

```
int                               /*   returns standard GME status codes  */
readmsg(                          /* read current message in context      */
void *workb,                      /*   GME work space (provided by caller)*/
struct message *msg,              /*   message header structure buffer    */
char *text);                      /*   message body text buffer           */

int                               /*   returns standard GME status codes  */
nextmsg(                          /* read next message in context         */
void *workb,                      /*   GME work space (provided by caller)*/
struct message *msg,              /*   message header structure buffer    */
char *text);                      /*   message body text buffer           */

int                               /*   returns standard GME status codes  */
prevmsg(                          /* read previous message in context     */
void *workb,                      /*   GME work space (provided by caller)*/
struct message *msg,              /*   message header structure buffer    */
char *text);                      /*   message body text buffer           */
```

To read the message nearest the current message in context, the following function can be used:

```
int                               /*   returns standard GME status codes  */
nearmsg(                          /* read nearest to cur msg in context   */
void *workb,                      /*   GME work space (provided by caller)*/
int nearop,                       /*   get near "style" to use            */
struct message *msg,              /*   message header structure buffer    */
char *text);                      /*   message body text buffer           */
```

where the nearop argument can have one of the following values:

```
                                  /* read nearest message "style" codes   */
#define LEGT 0                    /*   try less/equal then greater        */
#define LTGE 1                    /*   try less then greater/equal        */
#define GELT 2                    /*   try greater/equal then less        */
#define GTLE 3                    /*   try greater then less/equal        */
```

These codes specify how nearmsg() should determine what is the nearest message. For example, if LEGT is used, the engine will first try to find a message less than or equal to the current message in the read context. If it can't find one less than or equal, it will then try to find one that is greater than the current message. Note that less than and greater than are within the current sequence context.

Note: nearmsg() cannot be used if the sequence is FSQSCN.

To read the next or previous message in a given sequence without changing the current sequence context, use the following functions:

```
int                          /*   returns standard GME status codes  */
nextseq(                     /* read next message a sequence         */
void *workb,                 /*   GME work space (provided by caller)*/
int sequence,                /*   sequence to use                    */
struct message *msg,         /*   message header structure buffer    */
char *text);                 /*   message body text buffer           */

int                          /*   returns standard GME status codes  */
prevseq(                     /* read previous message in a sequence  */
void *workb,                 /*   GME work space (provided by caller)*/
int sequence,                /*   sequence to use                    */
struct message *msg,         /*   message header structure buffer    */
char *text);                 /*   message body text buffer           */
```

To read the parent of the current message (the message to which the current message is a reply), use this function:

```
int                          /*   returns standard GME status codes  */
readpar(                     /* read parent of message               */
void *workb,                 /*   GME work space (provided by caller)*/
struct message *msg,         /*   message header structure buffer    */
char *text);                 /*   message body text buffer           */
```

You do not actually have to read the current message before reading its parent. Just setting the context is sufficient.

These read-message functions may return one of the following status codes:

GMEAGAIN	Cycle, then call the read-message function again.
GMEOK	Found the requested message.
GMEERR	The read context was not correctly initialized, or the current message was not found when calling readpar(), or nearmsg() was called with the FSQSCN sequence (searching for messages and the FSQSCN sequence are discussed in the section on One-Time Search, below).
GMENFND	Requested message could not be found.
GMEACC	User does not have access to read requested message.
GMECRD	User does not have enough credits to read requested message.

Note: the msg and text buffers are not altered by the read-message functions unless a message is found. And as always, other error codes may be added later.

After Reading a Message

After the user has read a message, the message should be marked as having been read. When a message is marked read, any per-message read charges are deducted from the user's account, the highest message read in the current forum is updated if necessary, and return receipts are generated if necessary.

This is all done by the following function:

```
int                                  /*   returns standard GME status codes  */
markread(                            /* mark a message as read               */
void *workb,                         /*   GME work space (provided by caller)*/
struct message *msg,                 /*   message header structure buffer    */
char *text);                         /*   message body text buffer           */
```

There are a couple things you must do to use this function correctly. First, the message *must* have been read before calling this function, and the msg parameter passed to markread() without being altered. Some functions which rely on the read context (see While Reading Messages on page 195) will work properly with only the read context, this one will not. Second, this function may require cycling, so the buffers passed to it must retain their contents across cycles.

The following status codes may be returned by this function:

GMEAGAIN	Cycle, then call markread() again.
GMEOK	The message has been marked read.
GMERRG	A return receipt was generated.
GMEAFWD	A return receipt was generated and it was auto-forwarded. The address to which it was forwarded can be gotten from gmexinf().
GMEERR	The read context was not correctly initialized or the from address specified an exporter that is not available (and a return receipt was to be generated).
GMENRGM	The engine was unable to re-read the message after generating a return receipt, so the contents of the msg and text buffers are invalid.

Other error codes may be added later or returned by exporters. In any event, if an error code is returned, the contents of the msg and text buffers should be considered invalid.

The meaning of "the user has read a message" can vary depending on the implementation.

In terminal-mode E-mail/Forums, a message is marked read only after the body text of the message is displayed to the user.

In C/S E-mail, the messages must be delivered to the client before they can be read by the user. In this case, the messages are marked read on the server as they are delivered to the client, before the user has actually seen them displayed on his screen.

In C/S Forums, there are two types of dynapak for reading messages. One type returns only the header information while the other returns the whole message (header and body). In this case, the whole-message dynapak calls markread() while the header-only dynapak does not.

One thing to note is that this function will clear the return-receipt-requested flag (RECREQ) of the message after generating a return receipt. So if you need to know whether a return receipt was requested after calling markread(), you will need to save that information in your own format and not rely on the flags field of the message header.

While Reading Messages

Download Attachment

Since access restrictions and credit charges apply when downloading attachments, GME provides a set of functions to manage access, credit charges, and auditing attachment downloads. These functions can be used to tag an attachment then download it later. The actual transfer is not handled by GME: if terminal mode, download it via the File Transfer Service (page 147); if C/S mode, download it as a file dynapak (**Error! Reference source not found.**).

First initialize the read context and read the message. Then tag the attachment, using the following function:

```
int                              /*   returns standard GME result codes  */
tagatt(                          /* create tag for message attachment    */
void *workb,                     /*   GME work space (provided by caller)*/
struct message *msg,             /*   message header structure           */
char *tag);                      /*   file tag buffer to use             */
```

It will generate a GME-specific tag structure and put it in the tag argument. The buffer used for the tag should be at least TSLENG bytes long. This tag is then used for all subsequent operations.

The tagatt() function does *not* require cycling and may return one of the following status codes:

GMEOK	The attachment has been tagged and the GME-specific tag information stored in the tag argument.
GMEACC	The user does not have access to download the attachment.
GMECRD	The user does not have enough credits to download the attachment.
GMENAPV	The attachment has not been approved for download.
GMENFND	The attachment file could not be found.

To get the path and filename to download from, call:

```
char *                          /*   path & filename of attachment     */
dlname(                         /* path & filename to dnload att from   */
struct message *msg);           /*   header of message with attachment  */
```

This returns a pointer to a temporary buffer.

When the download is to begin, call this function:

```
BOOL                            /*   ok to start download?              */
dlstart(                        /* starting to dnload tagged attachment */
char *tag);                     /*   file tag structure in use          */
```

It will return FALSE if the user has run out of credits since tagging the attachment.

If the download is aborted, call this function:

```
void
dlabt(                          /* attachment download aborted          */
char *tag);                     /*   file tag structure in use          */
```

And when the download successfully completes, call this function:

```
void
dldone(                         /* attachment download finished         */
char *tag);                     /*   file tag structure in use          */
```

These three functions require that the user doing the downloading be online and be the current user.

If your application operates in the background or does not have to worry about the download possibly being aborted, you can use the following function instead:

```
BOOL                              /*   ok to download?                */
gdlatt(                           /* tagged attachment downloaded for user*/
char *userid,                     /*   User-ID attachment downloaded for  */
char *tag);                       /*   file tag structure in use          */
```

Call this function before you start your transfer, to be sure it's still ok for the user to download the file.

To get various information about a tagged attachment, the following functions are provided:

```
struct message *                  /*   pointer to temporary msg struct    */
tagmsg(                           /* get tagged message                   */
char *tag);                       /*   file tag structure in use          */

long
msgintg(                          /* what's the message ID associated     */
char *tag);                       /*   with this tag?                     */

unsigned
forintg(                          /* what's the forum ID associated       */
char *tag);                       /*   with this tag?                     */
```

Reply to Message

To reply to a message, the read context must be initialized and the message must be read. After reading the message, the process is very similar to writing a new message. The message packet must be formed and validated using the same functions, and the same fields in the message header must be filled in.

The difference is that you must start with the header from the message you just read, and only change the following fields: from, to, topic, attname, and flags (PRIMSG, RECREQ, FILATT, and FILIND may be set, all others should be cleared). The from field of the original message should be copied into the to field, and the current user's User-ID copied into the from field. The forum field usually will not need to be changed unless it is an E-mail reply to a forum message, in which case the forum should be changed to EMLID. No other fields should be changed.

The function to call to send the reply is:

```
int                               /*   returns standard GME status codes  */
reply(                            /* reply to current message in context  */
```

```
void *workb,                    /*   GME work space (provided by caller)*/
struct message *msg,            /*   message header structure           */
char *text,                     /*   new message text                   */
char *filatt,                   /*   path+filename of att (if any)      */
char *cclist);                  /*   list of cc: addresses (if any)     */
```

The arguments to this function have the same meaning and requirements as the arguments to sendmsg(). The status codes returned are the same as those returned by sendmsg(). When reply() finishes, it closes the work area, so you must re-initialize the work area and the read context before reading more messages.

Copy Message

To copy a message, you must initialize the read context and read the message. Then the address to which the message is to be copied (and any features associated with the message, e.g. file attachments) must be validated.

To validate the destination, use the standard validation functions (valadr(), valatt(), and valpri()), but pass the address to which the message is to be copied as the to argument to these functions (see page 181). Note that valrrr() need not be called: you should markread() the message before copying it so that the "return receipt requested" flag will be clear before you copy the message.

Once the message has been validated, the destination fields of the header (forum and to) must be filled in with the same values passed to the validation functions, and the following function called:

```
int                             /*   returns standard GME status codes  */
copymsg(                        /* send copy of current msg in context  */
void *workb,                    /*   GME work space (provided by caller)*/
struct message *msg,            /*   message header structure           */
char *text);                    /*   message text                       */
```

This function requires cycling, like reply(). Unlike reply(), this function does not close the work area when it finishes. However, the msg and text buffers will no longer contain the current message.

The following status codes may be returned by copymsg():

GMEAGAIN	Cycle, then call copymsg() again.
GMEOK	The message was copied successfully.
GMEAFWD	The message was copied and auto-forwarded. The address to which it was forwarded can be gotten from gmexinf().
GMEERR	The current message could not be re-read or the destination was not validated properly.
GMECRD	The user doesn't have enough credits to copy the message.
GMENOAT	The attachment to the message could not be copied (so the message was not copied either).
GMENCFL	The address specified was a distribution list. Messages can not be copied to a distribution list.

Forward Message

Forwarding a message is very similar to copying a message. You first initialize the read context and read the message. Then validate the destination and call the forward-message function. The main difference is that special validation routines are provided for forwarding:

```
int                             /*   returns VAL code              */
vfwdadr(                        /* validate address for forwarding */
void *workb,                    /*   GME work space                */
char *from,                     /*   User-ID doing the forwarding  */
char *to,                       /*   to address                    */
unsigned forum);                /*   forum message being written in */

int                             /*   returns VAL code              */
vfwdatt(                        /* validate attachment for forwarding */
void *workb,                    /*   GME work space                */
char *from,                     /*   User-ID doing the forwarding  */
char *to,                       /*   to address                    */
unsigned forum);                /*   forum message being written in */

int                             /*   returns VAL code              */
vfwdpri(                        /* validate priority message for fwding */
void *workb,                    /*   GME work space                */
char *from,                     /*   User-ID doing the forwarding  */
char *to,                       /*   to address                    */
unsigned forum);                /*   forum message being written in */
```

These functions return the same codes as the normal validation functions.

Once the destination has been validated, the destination fields of the header (forum and to) must be filled in with the same values passed to the validation functions, and the following function called:

```
int                               /*   returns standard GME status codes  */
fwdmsg(                           /* forward current message in context   */
void *workb,                      /*   GME work space (provided by caller)*/
struct message *msg,              /*   message header structure           */
char *text);                      /*   message text                       */
```

Like copymsg(), this function requires cycling, does not close the work area when finished, the msg and text buffers no longer contain the original message, and the same status codes are returned.

Delete Message

To delete a message, initialize the read context then call:

```
int                               /*   returns standard GME status codes  */
delmsg(                           /* delete current message in context    */
void *workb);                     /*   GME work space (provided by caller)*/
```

The message to be deleted need not be read beforehand. This function currently does not require cycling, but it very likely will in the future. It does not close the work area.

The following status codes may be returned:

GMEAGAIN	Cycle, then call delmsg() again.
GMEOK	The message was successfully deleted.
GMENFND	The message specified in the read context was not found.
GMEACC	The user does not have access to delete the message.
GMEUSE	The message could not be deleted because someone else is using it.
GMENDEL	The message cannot be deleted (the NODEL flag is set).

Modify Message

To modify a message, initialize the read context, fill in the new topic (in the message header) and text, then call:

```
int                               /*   returns standard GME status codes  */
modmsg(                           /* modify current message in context    */
void *workb,                      /*   GME work space (provided by caller)*/
struct message *msg,              /*   message header structure buffer    */
char *text);                      /*   message body text buffer           */
```

The message need not be read before modifying. Only the topic and text are modified.

The following status codes may be returned:

GMEAGAIN	Cycle, then call modmsg() again.
GMEOK	The message was successfully modified.
GMENFND	The message specified in the read context was not found.
GMEACC	The user does not have access to modify the message.
GMEUSE	The message could not be modified because someone else was using it.
GMENMOD	The message cannot be modified (the NOMOD flag is set).

This function currently does not require cycling, but it very likely will in the future.

Note: message pointers that are not true copies of another message (see page 177) cannot be modified. The modmsg() function will return GMEOK, but the changes will not take effect.

Approve Attachment

To approve an attachment to a forum message, you must first read the message, then call:

```
int                            /*   returns standard GME status codes  */
aprvmsg(                       /* approve/unapprove current message    */
void *workb,                   /*   GME work space (provided by caller)*/
struct message *msg,           /*   message header structure buffer    */
char *text,                    /*   message body text buffer           */
BOOL approve);                 /*   TRUE=approve, FALSE=unapprove      */
```

The following status codes may be returned:

GMEAGAIN	Cycle, then call aprvmsg() again.
GMEOK	The message was successfully approved.
GMEERR	The message is an e-mail message.
GMEACC	The user does not have access to approve attachments.
GMENOAT	The message does not have an attachment.

This function currently does not require cycling, but may in the future.

Exempt Message

To exempt a forum message from auto-delete, first read the message, then call:

```
int                                /*   returns standard GME status codes  */
exmtmsg(                           /* exempt/unexempt current message      */
void *workb,                       /*   GME work space (provided by caller)*/
struct message *msg,               /*   message header structure buffer    */
char *text,                        /*   message body text buffer           */
BOOL exempt);                      /*   TRUE=exempt, FALSE=unexempt        */
```

The following status codes may be returned:

GMEAGAIN	Cycle, then call exmtmsg() again.
GMEOK	The message was successfully exempted.
GMEERR	The message is an e-mail message.
GMEACC	The user does not have access to exempt messages.

This function currently does not require cycling, but may in the future.

Thread Information

To get information about the thread a message belongs to, initialize the read context (be sure to initialize the thread ID context either explicitly by using inictx() or by reading a message) then call:

```
int                                /*   returns standard GME status codes  */
thrinfo(                           /* read info on a thread (orig message) */
void *workb,                       /*   GME work space (provided by caller)*/
int direct,                        /*   direction (0=same, 1=next, -1=prev)*/
unsigned *nmsgs,                   /*   number of messages in thread       */
struct message *msg,               /*   message header structure buffer    */
char *text);                       /*   message body text buffer           */
```

This function will read the first message in the thread and the number of messages in the thread. The direct parameter allows you to generate lists of threads in a forum. Passing 1 in the direct parameter will return the first message in the next thread in the forum (based on thread ID). Likewise, passing -1 in direct will return the first message in the previous thread.

The following status codes may be returned:

GMEAGAIN	Cycle, then call thrinfo() again.
GMEOK	Thread information read successfully.
GMENFND	The current thread (if direct == 0) or the next or previous thread (for direct == 1 or -1) could not be found.
GMEACC	The user does not have access to read messages in this forum.

This function may require cycling.

Forum Information

A wide variety of information about forums is available. You can retrieve this information all at once, or just the bits you are most interested in.

General Information

One of the most basic pieces of information about Forums is:

```
unsigned
numforums(void);                /* get number of forums                 */
```

Most of the information about a forum is stored in the forum definition structure (see page 178). There are three ways to get the definition for a given forum: by forum ID, by name, and by sequence ID.

To determine whether a given forum ID even exists, you can use the following macro:

```
#define fidxst(fid) (fid != EMLID && fididx(fid) != NOIDX)
#define NOIDX    -1             /* valid index not found return value   */

int                             /*   (returns NOIDX if not found)       */
fididx(                         /* get index of forum ID in def array   */
unsigned fid);                  /*   forum ID                           */
```

Note: the index returned by fididx() cannot be used directly.

To get the in-memory definition for a forum given the forum ID, you can use one of the following functions:

```
struct fordef *                 /*   pointer to temporary buffer        */
getdefp(                        /* get pointer to forum definition      */
unsigned forum);                /*   forum ID of def to get             */
```

```
struct fordef *                  /*   pointer to destination buffer     */
getdefb(                         /* copy forum def into a work buffer   */
unsigned forum,                  /*   forum ID of def to get            */
struct fordef *workdef);         /*   pointer to work buffer            */
```

The function getdefp() will return a pointer to a temporary buffer containing a copy of the forum definition you requested, while getdefb() will copy the definition into a buffer you supply. If you pass an invalid forum ID to either of these functions, NULL will be returned.

Note: for GME information functions that return a pointer, assume that the pointer is to a single, temporary buffer. So if you need to compare the same piece of information from different sources, you must copy the first piece of information into your own buffer before getting the second.

To get forum information in forum ID order, skipping over unused forum IDs, you can use the following functions:

```
struct fordef *                  /*   pointer to temporary buffer       */
fiddefp(                         /* ptr to forum def in forum ID order  */
unsigned idx);                   /*   index of forum ID                 */

struct fordef *                  /*   pointer to destination buffer     */
fiddefb(                         /* get forum def in forum ID order     */
unsigned idx,                    /*   index of forum ID                 */
struct fordef *workdef);         /*   pointer to work buffer            */
```

The idx parameter passed to these functions is continuous from 0 to numforums()-1. If you pass an invalid index, these functions will return NULL.

To get forum information in sequence, you can use the following functions:

```
struct fordef *                  /*   pointer to temporary buffer       */
seqdefp(                         /* get pointer to forum def in sequence */
unsigned seqid);                 /*   sequence ID of forum              */

struct fordef *                  /*   pointer to destination buffer     */
seqdefb(                         /* get forum def into buffer in sequence*/
unsigned seqid,                  /*   sequence ID of forum              */
struct fordef *workdef);         /*   pointer to work buffer            */
```

These functions are useful for listing forums in their Sysop-defined sequence. Sequence IDs are also continuous, so you can list forums by getting sequence IDs from 0 to numforums()-1. If you pass an invalid sequence ID, these functions will return NULL.

To get forum information in alphabetical order by name, the following suite of functions is provided:

```
struct fordef *                  /*   pointer to temporary buffer       */
nxtdefp(                         /* get ptr to next forum definition    */
char *name);                     /*   given name to start with          */

struct fordef *                  /*   pointer to destination buffer     */
nxtdefb(                         /* copy next forum def into a work buf */
char *name,                      /*   given name to start with          */
struct fordef *workdef);         /*   pointer to work buffer            */

struct fordef *                  /*   pointer to temporary buffer       */
prvdefp(                         /* get ptr to prev forum definition    */
char *name);                     /*   given name to start with          */

struct fordef *                  /*   pointer to destination buffer     */
prvdefb(                         /* copy prev forum def into a work buf */
char *name,                      /*   given name to start with          */
struct fordef *workdef);         /*   pointer to work buffer            */
```

Say you have forums named Hello, Hola, and Shalom:

prvdefp("Hello")	will return NULL because Hello is the first forum.
nxtdefp("Hello")	will return the definition for Hola.
nxtdefp("Junk")	will return the definition for Shalom because it is the first existing forum name farther down the alphabet than the string "Junk". It is irrelevant that Junk is not the name of an existing forum.
nxtdefp("Shalom")	will return NULL because Shalom is the last forum.

If you know the exact name of the forum and just want the definition for that forum, you can call:

```
unsigned                         /*   returns EMLID if doesn't exist    */
getfid(                          /* get forum ID                        */
char *name);                     /*   given forum name                  */
```

Then use the return value in a call to getdefp() or getdefb() to get the forum definition.

If you just want to find out if a forum with a given name exists, you can use the following macro:

```
#define fnmxst(name) (fnmidx(name) != NOIDX)

int                              /*   (returns NOIDX if not found)      */
fnmidx(                          /* get index of named forum in def array*/
char *name);                     /*   forum name                        */
```

Again, the index returned by fnmidx() cannot be used directly.

To get everything there is to know about a forum, use:

```
void
getallf(                          /* get all info about a forum          */
unsigned forum,                   /*   forum ID to get                   */
struct fordsk *workdef,           /*   on-disk forum definition format   */
char *desc,                       /*   buffer for description            */
char *echoes);                    /*   buffer for echoes                 */
```

But if you just want specific pieces of information about a forum, you may be able to use one of the following functions:

```
char *                            /*   returns NULL if doesn't exist     */
getfnm(                           /* get forum name                      */
unsigned fid);                    /*   given forum ID                    */

char *                            /*   returns NULL if doesn't exist     */
getftpc(                          /* get forum topic                     */
unsigned fid);                    /*   given forum ID                    */

void
getecho(                          /* copy forum's echoes into work buffer */
unsigned forum,                   /*   forum ID to get                   */
char *echoes);                    /*   pointer to work buffer            */

void
getdesc(                          /* copy forum's desc into work buffer  */
unsigned forum,                   /*   forum ID to get                   */
char *desc);                      /*   pointer to work buffer            */
```

Access Information

An important subset of forum information is user access. This information is stored both in the forum definition (page 178) and in each user's quickscan record (page 214).

To see if a user is even allowed to know a forum exists, you can use:

```
BOOL                              /*   TRUE if access > NOAXES           */
faccok(                           /* current user has access to Forum?   */
unsigned forum);                  /*   forum ID to check access for      */
```

Typically, you will want to know a user's forum access level. The forum access level determines what operations a user may perform in a given forum.

The following access level codes are currently defined:

```
                                  /* forum access codes                  */
#define NOAXES 0                  /*   no access                         */
```

```
#define RDAXES 2                   /*   read access                     */
#define DLAXES 4                   /*   download access                 */
#define WRAXES 6                   /*   write access                    */
#define ULAXES 8                   /*   upload access                   */
#define COAXES 10                  /*   Co-Op access                    */
#define OPAXES 12                  /*   (ret from foracc() when Forum-Op) */
#define SYAXES 14                  /*   (ret from foracc() when Sysop)  */
#define NOTSET 15                  /*   not set yet (use default)       */
```

Access levels correspond to an increasing number of operations that can be performed. A user with a given access level can perform all operations that a user with a lower access level can, plus additional operations.

The various access levels indicate access to the following operations:

NOAXES	no access, cannot see forum at all
RDAXES	read messages, but not download attachments
DLAXES	download attachments, but not write messages
WRAXES	write messages, but not upload attachments
ULAXES	upload attachments, but they require Forum-Op approval
COAXES	uploads “pre-approved”, no Forum-Op approval required
OPAXES	approve uploads, maintain messages, set user access
SYAXES	create/delete forums, assign Forum-Ops

The last code (NOTSET) is not actually used as an access level code. Instead, it is used by the engine to indicate that the user has default access in the forum. Its use is discussed in more detail on page 214.

There are several functions to get the access level for a given user:

```
int                                /*   returns standard access level code */
foracc(                            /* get current user's forum access level*/
unsigned fid);                     /*   given forum ID                  */

int                                /*   returns standard access level code */
gforac(                            /* get a user's forum access level   */
char *uid,                         /*   given any User-ID               */
unsigned fid);                     /*   and forum ID                    */

int                                /*   returns standard access level code */
qforac(                            /* get a user's forum access level   */
struct qscfg *qsc,                 /*   given quickscan                 */
unsigned fid);                     /*   and forum ID                    */
```

The function foracc() is for the current (online) user; gforac() is for any user, online or offline; and qforac() is for any user, but you must supply a pointer to their quickscan record (e.g., using uqsptr(), see page 217).

When listing access levels, you may find it convenient to use the following function:

```
char *
accstr(                          /* string describing forum access level */
int acclvl);                     /*   access level to describe           */
```

It returns a string containing the English name of each access level: "Zero", "Read", "Download", "Write", "Upload", "Co-Op", "Forum-Op", "Sysop", or "<unassigned>" for NOTSET.

Forum Management

Forum management operations are performed by the Forum-Op or Sysop. These operations include creating, editing and deleting forums, and setting user access. Forum management also includes copying, forwarding, modifying, deleting and exempting messages, and approving attachments, but these operations are discussed in the “While Reading Messages” section, above.

Forum Definition Management

To create a forum, you must first fill in all of the fields of an on-disk format forum definition structure. The default values for most of the forum definition fields, as configured by the Sysop, are declared in GME.H:

```
int forccr,                 /* default forum credit consumption rate     */
    forlif,                 /* default lifetime of a forum message (days)*/
    fmschg,                 /* default credit charge to write forum msg  */
    fmrchg,                 /* default credit charge to read forum msg   */
    fulchg,                 /* default credit charge per file upload     */
    fkuchg,                 /* default per-kbyte charge per file upload  */
    fdlchg,                 /* default credit charge per file download   */
    fkdchg;                 /* default per-kbyte charge per file download*/
char *forprv;               /* privileged forum usage key                */
```

To initialize the definition structure with the various default values for access, credit charges, etc., you can use:

```
int                                  /*   returns standard GME status codes  */
inifdef(                             /* initialize forum def w/defaults      */
void *workb,                         /*   GME work space (provided by caller)*/
struct fordsk *newdef);              /*   memory area to initialize          */
```

This function initializes the access and credit charges to the default values set in the GALME.MSG file, initializes the Forum-Op field to the current user, and the attachment directory to a fixed default value. It then searches through all the forum data files and counts the number of messages in each. It then initializes the data file field to the file with the fewest messages. This default may be replaced by any valid path and filename. In addition, the data file and attachment path need not exist already — the engine will create them if necessary.

Determining the number of messages in each file does not take very long, but there may be many files, so this function may require cycling. In addition to GMEAGAIN, inifdef() will only return GMEOK, or GMEACC if the current user does not have Sysop access to the forums.

Note that inifdef() requires a work area, so the work area must be initialized before calling it; inifdef() will close the work area when it finishes.

Once the definition structure has been initialized, you can allow the user to edit it. Note that the forum name field is left blank by inifdef(). This field must be filled with a unique name before creating the forum. When the user enters a name for the forum, check it for validity using:

```
BOOL
valfornm(                            /* is this a valid forum name?          */
char *name);                         /*   name to check                      */
```

This function requires that the forum name be at least one but no more than FORNSZ-1 characters long, and contain no space, `@`, or `;` characters.

The topic, description, and echo fields are also left blank by inifdef() and should be filled in before creating the forum. The forum, seqid, nthrs, nmsgs, nfiles, and nw4app fields will be initialized by the engine when the forum is created.

The topic and description are simple text fields that can be edited as one would edit the topic and text of a message.

The echo fields (necho and the echo address array) are a bit more difficult to manipulate. Thus you may want to use the following functions when adding or deleting addresses in the echo array:

```
int                                /*   returns new number of echoes      */
addecho(                           /* add an echo to list                 */
char *echoes,                      /*   echo list buffer                  */
char *newadr,                      /*   echo address to add               */
int necho);                        /*   current number of echoes          */

int                                /*   returns new number of echoes      */
delecho(                           /* delete an echo from list            */
char *echoes,                      /*   echo list buffer                  */
int echonum,                       /*   index of echo address to delete   */
int necho);                        /*   current number of echoes          */
```

These functions do not check for size, so you must make sure the echoes buffer you pass to addecho() is at least (necho+1)*MAXADR bytes long.

Once the definition structure has been filled in, create the forum by calling:

```
int                                /*   returns standard GME status codes  */
creatfor(                          /* create a new forum                   */
void *workb,                       /*   GME work space (provided by caller)*/
struct fordsk *newdef,             /*   new forum definition structure     */
char *desc,                        /*   descriptive text                   */
char *echoes);                     /*   pointer to array of echo addresses */
                                   /*   (may be NULL if no echoes)         */
```

This function requires a work area (automatically closed at completion) and cycling. One of these status codes may be returned on completion:

GMEAGAIN	Cycle, then call creatfor() again.
GMEOK	The forum was successfully created.
GMEERR	The current user does not have access to create forums, or the engine could not create either the data file or the attachment directory.
GMEDUP	A forum with the specified name already exists.
GMENFND	The User-ID specified in the forop field does not exist.
GME2MFR	The maximum number of forums that can be supported by the system has been reached.
GME2MFL	The data file specified is not currently in use and the maximum number of data files allowed by the “Dynamic Btrieve files” line in GALME.MDF is already in use.
GMENFID	There are no available forum IDs — deleted forums must be cleaned up to free up some forum IDs.

GMEMEM	There is not enough memory to add another forum.

A couple things to note about creatfor(): no validity checks (other than duplicate names or non-existent Forum-Ops) are done by the engine, thus you must be sure to initialize the definition with valid values; the echoes parameter may be NULL if necho == 0, otherwise, the buffer pointed to by echoes must be at least necho*MAXADR in size; the desc parameter may be any size, but it must not be NULL and it will be truncated if strlen(desc) > MAXFDV-necho*MAXADR.

Once a forum has been created, it can be modified using:

```
int                              /*   returns standard GME status codes  */
modfor(                          /* modify a forum                       */
void *workb,                     /*   GME work space (provided by caller)*/
struct fordef *newdef,           /*   modified forum description         */
char *desc,                      /*   descriptive text                   */
char *echoes);                   /*   pointer to array of echo addresses */
                                 /*(desc & echoes may be NULL if no chng)*/
```

This function is similar to creatfor(), but it takes an in-memory forum definition structure rather than an on-disk structure. The work area is automatically closed upon completion. Only the following fields can be modified using this function: name, topic, forop, forlok, dfnpv, dfprv, mxnpv, msglif, chgmsg, chgrdm, chgatt, chgadl, chgupk, chgdpk, ccr, pfnlvl, and necho. Note that the desc and echoes parameters may be NULL if they are to remain unaltered.

The following status codes may be returned by modfor():

GMEAGAIN	Cycle, then call modfor() again.
GMEOK	The forum was successfully updated.
GMEACC	The current user does not have access to modify forums.
GMEUSE	The forum is being used by someone and cannot be modified right now.
GMENFND	The User-ID specified in the forop field does not exist or the forum ID does not exist.
GMEDUP	A forum with the specified name already exists.
GMEERR	The number of echoes has been altered, but the echoes parameter is NULL.
GMEFDV	The number of echoes has been increased so that there is no longer enough room for both the echo address list and the forum description.

This function currently does not require cycling, but may in the future.

Normally both Sysops and Forum-Ops can use modfor(). However, the Sysop may change the following configuration option (declared in GME.H) so that Forum-Ops no longer have access to edit the whole forum definition:

```
BOOL fopmfd;                    /* allow forum-ops to modify forum defs?*/
```

If this is the case, Forum-Ops can still configure the default access parameters for the forum using:

```
int                             /*   returns standard GME status codes  */
cfgfacc(                        /* configure default access             */
unsigned forum,                 /*   for this forum                     */
struct foracc *acc);            /*   forum access structure             */
```

This function takes the following structure as its primary parameter:

```
struct foracc {                 /* forum access info structure          */
     int dfnpv;                 /*   default non-privileged access      */
     int dfprv;                 /*   default privileged access setting  */
     int mxnpv;                 /*   maximum non-privileged access      */
     char forlok[KEYSIZ];       /*   key required for privileged access */
};
```

The forum definition fields that correspond to the foracc fields are updated by cfgfacc(). Either GMEOK or GMEACC (if the user doesn't have at least OPAXES in the specified forum) will be returned. No cycling is required.

To delete a forum, the following function is used:

```
int                             /*   returns standard GME status codes  */
delfor(                         /* delete a forum                       */
void *workb,                    /*   GME work space (provided by caller)*/
unsigned forum);                /*   forum ID to delete                 */
```

This functions requires a work area, but no other setup (the work area is automatically closed when delfor() finishes). It currently does not require cycling, but may in the future.

The following status codes may be returned:

GMEAGAIN	Cycle, then call delfor() again.
GMEOK	The forum has been deleted.
GMEACC	The current user does not have access to delete forums.
GMENDEL	The forum cannot be deleted because it is the default forum specified by the Sysop in GALME.MSG CNF option DFTFOR.
GMENFND	The specified forum does not exist.
GMEUSE	The forum is being used by someone and cannot be deleted now.

One thing to note about deleting forums is that the forum definition and messages are not actually deleted when you call this function. The forum definition is removed from memory (so the messages can no longer be accessed either) and marked “deleted” on disk, but the definition record and messages are not actually deleted from disk until the next “deleted forum cleanup” occurs.

User Access Management

Two utilities are provided to manage a user’s forum access levels.

To set a user’s access level in a single forum, you should use:

```
int                                /*   returns standard GME status codes  */
setaxes(                           /* set access for a user                */
unsigned fid,                      /*   forum to set access for            */
char *uid,                         /*   user to set access for             */
int acc);                          /*   access to set to                   */
```

This function will handle all the complexity involved in setting a Forum-Op and dealing with online vs. offline users. The following status codes may be returned:

GMEOK	User’s access updated successfully.
GMEACC	The current user does not have access to set access in the specified forum. This code will also be returned if a Forum-Op tries to give another user Forum-Op access.
GMEMEM	The specified forum is not already in the specified user’s quickscan record, and there is not enough room to add another forum.

The acc parameter should be a valid access level, including NOTSET, which will give the user default access in the specified forum.

To set a user's access in many forums at once, the following function can be used:

```
int                                 /*   returns standard GME status codes  */
cpyaxes(                            /* copy forum access                    */
char *dstusr,                       /*   to this User-ID                    */
char *srcusr);                      /*   from this User-ID                  */
```

This function copies all of the non-default access levels from the srcusr to the dstusr. It will work regardless of whether or not the user(s) are online or not. The following status codes may be returned:

GMEOK	Access levels were copied successfully.
GMEACC	The current user does not have access to copy access levels.
GMENSRC	The User-ID specified in the srcusr parameter does not have a quickscan record.
GMENDST	The User-ID specified in the dstusr parameter does not exist.

If there is not enough room in the destination user's quickscan record to add all the new forums, no error code is returned, but not all the access levels will be copied.

Quickscan

The quickscan configuration record (henceforth referred to as the qscfg) is the repository for most per-user information in E-mail and Forums. Its structure is as follows:

```
#define MAXSKWD  80                 /* size of scan keyword string          */

struct qscfg {                      /* quickscan/config per-user data       */
     char userid[UIDSIZ];           /*   master key for lookups             */
     char fwdee[MAXADR];            /*   auto-forwardee for E-mail (if any) */
     unsigned fwdate;               /*   date that forwardee was chosen     */
     char kwds[MAXSKWD];            /*   quickscan keywords                 */
     unsigned curfor;               /*   current forum ID                   */
     int nforums;                   /*   number of forums in quickscan      */
     int flags;                     /*   preference and quickscan flags     */
     long stmsg;                    /*   quickscan starting message ID      */
     char spare[26];                /*   spare space                        */
     char accmsg[1];                /*   start of variable-len accmsg stuff */
};                                  /*   2 arrays: forumID+msgID, acclvl    */
                                    /*   msgID > 0 == "real" quickscan forum*/
```

User Preferences

One of the many bits of per-user information stored in the qscfg is the user's personal preferences. These preferences primarily affect the user interface (and the terminal-mode user interface at that), but are stored and maintained by GME. The individual preferences are stored in the flags field of the qscfg as follows:

```
                                   /* preference-related flag definitions  */
#define CLARPL  0x0001             /*   clear messages after replied to?   */
#define P4NEWM  0x0002             /*   pause at logon for new mail?       */
#define GORTIN  0x0004             /*   if pause, go right into E-mail?    */
#define FORUM2  0x0008             /*   consider forum mail E-mail "to"?   */
#define MSGQUO  0x0010             /*   use message quoting at all?        */
#define ALWQUO  0x0020             /*   always use message message quoting?*/
#define USRSET  0x0040             /*   user has set preferences           */
#define QWKATT  0x0080             /*   include attachments in QWK-Mail?   */
#define CMBHDR  0x0100             /*   combine header and body of messages*/
#define NEWMSG  0x0200             /*   new messages only in quickscan?    */
#define WATONL  0x0400             /*   msgs w/atts only in quickscan?     */
#define TOMEONL 0x0800             /*   msgs to me only in quickscan?      */
#define FRMEONL 0x1000             /*   msgs from me only in quickscan?    */
#define CFWCMT  0x2000             /*   comment when copy/fwding at all?   */
#define ALWCMT  0x4000             /*   always comment when copy/fwding    */
```

The preferences may be set by directly manipulating a user's qscfg, but one general rule should be followed: when setting the "always" flag of an "ever/always" pair (e.g., P4NEWM/GORTIN, MSGQUO/ALWQUO, and CFWCMT/ALWCMT), be sure to set the "ever" flag as well.

Auto-Forwardee

The fwdee field of the qscfg specifies an address to which any e-mail addressed to the user will be forwarded. You should use the following function to set the auto-forwardee:

```
int                                /*   returns VAL code                   */
setafwd(                           /* set auto-forwardee for current user  */
char *newfwde);                    /*   new auto-forwardee                 */
```

This function sets both the fwdee and fwdate fields of the qscfg. It will return VALYES if the specified forwardee is ok, VALNO if the auto-forward address is too long or is invalid (forum addresses and distribution lists are not considered valid auto-forward addresses), or VALACC if the user doesn't have access to auto-forward to the specified address. The auto-forwardee can be cleared by passing an empty string as newfwde.

Per-Forum Information

The primary purpose of the qscfg is to record per-forum information for each user. The information stored per forum is the access level, the highest message read in the forum, and whether or not the forum is included in the user's "quickscan" of forums. This per-forum information is stored in the variable-length accmsg field of the qscfg structure.

The access level is an unsigned integer that governs what actions a user may perform in a forum. The access level codes are discussed in the Forum Information section, page 203. The last three access level codes are of particular interest when discussing the qscfg per-forum information.

First, the NOTSET code is *only* found in the qscfg record. It indicates that the user's access has not been explicitly set by the Forum-Op of the forum. Thus the user has the default access for that forum. The OPAXES and SYAXES codes are unique in that they should *never* be found in the qscfg. Their significance is discussed in greater detail above; for now it is important to note that these two values should never be explicitly put into a user's qscfg.

The highest message ID serves a dual role. In addition to being a starting point when reading new messages, it is used to indicate whether or not the forum should be included in the user's quickscan. This is indicated by the sign of the high message ID: if the high message is positive, the forum should be included in the quickscan, if the high message is negative (or zero), the entry is just being used to track the highest message read, and the forum should not be included in the quickscan (this is referred to as a marker entry).

The number of forums is not limited by the size of the qscfg. This is done by not having a slot in the qscfg for every forum on the system. Instead, a forum is only added to the quickscan when the user explicitly adds it to their qscfg, reads messages in the forum, or the Forum-Op sets their access level to a non-default value.

While there is a relatively high limit on the number of forums on a system, there is a lower limit on the number of forums that can be stored in the qscfg record. This limit is defined in GME.H as:

```
#define MAXNQSF  2400             /* max # forums in a user's quickscan   */
```

The number of forums is limited by the way the per-forum information is stored in the qscfg. Because not every forum is present in the qscfg, the forum ID must be stored along with the high message ID and the access level, so 2400 is the most forum entries that will fit in a 16K qscfg record.

The 2400 forum limit is the maximum limit set by the software. However, Sysops can configure a lower limit in order to preserve memory. The actual limit is stored in the variable:

```
int _maxqsf;                         /* use MAXQSF, not this              */
#define MAXQSF _maxqsf               /* max forums in a user's quickscan  */
```

Since this number is read from an .MCV file at startup, it may or may not have been initialized at the time your module's init__ function is called. If you need to perform operations based on this number as part of your initialization, you can use the function:

```
int                                  /*   returns max # forums allowed in qs */
maxqsf(void);                        /* init__ callable get-max-qscan-forums */
```

This will initialize MAXQSF if it has not already been initialized.

The per-forum information is stored in two parallel arrays, and the number of elements in the arrays is stored in the nforums field. The first array (which starts at accmsg[0]) is an array of structures containing the forum ID and the high message ID (the forum ID is the first field). The second array immediately follows the first. It stores the access level, and is an array of 4-bit integers stored as an array of characters. Thus element 0 is the low order 4 bits of character 0, element 1 is the high order 4 bits of character 0, element 2 is the low order 4 bits of character 1, etc.

The elements in the two arrays must be kept in sync (so that element 0 in the forum ID/high message array corresponds to element 0 in the access level array), and they must be stored in forum ID order.

Due to the complexity of this structure, a large suite of functions is provided to manipulate the per-forum information in a qscfg.

To get a pointer to an online user's qscfg, use:

```
struct qscfg *
uqsptr(                              /* get online user's quickscan       */
int unum);                           /*   user number to get              */

struct qscfg *
myqsptr(void);                       /* get current user's quickscan      */

struct qscfg *                       /*   returns NULL if not found       */
onsqsp(                              /* get online user's quickscan       */
char *uid);                          /*   given User-ID                   */
```

The last function can also be used to see if a User-IDs qscfg is "online" (has been initialized using inigmeu()). A macro is defined in GME.H for this purpose:

```
#define qsonsys(uid) (onsqsp(uid) != NULL)
```

To get the index of a particular forum in the qscfg, use:

```
int                            /*   index in qs arrays (NOIDX if err)  */
qsidx(                         /* get index of forum in quickscan      */
struct qscfg *qsc,             /*   quickscan record                   */
unsigned fid);                 /*   forum ID to get                    */
```

Since this function returns NOIDX if there is no entry for the forum in the qscfg, a macro is available to determine whether a forum is in the qscfg:

```
#define inqs(qsc,fid) (qsidx((qsc),(fid)) != NOIDX)
```

To add a forum entry to the qscfg, use:

```
int                            /*   index in qs arrays (NOIDX if err)  */
add2qs(                        /* add slot for forum to quickscan      */
struct qscfg *qsc,             /*   quickscan record                   */
unsigned fid);                 /*   forum ID to add                    */
```

This function adds a slot for the specified forum (with the high message set to 0 and the access level set to NOTSET) and returns the index at which it added it. If the forum was already in the qscfg, it just returns the index. If there is not enough room to add another forum, it returns NOIDX.

A somewhat more useful function is:

```
int                            /*   index of new entry (NOIDX if error)*/
absadqs(                       /* add forum to qs, del others for room */
struct qscfg *qsc,             /*   quickscan record                   */
unsigned fid);                 /*   forum ID to add                    */
```

This function acts just like add2qs(), but if there is no room to add another entry, it will try to make room. First, it will try to find a deleted forum to remove from the qscfg. If there are no deleted forums to remove, it will search for marker entries for which the access level has not been explicitly set, and will delete the oldest one (the one for which the absolute value of the high message is lowest). Only if neither of these can be found will it return NOIDX.

An even more useful function, for use when configuring a user's quickscan, is:

```
BOOL                           /*   returns FALSE if unable to add     */
```

```
sfinqs(                            /* set forum in quickscan               */
struct qscfg *qsc,                 /*   quickscan record                   */
unsigned fid,                      /*   forum to set                       */
BOOL turnon);                      /*   TRUE = make forum "in" quickscan   */
```

This function will switch forums in and out of the quickscan by adding an entry with a high message ID of 1 if necessary, or changing the sign of the high message if the forum is already present. It will only return FALSE if it cannot absadqs() the specified forum.

The indices returned by qsidx(), add2qs(), and absadqs() can be used to manipulate the qscfg using the following functions:

```
void
idelqs(                            /* delete an entry from the quickscan   */
struct qscfg *qsc,                 /*   pointer to quickscan               */
int idx);                          /*   index of entry to delete           */

void
isethi(                            /* set hi message in quickscan (w/index)*/
struct qscfg *qsc,                 /*   pointer to quickscan               */
int idx,                           /*   index of forum to set              */
long msgid);                       /*   message ID to set as high message  */

void
isetac(                            /* set forum acc in quickscan (w/index) */
struct qscfg *qsc,                 /*   pointer to quickscan               */
int idx,                           /*   index of forum to set              */
int acc);                          /*   access level to set                */

unsigned
igetfid(                           /* get forum ID in quickscan (w/index)  */
struct qscfg *qsc,                 /*   pointer to quickscan               */
int idx);                          /*   index of forum to get              */

long
igethi(                            /* get hi message in quickscan (w/index)*/
struct qscfg *qsc,                 /*   pointer to quickscan               */
int idx);                          /*   index of forum to get              */

int
igetac(                            /* get forum acc in quickscan (w/index) */
struct qscfg *qsc,                 /*   pointer to quickscan               */
int idx);                          /*   index of forum to get              */
```

In general, igetac() and isetac() should *not* be used to manipulate a user's access level. Instead, use foracc() or gforac() or qforac(), and then setaxes() (see pages 207 and 213).

To read or manipulate a forum's qscfg entry without first getting an index, the following functions are available:

```
void
delqs(                          /* delete an entry from the quickscan   */
struct qscfg *qsc,              /*   pointer to quickscan               */
unsigned forum);                /*   forum ID to remove from quickscan  */

BOOL                            /*   returns TRUE if able to set        */
sethi(                          /* set hi message in quickscan          */
struct qscfg *qsc,              /*   pointer to quickscan               */
unsigned forum,                 /*   forum ID to set for                */
long msgid);                    /*   message ID to set as high message  */

BOOL                            /*   returns TRUE if able to set        */
setac(                          /* set forum acc in quickscan           */
struct qscfg *qsc,              /*   pointer to quickscan               */
unsigned forum,                 /*   forum ID to set for                */
int acc);                       /*   access level to set                */

long
gethi(                          /* get hi message in quickscan          */
struct qscfg *qsc,              /*   pointer to quickscan               */
unsigned forum);                /*   forum ID to get for                */
```

Again, setac() generally should not be used.

Quickscan Configuration

Finally, the use for the qscfg from which its name is derived is to quickly scan forum messages. This lets a user read messages in multiple forums without manually selecting each forum. The quickscan is traditionally used to read new messages in forums of interest. Our implementation also allows the user to take advantage of the full power of the One-Time Search service (page 221). In addition to the per-forum information, which specifies what forums are included in the quickscan, the following qscfg fields are used to save the user's quickscan: kwds, stmsg, and the NEWMSG, WATONL, TOMEONL and FRMEONL flags.

The quickscan as stored in the qscfg cannot be used directly to read messages. To read messages using the quickscan, you must first use the function:

```
struct otscan *                 /*   returns copy of pointer to dest    */
qsc2ots(                        /* copy quickscan to a one-time scan buf*/
struct qscfg *qsc,              /*   quickscan to copy                  */
struct otscan *ots);            /*   one-time scan buffer               */
```

to copy the information in the qscfg into a buffer with the proper format to be used by the one-time search facility discussed in the One-Time Search section (described immediately below).

Note: This function puts forums into the scan buffer in sequence-ID order, not alphabetical order (the two do not necessarily correspond). However, C/S-mode users require that the forums be in alphabetical order, so if this function is used for C/S-mode users, the forums must be re-sorted into alphabetical order.

One-Time Search

To search for messages in multiple forums, using keywords and other criteria, a one-time search (a.k.a. one-time scan) system is provided. To perform a search, first set up a buffer containing the search criteria to be used. Then initialize a work area, associate the search criteria buffer with the work area, and initialize the read context of the work area with the FSQSCN sequence. That work area can then be used to read messages selected by the search criteria.

The search criteria are stored in the following structure:

```
struct otscan {                    /* one-time scan setup structure       */
     char keywds[MAXSKWD];         /*   keywords                          */
     long stmsgid;                 /*   starting message ID (0 for none)  */
     char flags;                   /*   search flags                      */
     int nforums;                  /*   number of forums in list          */
     unsigned forlst[1];           /*   list of forums to scan            */
};
                                   /* one-time scan flags                 */
#define SCNEW  0x01                /*   include "new" messages only       */
#define SCATT  0x02                /*   include msgs w/attachments only   */
#define SCTOU  0x04                /*   include msgs to user only         */
#define SCFRU  0x08                /*   include msgs from user only       */
#define SCALL  0x10                /*   include all Forums in Forum list */
```

The forlst field is an array of forum IDs, and nforums is the number of forums in this array. The order in which the forums are listed does not matter to the engine, but it is the order in which the forums will be searched. So the forums should be put in forum sequence ID order for terminal-mode users and in alphabetical order for C/S-mode users.

If the SCALL flag is set, all forums *except* those in forlst will be searched. In this case, the forums are always searched in alphabetical order.

To associate a search buffer with a work area, initialize the work area, then call:

```
void
setscan(                        /* set the scan context              */
void *workb,                    /*   for this work space             */
struct otscan *newscn);         /*   to this scan buffer             */
```

To get a pointer to the search buffer associated with a work area, use:

```
struct otscan *
getscan(                        /* get the current scan context buffer  */
void *workb);                   /*   for this work space             */
```

This will return NULL if no search buffer is associated with the work area.

Configuring the one-time search setup is fairly straightforward. However, a few functions are provided to perform the more complex tasks.

To get the index of a forum in a search setup, you can use:

```
int                             /*   returns NOIDX if not found      */
scnfidx(                        /* get index of forum in search list */
struct otscan *ots,             /*   in this scan                    */
unsigned forum);                /*   forum ID to find                */
```

To keep the search forums in forum sequence ID order when configuring searches for terminal-mode users, you can use:

```
void
addf2ots(                       /* add forum to scan list in seq order  */
struct otscan *ots,             /*   one-time scan structure to update  */
unsigned forum);                /*   forum to add                    */
```

Since the number of forums that can be put in a one-time search setup is not fixed by the engine, this function does no limit checking — you must be sure not to add more forums than will fit in your buffer.

When initializing the read context for a search, the following two functions can be used to start with the first message in the first forum in the search (as determined by the stmsgid field and the SCNEW flag):

```
unsigned                        /*   returns EMLID if no forums      */
fstscnf(                        /* get first forum ID                */
char *userid,                   /*   User-ID doing scan              */
struct otscan *ots);            /*   in this search                  */

long
fstscnm(                        /* get first message ID              */
```

```
char *userid,                   /*   User-ID doing scan            */
struct otscan *ots,             /*   in this scan                  */
unsigned forum);                /*   in this forum (in scan)       */
```

While reading messages using a one-time search, the engine may search through many messages and forums before finding a match (i.e., while cycling nextmsg() or prevmsg()). To keep the caller appraised of its activities, the engine will generate two callback events: EVTNEWF and EVTNEWM. They are generated when the engine starts to process a new forum and a new message, respectively. During an EVTNEWF event, gmexinf() returns the name of the new forum. During an EVTNEWM event, no extended information is provided. See page 169 for details on using callbacks.

It is important to note that the function nearmsg() does not support one-time searches. If you call nearmsg() with FSQSCN as the read context sequence, it will return GMEERR.

Distribution Lists

Distribution lists are an easy way to send the same message to a large group of people without having to type their addresses in every time you want to send a message to that group. Any type of address can be put in a list except for another list. A distribution list name has a unique format that distinguishes it from other types of addresses. It is either a ! or @ character followed by up to eight letters or numbers. Three types of distribution lists are supported by GME: mass mailing lists, personal lists, and Sysop-defined lists.

The mass mailing list is called !MASS. If a user writes a message to !MASS, a copy of the message will be sent to every User-ID on the system. This list is not configurable and there are no special functions or variables for dealing with it.

Personal Lists

Every User-ID on a system has a personal distribution list that can contain up to 40 addresses. When a user addresses a message to !QUICK, a copy of the message is sent to every address in their personal list.

These personal lists are stored in a Btrieve data file in the following structure:

```
#define MAXQIK 40               /* max entries in a !QUICK list    */

struct qikdat {                 /* !QUICK list records structure   */
     char userid[UIDSIZ];       /*   key value for look-up         */
```

```
     int  idx[MAXQIK];             /*   array of entry indexes          */
     char list[1];                 /*   variable-length entry data      */
};
```

Since exported addresses, which can be up to 256 characters (MAXADR) in length, are allowed in distribution lists, and MAXQIK*MAXADR would make for a rather large structure, the addresses in the !QUICK list are stored in a compressed format rather than an array of fixed-length strings.

The list field of the qikdat structure is where the addresses are actually stored. They are appended to one another so that the terminating NUL character of one address is immediately followed by the first character of the next one. The idx field of the qikdat structure is used to locate the individual addresses within the list field.

Important: the order of entries in the idx array does not necessarily correspond to their order in the list field, and entries in the idx array are not necessarily continuous. If an idx array entry contains NOIDX, there is no address associated with that slot.

Due to the complexity of this structure, a suite of functions is provided to manipulate the !QUICK list structure.

To read a user's !QUICK list into a buffer you supply, use:

```
struct qikdat *                    /*   copy of pointer to buffer       */
getqik(                            /* get a !QUICK list from disk       */
struct qikdat *qikbuf,             /*   pointer to !QUICK buffer        */
char *userid);                     /*   user ID to use                  */
```

To create a new, empty list for a specified user, you can use:

```
struct qikdat *                    /*   copy of pointer to buffer       */
iniqik(                            /* initialize !QUICK record for cur user*/
struct qikdat *qikbuf,             /*   pointer to !QUICK buffer        */
char *userid);                     /*   user ID to use                  */
```

To insert an address into a !QUICK list at a given idx slot, you can use:

```
void
insqik(                            /* insert entry into !QUICK buffer   */
int pos,                           /*   position to insert at           */
char *entry,                       /*   new entry for position          */
struct qikdat *qikbuf);            /*   pointer to !QUICK buffer        */
```

If there was an address in that slot already, it will be replaced by the new address.

To delete an address from a !QUICK list, you can use:

```
void
delqik(                         /* delete an entry from !QUICK buffer   */
int pos,                        /*   position to delete                 */
struct qikdat *qikbuf);         /*   pointer to !QUICK buffer           */
```

When you are finished editing a !QUICK list, you can save it using:

```
void
savqik(                         /* save a !QUICK list to disk           */
struct qikdat *qikbuf);         /*   pointer to !QUICK buffer           */
```

Sysop Lists

Sysop-defined distribution lists have names beginning with an @ character. They are configurable (unlike !MASS) and available to the public (unlike individual !QUICK lists). Each list has its own key required for use and its own surcharge for use. Sysop lists are stored in text files with a filename of the list name (less the preceding @) and the extension .DIS, in the directory specified in the GALME.MSG CNF option DLSTPTH.

A suite of functions is provided to list, create, and edit Sysop lists.

To get information on a specific list, use the function:

```
int                             /*   returns standard GME status codes  */
getslst(                        /* get info on Sysop dist list          */
char *lstnam,                   /*   list name                          */
char *lstkey,                   /*   buffer for list key                */
int *surchg);                   /*   buffer for list surcharge          */
```

This function looks for a list with the name specified in the lstnam parameter and puts the basic information about that list (key required for use and surcharge) in the lstkey and surchg buffers. It returns GMEOK if it found the requested list, and GMENFND if not.

To get a list of available Sysop lists, you can use:

```
int                             /*   returns standard GME status codes  */
nxtslst(                        /* get name of next Sysop dist list     */
char *lstnam,                   /*   buffer for list name               */
char *lstkey,                   /*   buffer for list key                */
int *surchg);                   /*   buffer for list surcharge          */
```

This function finds the next list (in alphabetical order by name) after the name specified in the lstnam parameter, and returns the information about it (including the name of the list it found) in the lstnam, lstkey, and surchg buffers. It returns GMEOK if it found another list, and GMENFND if not. Note that this function does not check the current user's access, so you should check lstkey when displaying lists of Sysop lists and not display a list if the user doesn't have the key required to use the list.

Sysop list names may be up to 9 characters long and are composed of an @ followed by up to eight alpha-numeric characters. The maximum list name length is declared as:

```
#define DLNMSZ   10                /* Sysop dist list name size           */
```

Before creating a new Sysop list, you may want to call:

```
BOOL
valslnm(                           /* is this a valid Sysop dist list name?*/
char *name);                       /*   complete name (including @)       */
```

to see if the proposed name is a valid list name, and call:

```
BOOL
dlstxst(                           /* does dist list exist?               */
char *name);                       /*   complete list name (incl @ or !)  */
```

to see if the list already exists.

To create a Sysop list, you should use the function:

```
int                                /*   returns standard GME status codes  */
newslst(                           /* create a new Sysop distribution list */
void *workb,                       /*   GME work space                     */
char *lstnam,                      /*   name for new list                  */
char *lstkey,                      /*   key required to use list           */
int surchg);                       /*   surcharge to use list              */
```

This function will create an empty Sysop list with the specified name, key, and surcharge, and add the list to the list of known Sysop lists maintained by GME. The following status codes may be returned:

GMEOK	The list has been created successfully.
GMEACC	The current user does not have access to create Sysop lists. A user must have the key specified in the GALME.MSG CNF option EDSTKY and declared in GME.H as edstky in order to create or edit Sysop lists.
GMEERR	The list name specified is invalid.
GMEDUP	A list with the same name already exists.
GMEUSE	The engine encountered a file sharing conflict when trying to open the list file.
GMEMEM	There is not enough memory to add the new list to the in-memory list of Sysop lists kept by GME.

Note that newslst() requires a work area. This work area is automatically closed if an error occurs, but if newslst() returns GMEOK, the work area is left open so that it can then be used to add addresses to the new list.

To edit an existing list, you must first open it using:

```
int                              /*   returns standard GME status codes  */
edtslst(                         /* open a Sysop dist list for editing   */
void *workb,                     /*   work area to initialize            */
char *lstnam);                   /*   name of list                       */
```

This function opens the specified list file so that new addresses can be added. The following status codes may be returned:

GMEOK	The list file was successfully opened.
GMEACC	The current user does not have access to edit Sysop lists.
GMEERR	Either the list name or the list file is invalid.
GMENFND	No such Sysop list exists.
GMEUSE	Someone else is currently editing the specified list.

Like newslst(), edtslst() automatically closes the work area if there is an error, but leaves it open if it successfully opens the list file.

To add addresses to a Sysop list, use:

```
int                                /*   returns standard GME status codes  */
addslst(                           /* add an address to a Sysop list       */
void *workb,                       /*   GME work space                     */
char *addr);                       /*   address to add to list             */
```

This function appends the specified address to the list. Note that the work area passed to this function must have been used to open the list file using newslst() or edtslst(). If the work area and list file were not opened correctly, addslst() will return GMEERR, otherwise it will return GMEOK.

To list the addresses in a Sysop list while editing, first call:

```
int                                /*   returns standard GME status codes  */
rstlstp(                           /* set Sysop list pointer to top of list*/
void *workb);                      /*   GME work space                     */
```

This will set the file pointer to the first address in the list. Then call:

```
BOOL
nxtsys(                            /* get next entry in Sysop list         */
void *workb,                       /*   work area to use                   */
char *addr);                       /*   buffer for address                 */
```

to get each successive address in the list into the addr buffer. This buffer must be at least MAXADR in size to accommodate the largest possible address. This function returns FALSE when there are no more addresses (and it doesn't put anything into addr when it returns FALSE).

When you are through editing a list, just close the work area. This will close the list file and free up any other associated resources.

To delete a Sysop list, you can use:

```
int                                /*   returns standard GME status codes  */
delslst(                           /* delete a Sysop dist list             */
char *lstnam);                     /*   name of list                       */
```

This will delete the list file and remove it from the GME-internal list of Sysop lists. The following status codes may be returned:

GMEOK	The list was successfully deleted
GMEACC	The current user does not have access to edit Sysop lists.
GMEERR	The specified list name is invalid.
GMENFND	The specified list does not exist.

Finally, if you need to get the actual list file for some reason (e.g., so a user can download the file), you can use:

```
int                              /*   returns standard GME status codes  */
slstfil(                         /* get info about Sysop list            */
char *lstnam,                    /*   list name                          */
char *lstpath);                  /*   buffer for list path & filename    */
```

This function puts the path and filename that correspond to the specified list in the lstpath buffer (this buffer must be at least MAXPATH in length). The following status codes may be returned:

GMEOK	Found the requested list file, the path and filename are in lstpath.
GMEACC	The current user does not have access to edit Sysop lists.
GMENFND	The specified list does not exist.

Importers and Exporters

Messages addressed to (and sent by) users who are signed up on the local system are addressed by User-ID. The messaging engine exchanges messages with *other* mail systems using importers and exporters. These are special modules responsible for converting messages between The Major BBS's format and other mail systems' formats. Messages are transferred between importers/exporters and the engine through a standard interface.

Exporters

Messages sent from the local system to a remote system are handled by exporters. Each exporter is identified by a unique address prefix — two or three alphanumeric characters (e.g., IN, MHS) that indicate which exporter should be used to send a given message.

When a message is to be sent using an exporter, the address consists of the exporter prefix, followed immediately by a colon, followed by the rest of the address.

Exporters identify themselves to the engine using an exporter control block. In order to be recognized by the engine, an exporter must fill in an exporter control structure, and pass a pointer to it to the GME function:

```
void
register_exp(                    /* register an exporter                 */
struct exporter *exp);           /*   pointer to exporter control block  */
```

The engine checks for any conflicts with the prefixes of the already registered exporters (and catastro()s if there is a conflict), then adds the pointer to the list of registered exporters.

Outside processes can determine how many exporters have been registered by calling:

```
int
numexp(void);                     /* get number of registered exporters   */
```

To see if there are any exporters available to the current user, you can use:

```
BOOL
expavl(void);                     /* are any exporters available?         */
```

This function not only checks to see if any exporters have been registered, but also checks to see if the current user has access to use any of them.

The format of the exporter control block is as follows:

```
#define PFXSIZ 4                  /* max exporter prefix size             */
#define EXPNSZ 16                 /* max exporter name size               */
#define EXPDSZ 51                 /* max exporter description size        */

struct exporter {                 /* message exporter control block       */
     char prefix[PFXSIZ];         /*   address prefix for this exporter   */
     char name[EXPNSZ];           /*   name of exporter (e.g., "Internet")*/
     char desc[EXPDSZ];           /*   description of exporter            */
     char exmp[MAXADR];           /*   example address to this exporter   */
     char wrtkey[KEYSIZ];         /*   key required to use this exporter  */
     int wrtchg;                  /*   per-message surcharge              */
     char attkey[KEYSIZ];         /*   key required for file attachments  */
     int attchg;                  /*   attachment surcharge               */
     int apkchg;                  /*   attachment per-kbyte surcharge     */
     char rrrkey[KEYSIZ];         /*   key required to request return recp*/
     int rrrchg;                  /*   return receipt request surcharge   */
     char prikey[KEYSIZ];         /*   key required to send priority msg  */
     int prichg;                  /*   priority surcharge                 */
     int flags;                   /*   supported features flags           */
     char *(*helpmsg)(void);      /*   get help message for this exporter */
     BOOL (*valadr)(char *adr);   /*   is this a valid address vector     */
     char *(*attspc)(char *to,struct message *msg);
                                  /* path+filename for attachment vector  */
     int (*sndmsg)(char *to,struct message *msg,char *text,char *att);
};                                /*   send message vector                */
```

This structure gives basic information about the exporter (prefix, name, description, example address), the access restrictions and credit charges that apply to the exporter, message features supported by the exporter (e.g., attachments, return receipts), and functions to be used when sending a message using the exporter.

The prefix field is the unique prefix used to identify the exporter (without the colon). Examples of exporter prefixes are MHS for the MHS exporter and IN or US for MG/I.

The name field is the short name by which the exporter (or the mail system to which the exporter interfaces) is known. Examples of exporter names are Novell MHS for the MHS exporter and Internet for MG/I.

The desc field is used to provide a longer description of the exporter. The name and description are typically used by the user interface in lists of exporters or help messages that refer to exporters.

The exmp field would also be found in help messages. This field should give an example of a typical address that would be handled by this exporter (including the prefix). The example address used by MG/I is IN:stryker@gcomm.com.

All of this information (and the keys and credit charges discussed below) can be accessed by outside processes using the following function:

```
struct expinfo *                   /*   returns pointer to static buffer   */
expinf(                            /* get info on an exporter              */
int idx);                          /*   given exporter index               */
```

This function returns a pointer to a temporary buffer with the following format:

```
struct expinfo {                   /* info about exporter for interface    */
     char prefix[PFXSIZ];          /*   address prefix for this exporter   */
     char name[EXPNSZ];            /*   name of exporter (e.g., "Internet")*/
     char desc[EXPDSZ];            /*   description of exporter            */
     char exmp[MAXADR];            /*   example address to this exporter   */
     char wrtkey[KEYSIZ];          /*   key required to use this exporter  */
     int wrtchg;                   /*   per-message surcharge              */
     char attkey[KEYSIZ];          /*   key required for file attachments  */
     int attchg;                   /*   attachment surcharge               */
     int apkchg;                   /*   attachment per-kbyte surcharge     */
     char rrrkey[KEYSIZ];          /*   key required to request return recp*/
     int rrrchg;                   /*   return receipt request surcharge   */
     char prikey[KEYSIZ];          /*   key required to send priority msg  */
     int prichg;                   /*   priority surcharge                 */
     int flags;                    /*   supported features flags           */
};
```

Note that this structure contains all the information about the exporter except for the function vectors.

The exporter index parameter (idx) to expinf() is the index of the desired exporter in the internal array of registered exporters. This can be gotten for a specific exporter by using the following function:

```
int                               /*   returns NOIDX if not found        */
expidx(                           /* index of exporter in handler array  */
char *to);                        /*   address containing prefix         */
```

Note that the to parameter to expidx() can be any string or address that begins with the desired exporter's prefix (including the colon). If no exporters have been registered with the specified prefix, expidx() returns NOIDX.

One other thing to note about expinf() is that the list of exporters is stored in alphabetical order by prefix, so you can generate a list of exporters, in alphabetical order by prefix, by calling expinf() with indexes 0 through numexp()-1.

The helpmsg function vector is also used for help messages. It should return a pointer to a string that gives even more extensive information about the exporter. Outside processes can access this vector through the following GME function call:

```
char *                            /*   ptr to temp area (may be msgbuf)   */
exphlp(                           /* get help message for an exporter     */
int idx);                         /*   given exporter index               */
```

The pointer returned by the helpmsg vector (and thus by exphlp()) may point to vdatmp, prfbuf, or msgbuf — any general-purpose text buffer is acceptable (though not required) — so processes that use this function need to be aware of this possibility.

Which message features can be handled by the exporter are specified by the flags field. The following feature flags are currently defined:

```
                                  /* exporter flags                       */
#define EXPATT   0x0001           /*   support attachments?               */
#define EXPRRR   0x0002           /*   support return receipts?           */
#define EXPPRI   0x0004           /*   support priority msgs?             */
```

Thus an exporter that can handle attachments and return receipts, but not priority messages, would set flags=(EXPATT¦EXPRRR). These flags are used to limit the choices presented to a user when he writes a message to this exporter. In this case, the user might be given the option to attach a file or request a return receipt with his message, but would not be given the option to send a priority message.

A set of keys specified by the exporter control block are also used to determine what features a user may use when sending a message using the exporter.

The primary key is wrtkey. A user must have this key in order to send a message using the exporter (or even know the exporter exists).

The attkey, rrrkey, and prikey fields specify keys required to use each of the corresponding features (if the exporter supports that feature). Thus, a user who does not have the attkey for an exporter will not be given the option to attach a file to a message to be sent using that exporter.

The credit charge fields can be used to place a premium on the use of an exporter. These charges are surcharges, that is, they are added to regular message writing charges to compute the total charge to send a message.

For example, it normally costs 95 credits to send a plain e-mail message (no attachments, return receipts, etc.) to another user on the local system (using his User-ID as his to address). If an exporter specifies wrtchg=100, it will cost a total of 195 credits to send a plain e-mail message to an e-mail address on the remote system reached through that exporter.

The last three function vectors are used by the engine to perform various tasks when sending a message to the exporter.

The first vector, valadr, is used to validate addresses. When a user enters an address to send an e-mail message to, the engine checks to see if it is a forum, a distribution list, or an existing User-ID. If it is none of these, it checks the address for an exporter prefix. If a valid prefix is specified, the engine passes the address less the prefix to the specified exporter's valadr vector. This vector should examine the address and return TRUE if the address is in the correct format for the mail system accessed by the exporter. The valadr vector does not need to try to determine whether the address actually exists or not.*

*While it's possible for a Major BBS server to verify the existence of a local address since that's one and the same as a local User-ID, it is usually not possible to verify the existence of a remote address. There is no master User-ID list for the Internet, for example. The best that can be done is to verify the format of the address.

If a valid prefix is not specified, or the valadr vector for the specified exporter returned FALSE and the Sysop has set the GALME.MSG CNF option STRXCHK to **NO**, all of the exporters to which the user has access are polled (the address less prefix is passed to each valadr vector).

- If none of the exporters accept the address, it is rejected as invalid.
- If only one of the exporters accepts the address, the engine assumes the message should be sent using that exporter, and the prefix for that exporter is prepended to the address.
- If more than one of the exporters accepts the address, it is rejected as ambiguous.

To increase the accuracy of this exporter auto-sensing, and to prevent misdirected mail as much as possible, an exporter's valadr vector should be very critical of addresses passed to it and return FALSE whenever possible.

The valadr vector is used while the user is composing their message packet. The next two vectors are used as the message is being sent.

If the user has attached a file to the message, the engine calls the destination exporter's attspc vector. The parameters passed to this vector are the address to which the message is to be sent (including the exporter prefix) and the completed message header structure (including the message ID, thread ID, etc.). The attspc vector must return a valid path and filename. The engine stores this string, then copies the attachment to this path and filename. After copying the attachment to the specified destination, the engine submits the rest of the message packet to the exporter.

The attspc vector must return a unique path and filename so that the new attachment will not overwrite any existing files, and the exporter must not process the attachment before it receives the associated message.

Two GME functions are provided to assist the attspc vector in providing a unique attachment name:

```
char *                          /*   returns pointer to filename       */
tmpanam(                        /* create temp attachment filename     */
char *dest);                    /*   buffer for name (NULL for internal)*/
```

This function will generate up to 4 billion unique filenames, using either an externally-supplied buffer (which must be at least FLNMSZ in length) or a static internal buffer (selected by passing NULL for the dest parameter).

If you need to determine whether a given filename was generated using tmpanam(), you can call the following function:

```
BOOL
istmp(                          /* is this a temporary attachment file? */
char *fname);                   /*   path+filename to check             */
```

It will work whether the fname parameter is just a filename or a path\filename combination.

The message is actually submitted to the exporter using the sndmsg vector. The parameters passed to this vector are the address to which the message is to be sent (without the prefix), the completed message header structure, the message body text, and the attachment path and filename (if any).

There are several important things to note about the sndmsg vector. First, the to parameter should be used as the destination address, not the to field of the message header (the to field of the message header is generally only significant for echoed forum messages).

There are no provisions for cycling this vector, so it should do whatever processing it needs to do expeditiously. The att parameter used is an exact copy of what the attspc vector returned to the engine. Finally, the sndmsg vector should return one of the standard GME status codes to indicate the result of the operation. At a minimum, GMEOK should be used to indicate success and GMEERR to indicate failure.

Importers

An importer is simpler from the engine's point of view, but a bit more complex to implement than an exporter.

There is one function used to import messages into GME:

```
int                                /*   returns standard GME status codes  */
impmsg(                            /* import a message                     */
void *workb,                       /*   GME work space (provided by caller)*/
struct message *msg,               /*   new message structure              */
char *text,                        /*   message body text                  */
char *filatt);                     /*   path+filename of att (if any)      */
```

This function is very similar in use to gsndmsg() – the caller must initialize a work area, form a message packet, then cycle the function until it returns a "done" status code. Like gsndmsg(), it closes the work area when finished.

The primary source of complexity when using impmsg() is forming the message packet. First, the importer must convert the to addresses of messages it receives to valid local addresses (User-IDs or forum names, but importing to a distribution list is not supported). E-mail addresses must be converted to existing User-IDs (and the User-ID put in the to field of the message header).

Messages being imported into forums must be given valid forum IDs, and the message to field must be filled in with the address of the user to whom the message is directed (it need not be a local User-ID) or ** ALL **. A constant defined in GME.H can be used for this "all users in forum" string:

```
#define ALLINF    "** ALL **"      /* "to all in forum" string            */
```

Since most importers will be associated with an exporter (and the engine has no way of knowing about this association), the importer should format the from field of the message header so that a reply to the message will go to the appropriate exporter. This formatting must include prepending the appropriate prefix to the address, and formatting the address so that it will be recognized by the exporter's valadr vector.

Like gsndmsg(), it is up to the importer to correctly set all necessary message header flags. Be sure to set FILAPV when importing an e-mail message with file attachment.

In addition to the required header fields (forum, from, to, topic, flags) the importer has the option to explicitly set several other fields or let impmsg() do it:

- If the crdate and crtime fields are not both zero, their contents are used as the message creation date and time. Otherwise they are set to the date and time that the message is imported.
- If your mail handler can transmit the global IDs of the messages it carries, you should set the gmid field of the message header to this global ID. If the gmid field is all zeros, impmsg() will give the message a new global ID based on the current system and message IDs.
- If your mail handler transmits thread IDs, set the thrid field of the message header. If the thrid field is zero, it will be set to a thread ID based on the topic of the message.
- Even more important than the thread ID is the reply-to ID. If your mail handler can transmit reply-to IDs, set the rplto field of the message header to the global reply-to ID. If this field is not zeros, impmsg() will try to find a message with the global ID specified in the rplto field of the message being imported. If this original message is found, impmsg() will set the thread ID of the new message to the thread ID of the original message (and add "Reply to #..." to the history field). This is how global threading is achieved (see page 174 for a detailed discussion of global threading).
- Finally, if your mail handler transmits attachment names, you can set the attname field of the message header. Otherwise, impmsg() will set this field to the filename used to associate the attachment with the message (this will have the form "00012345.67A" where 1234567 is the message ID).

In general, you should setmem() to all-zeros any fields you do not want to explicitly set yourself. Only the msgid and nrpl fields will always be set by the engine.

Attachments are very easy to deal with. The caller need merely provide the current path and filename of the attachment in the filatt parameter, and the file will be copied to the correct path and filename by impmsg().

The following status codes may be returned by impmsg():

GMEAGAIN	Cycle, then call impmsg() again.
GMEOK	The message has been imported successfully.
GMEAFWD	The message has been imported successfully and has been auto-forwarded. The original addressee is in the to field of the message header, and the address to which the message was forwarded can be gotten by calling gmexinf().
GMEERR	The forum ID specified does not exist.
GMENFND	The User-ID to which an e-mail message was being imported does not exist.
GMEDUP	A message with the global ID specified by this message is already present on the system.
GMENOAT	The file attached to the message could not be imported for some reason (file not found, couldn't rename, I/O error, etc.).

Conflict Checking

Because all information about a GME request is stored in the work area, and the work area is owned by the caller, not the engine, GME relies on a system of cooperative conflict checking to avoid problems. Modules that use GME to read or write messages (including importers) should register conflict checking functions. Before a message or forum is deleted or modified, GME polls all registered conflict checking functions to see if anyone is currently using that message or forum. If anyone is using it, GME rejects the modify or delete request.

The function to register a conflict checker is:

```
typedef BOOL (*cflfunc)(void *work,unsigned forum,long msgid);

BOOL                              /*   returns TRUE if able to add to list*/
setcfl(                           /* set cooperative conflict checker     */
cflfunc cflchk);                  /*   function to check                  */
```

The parameters passed to a conflict checker are: a work area pointer, a forum ID, and a message ID. The work area parameter is the work area being used for the delete or modify request that is having its conflict checked. This is provided so your conflict checker can recognize if your module is the one doing the delete or modify request. The message ID is the message to be deleted or modified. If the message ID is zero, then it is the forum that is being deleted or modified, and its ID is passed in the forum ID parameter.

As a simple example, consider the following conflict checker, for a module which deals with one message at a time:

```
void *mywrkbuf;                   /* my GME request work area           */
unsigned myforum;                 /* the forum I'm currently working with */
long mymsgid;                     /* the msg I'm currently working with   */

void EXPORT
init__mymod(void)
{
     ...
     mywrkbuf=alczer(GMEWRKSZ);
     setcfl(mychecker);
     ...
}

BOOL
mychecker(                        /* my conflict checker                */
void *workb,
unsigned forum,
long msgid)
{
     if (workb != mywrkbuf) {
          if (msgid == 0L) {
               return(forum == myforum);
          }
          else {
               return(msgid == mymsgid);
          }
     }
     return(FALSE);
}
```

If you have a number of work areas that you use for reading messages, you may want to use this function:

```
BOOL                              /*   returns TRUE if a conflict        */
chkmycfl(                         /* check my current message for conflict*/
void *workb,                      /*   my work area being used to read   */
unsigned forum,                   /*   forum ID w/possible conflict      */
long msgid);                      /*   message ID w/possible conflict    */
```

It checks for conflict with the read context (see page 189) in the work area passed to it. This function can *only* be used to check work areas being used to read messages. If your module also writes messages, you will have to provide some other means for detecting conflicts with your write requests. To keep things somewhat simpler, it is only necessary to check for forum conflicts with write requests.

An example use of chkmycfl() is as follows:

```
#define NUMWORKS 10                /* how many GME requests I can do @ once*/
void *myworks[NUMWORKS];           /* my array of work area pointers       */

void EXPORT
init__mymod(void)                   /* init my module that reads messages   */
{
     int i;
     ...
     for (i=0 ; i < NUMWORKS ; ++i) {
          myworks[i]=alczer(GMEWRKSZ);
     }
     setcfl(mychecker);
     ...
}

BOOL
mychecker(                         /* my conflict checker                  */
void *workb,
unsigned forum,
long msgid)
{
     int i;

     for (i=0 ; i < NUMWORKS ; ++i) {
          if (myworks[i] != workb && gmerqopn(myworks[i])) {
               if (chkmycfl(myworks[i],forum,msgid)) {
                    return(TRUE);
               }
          }
     }
     return(FALSE);
}
```

To check for any conflicts with the current message in your read context before beginning a modify or delete request, use the following function:

```
BOOL                               /*   returns TRUE if a conflict         */
chkcfl(                            /* chk others for conflict w/my cur msg */
void *workb);                      /*   work area being used to read       */
```

To check for conflict with any arbitrary message or forum, you can use the following function:

```
BOOL                               /*   returns TRUE if a conflict         */
gencfl(                            /* generic conflict checker             */
void *workb,                       /*   work area in use                   */
unsigned forum,                    /*   forum ID to check                  */
long msgid);                       /*   msg ID to check (0L for forum only)*/
```

Note: You should always pass a valid (though not necessarily initialized) work area to this function.

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 the ??GCOMM.LIB files:

```
printf(ctlstg,p1,p2,...,pn);
                              substitute for standard printf()
char *ctlstg;           control string, functions supported are %s,%c,%d,%u,%x,
                              all of them with field length, zero or blank fill,
                              left/right justification options
TYPE p1,p2,...,pn;      just like printf/cprintf parms
                              (note: no longs or floats)
```

There is no limit to the number of parameters (p1,p2,...,pn) that you may pass to printf(). They should correspond one-for-one with the % directives in the control string. See page 246 for a description of the ANSI graphics capability of printf().

```
setatr(attrib);         sets video attributes
int attrib;             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:

Foreground Attribute +	**Background Attribute +**	**Blink**
`0x00` Black	`0x00` Black	`0x00` non-blinking
`0x01` Dark blue	`0x10` Dark blue	`0x80` blinking foreground
`0x02` Green	`0x20` Green	
`0x03` Cyan	`0x30` Cyan (blue-green)	
`0x04` Red	`0x40` Red	
`0x05` Magenta	`0x50` Magenta	
`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		
`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 clreol() function also sets the background color for the remainder of the line according to the latest setatr() attribute.

See page 248 for converting IBM screen attribute codes into ANSI color coding sequences.

See page 274 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 coordinate
int yul;	upper left y coordinate
int xlr;	lower right x coordinate
int ylr;	lower right y coordinate
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;	segment:offset start address 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:

```
scblank(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() below). 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,p1,...,pn);
combination of locate() and printf() into one routine (saves code space)
int x,y; screen coordinates
char *ctlstg; control string
p1,...,pn parameters (max 8 bytes)

This routine is like printf(), except that the control string is preceded by a screen position. Parameters (p1,...,pn) can be no more than 8 bytes.

The following routines are similar to printfat(), except you can specify an origin and relative offsets:

proff(xbase,ybase); set origin for prat() locations
int xbase,ybase;

prat(x,y,ctlstg,p1,...,pn);
printf() at a relative point on the screen
int x,y; screen coordinates (relative to the most recent proff() setting)
char *ctlstg; control string
p1,...,pn parameters (max 12 bytes)

The explode() family of routines (page 272) calls proff() to set the upper left corner of the exploded region. Then the choose() family (page 277) and edtval() (page 275) 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 (CTRL+G) 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 beeps for signup notification (SGNBEL), the page-Sysop warning (SOPBEL), and the emulation screen (EMUBEL).

ansion(on);	enable/disable ANSI graphics
	1=process ANSI graphics sequences
	0=ignore ANSI graphics sequences (display as literal)

This function turns on or off printf()'s interpretation 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:

ANSI Commands

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 [2J	Erase display
ESC [K	Erase to the end of the current line
ESC [0m	Normal display attribute
ESC [1m	Bold display attribute
ESC [4m	Underscore display attribute

ESC [5m	Blink display attribute
ESC [7m	Reverse display attribute
ESC [8m	Invisible display attribute
ESC [30m	Black foreground
ESC [31m	Red foreground
ESC [32m	Green foreground
ESC [33m	Yellow foreground
ESC [34m	Blue foreground
ESC [35m	Magenta foreground
ESC [36m	Cyan foreground
ESC [37m	White foreground
ESC [40m	Black background
ESC [41m	Red background
ESC [42m	Green background
ESC [43m	Yellow background
ESC [44m	Blue background
ESC [45m	Magenta background
ESC [46m	Cyan background
ESC [47m	White background

None of the above commands include any spaces.

ESC	the Escape character, ASCII 27
<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. The **;** may be omitted if the <column> parameter is 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 **m** commands (display attributes) may be combined using semicolons. For example, ESC [1m ESC [33m ESC [45m has the same effect as ESC [1;33;45m Both commands set the display attribute to bright yellow on magenta. The following code displays a message with these attribute settings in the current display window:

```
printf("\33[1;33;45mWafer yield for the 200MHz P6:  85%");
```

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 249.

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 241)
char *buffer;	where to put ANSI sequence (up to 15 bytes, including terminator)

The following code 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 setatr.

Keyboard Input: getchc()

```
yes=kbhit();          Has the operator pressed 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 '~' (printable ASCII characters 20 to 7E)

\x00 through \xFF (all ASCII plus extended 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:

cursiz(howbig);		Set the size of the video cursor
int howbig;	NOCURS	cursor disappears
	LILCURS	small standard cursor
	BIGCURS	big insert-mode cursor
rstcur();		Restore the cursor to the size it was before the last cursiz()
howbig=curcurs();	Find out how big the current cursor is.	
int howbig;	NOCURS, LILCURS, or BIGCURS	

Operator Services

Statistics

You can add your own statistical graphs to those already available on the operator console. There are two parts to this:

- Generating the graph
- 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 Major BBS server is on the air.

So, to register your screen:

1. Create one copy of the statsc data structure in your online code. This structure is defined in STATSCNS.H. Here is an example, with the blanks filled in:

```
struct statsc mygraph={        /* statistic screen interface structure   */
     "Sat activity",           /* name of statistic screen               */
     "DDDSTA1.BIN",            /* filename 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               */
};
```

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.

These routines do:

void (*inirou)();	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.
unsigned (*keyhit)(unsigned scncod);	This routine is called with each and every keystroke from the Sysop when your screen is on display. The parameter if the routine is the same as the getchc() return value (page 250). 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.
void (*occrou)();	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 data.
void (*cycrou)();	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.

void (*finrou)();	This routine gets called when your display goes away. It gets called once for each call to the inirou() 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 63 characters long. The parameters passed can take up to 12 bytes on the stack.

This function makes 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
-3	Event N (1 to 4)
-2	Cleanup
-1	Console
0..nterms-1	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.

Channel Status Reporting

shochl(legend,sing,attr);
Show a line on the Online User Info screen

char *legend; information, up to 29 characters

char sing; single-character indicator for the user matrix (Summary screen too)

int attr; IBM color display attribute

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

int attr; IBM 8-bit display attribute

unsigned baud; baud rate, 300 to 38400

int blink; 1=give us a blinking attribute

Databases

Database Functions: xxxbtv()

Btrieve, by Btrieve Technologies, 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;                /*  filename                            */
     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, the catastro() error message BTRIEVE OPEN ERROR 12 is generated automatically by opnbtv() — you never have to check the return value for error conditions.

Btrieve allows you to create Btrieve files with an *owner* for security over who can open the file. You can only open such a file if you specify the file's owner name. The opnbtvl() function can be used instead of opnbtv() to specify an owner name when opening a Btrieve data file:

```
agtbb=opnbtvl("ZZZAGENT.DAT",maxlen,owner);
```

The owner parameter is a NUL-terminated, case sensitive string between 0 and 8 characters in length. The return value and other parameters are the same as for the standard opnbtv() function.

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 xxxbtv 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() use a stack and 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=obtbtv(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 obtopt;	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
gabbtv(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=aabbtv(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 gabbtv() or aabbtv() 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 the database
int is;	1=there was one 0=database empty
is=snxbtv(recptr);	Read the physically next record in the database
int is;	1=there was one 0=already read last
is=sprbtv(recptr);	Read the physically previous record in the database
int is;	1=there was one 0=already read first
is=shibtv(recptr);	Read the physically last record in the database
int is;	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 below) is their speed: physical access can be about eight times as fast as keyed-sequential.

Database Update Routines

updbtv(recptr);	update current record
char *recptr;	replacement record buffer ptr (NULL to use bbptr->data)
ok=dupdbtv(recptr);	(more tolerant) update current record
int ok;	1=updated 0=duplicate collision
char *recptr;	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():

upvbtv(recptr,length);	update variable length record
char *recptr;	replacement record buffer ptr NULL to use bbptr->data)
int length;	number of bytes for new record contents

Database Insert Routines

insbtv(recptr);	insert new fixed-length record
char *recptr;	new record buffer ptr (NULL to use bbptr->data)
ok=dinsbtv(recptr);	(more tolerant) insert new record
int ok;	1=inserted 0=duplicate collision
char *recptr;	new record buffer ptr (NULL to use bbptr->data)

insbtv() will automatically generate the 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.

invbtv(recptr,length);	insert variable length record
char *recptr;	new record buffer ptr (NULL to use bbptr->data)
int length;	number of bytes for new record

Deleting a Database Record

delbtv();	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

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=qeqbtv(key,keynum);	query for "equal to" specified 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 sequence
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 specified key
char *recp;	destination record buffer pointer
char *key;	key specification
int keynum;	key number involved
gnxbtv(recp);	get next record in sequence
char *recp;	destination record buffer pointer
gprbtv(recp);	get previous record in sequence
char *recp;	destination record buffer pointer
ggtbtv(recp,key,keynum);	get first record > specified key
char *recp;	destination record buffer pointer
char *key;	key specification
int keynum;	key number involved
ggebtv(recp,key,keynum);	get first record >= specified key
char *recp;	destination record buffer pointer
char *key;	key specification
int keynum;	key number involved
gltbtv(recp,key,keynum);	get highest record < specified key
char *recp;	destination record buffer pointer
char *key;	key specification
int keynum;	key number involved

glebtv(recp,key,keynum); get highest record <= specified key
char *recp; destination record buffer pointer
char *key; key specification
int keynum; key number involved

globtv(recp,keynum); get lowest record present
char *recp; destination record buffer pointer
int keynum; key number involved

ghibtv(recp,keynum); get highest record present
char *recp; destination record buffer pointer
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->key);
     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 256) 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 specified key
int is;	1 if record exists, else 0
char *recptr;	destination record buffer pointer
char *key;	key value to search for
int keynum;	key number

is=agtbtv(recptr,key,keynum);	
	acquire first record > key
int is;	1 record exists, else 0
char *recptr;	destination record buffer pointer
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 pointer
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=alebtv(recptr,key,keynum);	
	acquire highest record <= key
char *recptr;	destination record buffer pointer
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 pointer
int keynum;	key number involved

is=ahibtv(recptr,keynum); acquire highest record in database

int is;	1 record exists, else 0
char *recptr;	destination record buffer pointer
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->key);
     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=aqnbtv(recptr); acquire next record in sequence

int is;	1 if another record exists, else 0
char *recptr;	destination record buffer pointer

is=aqpbtv(recptr); acquire previous record in sequence

int is;	1 if previous record exists, else 0
char *recptr;	destination record buffer pointer

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 Btrieve Technologies, 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 run-time, 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 {                   /* system-variable btrieve record layout */
     char key[4];                 /* 4-character dummy key of "key"        */
     char dspopt[6];              /* display options by position number    */
     long calls[8];               /* number of calls this month/baud rt    */
     char lonmsg[MTXSIZ];         /* log-on message in effect              */
     long dwnlds;                 /* total downloads to date               */
     long uplds;                  /* total uploads to date                 */
     long msgtot;                 /* msg (E-mail/Forums) total to date     */
     unsigned emlopn;             /* E-Mail open at the moment             */
     unsigned sigopn;             /* Forum messages open at the moment     */
     int hisign;                  /* highest Forum number used to date     */
     char monmal;                 /* Aux. CRT display selector             */
     char savmin;                 /* Minutes to save screen                */
     long oldsec[8][24];          /* old sec/grp/hr (now in sv3.secghr)    */
     char spare[1300-1230];       /* spare space for graceful upgrades     */
} sv;

extern
struct sysvb2 {                   /* second system variable btrieve layout */
     char ky2[4];                 /* 4-character dummy key of "ky2"        */
     unsigned matrix[NCOMTY][NAGEBK]; /* matrix of accts (computer/age)    */
     long oldcrd[8][24];          /* old crd/grp/hr (now in sv3.crdghr)    */
     int nliniu[48];              /* number of lines in use per hlf/hr     */
     int lstzer;                  /* date of last zeroing of stats         */
     long x25kps;                 /* X.25 kilopackets sent or received     */
     unsigned x25ps;              /* fractional X.25 kilopackets           */
     long x25mbs;                 /* X.25 megabytes sent or received       */
     long x25bs;                  /* fractional X.25 megabytes             */
     unsigned numact;             /* total number of user accounts         */
     unsigned numfem;             /* number of female users                */
     unsigned numcor;             /* number of corporate users             */
     unsigned numans;             /* number of ANSI users                  */
     unsigned long paidpst;       /* credits paid-for so far               */
     unsigned long freepst;       /* credits given away free so far        */
     long totcalls;               /* total calls-to-date                   */
     int lastmcu;                 /* date of last midnight cleanup         */
```

```
     char spare[1300-986];        /* spare space for graceful upgrades     */
} sv2;

extern
struct sysvb3 {                   /* third system variable btrieve layout  */
     char ky3[4];                 /* 4-character dummy key of "ky3"        */
     long secghr[NGROUPS-1][24];  /*seconds used (channel grp/hr)          */
     long crdghr[NGROUPS-1][24];  /*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 93.

```
struct usracc {
     char userid[UIDSIZ];         /* user-id                                   */
     char psword[PSWSIZ];         /* password                                  */
     char usrnam[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 scnfse;                 /* 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[USRACCSPARE];     /* spare space, for graceful upgrades        */
};

                                  /* ansifl bit definitions                    */
```

```
#define ANSON     1             /* ANSI on=1; off=0                        */
#define ANSMAN    2             /* ANSI manual override (0=auto sensing)   */
                                /* flags bit definitions                   */
#define HASMST    1             /* user has the "MASTER" key               */
#define UNDAXS    2             /* this account cannot be deleted          */
#define SUSPEN    4             /* this account is "suspended"             */
#define DELTAG    8             /* this account is tagged for deletion     */
#define GOINVS   16             /* this account is "invisible" upon logon  */
                                /* usrprf bit definitions                  */
#define PRFLIN    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 Operator Menu ACCOUNT submenu, CLASS command.

```
extern
struct acclass {                /* accounting class structure            */
     char clname[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 dbtlmt;               /* debt limit (0=none)                   */
     int fgvday;                /* 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 msgs[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 Galacticomm 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 Galacticomm 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 The Major 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 own system, 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 Major BBS, Galacticomm 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 Galacticomm 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 bggame {                /* generic user data records for my game */
     char userid[UIDSIZ];      /* User-ID                               */
     char modnam[MNMSIZ];      /* Module Name ("My Game")               */
     int score;                /* score                                 */
};
```

To store a record for the current user, with a score of 50, you could code:

```
struct bggame 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 descrp field of your module structure (see page 25 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");
}
```

This sure beats making a whole separate .DAT file for one measly integer.

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 241.

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 run-time, it uses iniscn():

```
iniscn(filspc,where);   Read an 80x25 character color screen from a .BIN file
char *filspc;           DOS path for the file
void *where;            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 241 about frzseg()).

See page 274 for what iniscn() does on monochrome screens.

Window Output: explode()

To make a window pop up on the screen, we use explode():

```
explode(sctptr,wulx,wuly,wlrx,wlry);
                                Pop up a window on the CRT
char *sctptr;           screen image (from iniscn())
int wulx,wuly;                  window upper left corner
int wlrx,wlry;          window lower right corner (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);
Pop up a window on the CRT

char *sctptr;	screen image (from iniscn())
int fux,fuy;	source window upper left corner
int flx,fly;	source window lower right corner (inclusive)
int tux,tuy;	destination 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 245 about proff() and prat()). This affects future operation of the choose() and edtval() 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

Monochrome Compatibility

To be compatible with both monochrome and color screens, you can call monorcol():

monorcol()	Determine monochrome versus color, based on the Hardware Setup CNF option CRT (which is set to COLOR, MONO or AUTO)
imonorcol()	Determine monochrome versus color based on BIOS settings only. This is equivalent to monorcol() 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.
int color;	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:

- 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.

save=edtval(sx,sy,maxlen,sval,valrou,flags);	edit a string field on the screen
int save;	0 means ESC was pressed, 1 means ENTER, TAB, SHIFT+TAB, UP, or DOWN was pressed
int sx,sy;	starting point for the field on the screen (sx is 0-79, sy 0-24)
int maxlen;	maximum size of string (including NUL — maxlen-1 is field width)
char *sval;	default value / return value
int (*valrou)();	validation routine
int flags;	options

The sx,sy coordinates are relative to the tux,tuy coordinates of the most recent call to the explode() family of routines (page 272). This is done via the proff() and prat() routines (page 245).

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 edtval():

```
#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 MULTIEX  8  /* allow multiple field-exit conditions  */
```

The operator can type in a new value, move the cursor RIGHT or LEFT, INS or DEL characters, hit HOME or END, and when finally done, press 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 edtval()'s return value. You can distinguish among the other cases using the edtvalc global variable:

int edtvalc;	Keystroke that ended edtval()

After edtval() returns, you can get the entry results in the buffer that the sval parameter pointed to.

While edtval() 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 edtval() 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 presses 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 250) 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 (as in 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 (a la 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.

Multiple Choice

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);
Pop up a window of choices

int choice;	index of choice 0..nchoices-1 or -ESC if operator escaped
int nchoices;	number of choices
char *choices[];	array of choices
int upx,upy;	upper left corner of window
int lox,loy;	lower right corner of window
int escok;	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 272). This is done via the proff() and prat() routines (page 245).

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);	Pop up a window of choices
int choice;	index of choice 0..nchoices-1 or -ESC if operator escaped
char *choices[];	array of choices, after the last of which is a NULL
int first;	index of the default choice
int upx,upy;	upper left corner of window
int lox,loy;	lower right corner of window
int escok;	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 Developer's Kit.

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);        <- this line is optional
dspchc();
cursiz(NOCURS);
do {
     choice=hdlchc(getchc());
while (choice == nchoices);
rstcur();
```

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 Galacticomm 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's Client/Server Developer's Kit:

BBSRPT.MAK	Offline Reports (choice **9** on the Introductory Menu, includes
GALP&QR.MAK	Offline Polls & Questionnaires analysis (choice **7** on the Intro

BBSRPT.MAK uses protected mode due to its use of the messaging engine, but GALP&Q.MAK is in large model, as are most offline utilities. Many more .MAK files come with the Extended C Source Developer's Kit.

Here are the most important differences in large model programming versus protected mode programming:

- Smaller memory limits on the program (640K or so total, up to 64K static data)
- Object files reside in \WGSERV\LOBJ instead of \WGSERV\PHOBJ.

To do the compiling and linking steps piece by piece:

Compile a <filename>.C source file that contributes to an .EXE offline utility program:

CD \WGSERV\SRC	or	**CD \WGSERV\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.

Relink a <utility name>.EXE file:

CD \WGSERV\LOBJ

LNK <utility name> <other file 1> <other file 2>

or, as appropriate

CD \WGSERV\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 WGSDRAW for all languages that end in /ANSI, or RIP*aint* 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 307 for more on running in protected mode.

To create your own language editor .DLL:

1. In your language .MDF file (page 13), 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 "I'm supposed to edit the text," or "No, 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 (!sameto("TESTIT.DLL",command)) {
          return(EONOTME);
     }
     return(edit(txtbuf,sizbuf) ? EOSAVE : EONOCHG);
}
```

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 presses F2 to 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 get to work (in the above example, to call edit(), and return either EOSAVE or EONOCHG) or 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 filename by the time it gets to you in the form of this command parameter. You probably don't care about that filename 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
EONOCHG	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
EONOCHG+EOPGUP	No change to data, skip to option above
EONOCHG+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 EONOCHG, 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.

For example:

```
void EXPORT
init__testit(regrou)
void (*regrou)(EDTHANDLER *edthdl);
{
     (*regrou)(tstedt);
}
```

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 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. For example, TESTIT.LNK:

```
%PHARLAP%\bc4\lib\c0phdll    +
%MBBS%\phobj\testit
%MBBS%\testit
nul
plhide phapi cnfimp /Twd /s /n
%MBBS%\dlib\nodef
```

And use it to make your .DLL file:

```
LTDLL TESTIT
```

If you get undefined symbols for Borland Library routines, you may have to find alternatives to using those routines. See page **Error! Bookmark not defined.** about the perils of linking BCH286.LIB in a .DLL.

6. Install your .DLL file, and the language .MDF file, on The Major BBS server computer 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 (not including types **S** or **T**)

To read the option from the .MCV file, use the getmsg() and xxxopt() routines, described starting on page 69. 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("BBSSUP.MSG");
```

When you bring The Major BBS server up again, new .MCV files will automatically be generated by WGSMSX.

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.

More Routines And Variables

Character and String Routines

match=sameas(stg1,stg2);	case-*in*sensitive string match
int match;	return code: true if strings are same
char *stg1,*stg2;	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);	
	case-*in*sensitive substring match
int match;	true if shorts = first part of longs
char *shorts;	short string: entirety must match char
char *longs;	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("baddest","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 samend(), 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 sameto("beg","beginning") and samend("ending","ing") are both true.

found=samein(subs,string);

Search a string for a case-*in*sensitive substring

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 pointer (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 destination, 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 NULs. 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).

stlcpy() is the same as stzcpy() except that it doesn't bother to pad with more than 1 NUL:

```
stlcpy(dest,source,maxbytes);
```

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=skpwrd(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(nyse),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.

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 a 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

oddparity=odd(somebyte);	Compute odd parity
char somebyte;	a byte
char oddparity;	a byte with odd parity and with the same lower 7 bits as somebyte

rmvwht(string);	Remove all whitespace characters
char *string;	string (conversion is in-place)

nremoved=depad(string); Remove trailing blanks from string
int nremoved; number of blanks removed
char *string; NUL-terminated string

bufptr=unpad(string); Remove trailing blanks from string
char *bufptr; copy of string pointer
char *string; NUL-terminated string

sortstgs(strings,nstrings);
Sorts a bunch of strings by rearranging an array of pointers
char *strings[]; array of pointers to the strings
int nstrings; number of strings

See also the memory handling routines movmem(), setmem(), and repmem() on page 48.

Real-Time Routines: rtkick(), rtihdlr()

rtkick(time,rouptr); "kick off" routine after time delay
int time; number of seconds before "kickoff"
void *rouptr(); pointer to routine to be kicked off

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.

rtihdlr(rouptr); Register a real-time routine
void (*rouptr)(void); routine to call at 18 Hz

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 Major BBS server 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 *GSBL Guide*). And be sure the routine uses as short a time as possible, or the server will lose its real-time responsiveness.

dsairp()	Disable interrupts
enairp()	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.

tix64k=hrtval();	Read the free-running 65KHz timer
unsigned long tix64k;	upper word counts seconds, lower word counts 1/65536 seconds

This routine reads the GSBL long integer variable btuhrt while interrupts are disabled, in order to avoid skew. You should almost always use hrtval() in place of referencing btuhrt. 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

date=today(); Find out today's date
int date; YYYYYYYMMMMDDDDD coding for today

This returns today's date in the format that DOS uses for dates:

YYYYYYYMMMMDDDDD Where...
YYYYYYY--------- (Year-1980) * 512 Representing 1980 through 2107
-------MMMM----- Month * 32 Representing 1 through 12
-----------DDDDD Day of month Representing 1 through 31

day=daytoday(); Find out what day of the week it is
int day; 0=Sunday 6=Saturday

time=now(); Find out what time of day it is
int time; HHHHHMMMMMMSSSSS coding for time

This returns the time of day in the format that DOS uses for time:

HHHHHMMMMMMSSSSS Where...
HHHHH----------- Hour * 2048 Representing 0 through 23
-----MMMMMM----- Minute * 32 Representing 0 through 59
-----------SSSSS 2-second intervals Representing the even numbers 0 through 58

ascdat=ncdate(date); Encode date into MM/DD/YY
char *ascdat; local buffer for date
int date; YYYYYYYMMMMDDDDD coding (see above)

asctim=nctime(time); Encode time into HH:MM:SS
char *asctime; local buffer for time
int time; HHHHHMMMMMMSSSSS coding (above)

ascdat=ncedat(date); Encode date into DD-MMM-YY (European style)
char *ascdat; local buffer for date
int date; YYYYYYYMMMMDDDDD coding

date=dcdate(ascdat); Decode date from MM/DD/YY

int date; YYYYYYYMMMMDDDDD coding or -1=invalid date format

char *ascdat; date string

time=dctime(asctim); Decode time from HH:MM:SS

int time; HHHHHMMMMMMSSSSS coding or -1=invalid time format

char *asctime; time string

count=cofdat(date); Count of days since 1/1/80

int count; number of days since 1/1/80

int date; YYYYYYYMMMMDDDDD coding

date=datofc(count); Compute DOS date

int date; YYYYYYYMMMMDDDDD coding

int count; number of days since 1/1/80

See also page 297 for reading and setting a file's time and date.

Numeric Routines

smaller=min(a,b); Find the smaller of two numbers

bigger=max(a,b); Find the larger of two numbers

Since these are implemented as macros, the numbers can be all int, all unsigned, or all long.

absval=abs(signednum); Find the absolute value of a signed integer (int or long)

newcrc=calcrc(oldcrc,ch);

Iteratively calculate a 16-bit CRC

int newcrc; CRC on N+1 bytes

int oldcrc; CRC on N bytes

char ch; N+1'th byte

This routine calculates a 16-bit CRC based upon the polynomial x^16+x^12+x^5+1. 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 (available with the Extended C Source Developer's Kit).

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 server 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 Developer's Kit).

But the text file scanning routines can be real handy for offline processing, for initializing things before the server 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 its 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 tfspfx() to find it and to isolate what follows the prefix:

found=tfspfx(prefix);	Is the current line prefixed with this?
char *prefix;	Prefix string
int found;	1=yes, 0=no. if yes, tfspst points to what follows the prefix
char *tfspst;	pointer to string following the prefix, with preceding white space removed

For example, if the line you read in was DLL=GALBLAST, then tfspfx("DLL=") would return 1 and tfspst would point to GALBLAST.

Suppose you needed to scan all .MDF files on the server 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() != TFSDUN) {
          switch(tfstate) {
          case TFSLIN:
               if (tfspfx("Module Name:")) {
                    modnam(tfspst);
               }
               else if (tfspfx("Developer:")) {
                    develn(tfspst);
               }
               break;
          }
     }
}
```

Notice how one or more spaces may follow the colons in the .MDF files, but they are skipped by tfspst.

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 (Extended C Source Developer's Kit) to isolate all of the Language lines in the .MDF files with one tfspfx() call, and then handle them individually with more tfspfx() 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 TFSDUN.

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.

```
got=fgetstg(buffer,size,fp);
                             Read a NUL-terminated string from file
int got;                     1=got it ok, 0=error or EOF
char *buffer;        where to store the line
int size;                    maximum size, including \0
FILE *fp;                    file (from fopen())
```

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].

nbytes=getdfre(diskno);	Find out how many bytes are free on disk
long nbytes;	number of bytes
int diskno;	disk number (0=default 1=A: 2=B:)

cntdir(path);	Count a set of files
char *path;	path specification (wildcards ok)
long numfils;	total number of files counted
long numbyts;	total number of bytes counted
long numbytp;	greater number of bytes occupied by the files, considering cluster size

drive=drvnum(path);	Determine drive number from the path
int drive;	0=current, 1=A:, 2=B:, 3=C:,...
char *path;	any DOS file specification

clbyts=clsize(drive);	How big are the clusters on disk?
unsigned clbyts;	number of bytes per cluster
int drive;	0=current, 1=A:, 2=B:, 3=C:,...

realsiz=clfit(siz,clbyts);	Total space used by a file
long realsiz;	siz rounded up to the next cluster
long siz;	useful size of file
unsigned clbyts;	cluster size (from clsize())

ok=setdtd(fname,time,date);	Set the time and date for a file (file must not be open at the time)
int ok;	1=ok 0=couldn't find file
char *fname;	file path
int time;	HHHHHMMMMMMSSSSS coding (see page 293)
int date;	YYYYYYYMMMMDDDDD coding (see page 293)

ok=settnd(fname,gmt70);	Set the time and date for a file (file must not be open at the time)
int ok;	1=ok 0=couldn't find file
char *fname;	file path
long gmt70;	Number of seconds since midnight 1/1/1970, GMT

timendate=getdtd(handle);	Get a file's time & date
long timendate;	32-bit number encoding date<<16 and time: HHHHHMMMMMMSSSSS coding (see page 293) YYYYYYYMMMMDDDDD coding (see page 293)
int handle;	file handle from an fopen()'d file (returned by fileno(fp))

gmt70=gettnd(handle);	Get a file's time & date
long gmt70;	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);	Extract 1-8 char file root
char *root;	extension and path prefix removed
char *filnam;	filename or entire file path

fnam=fnwext(filnam);	Extract filename with extension
char *fnam;	name after the path prefix
char *filnam;	filename or entire file path

rsvd=rsvnam(filnam);	Check if a DOS filename is reserved
int rsvd;	1=reserved, don't use, 0=ok to use
char *filnam;	filename or complete file path

This routine checks through the DOS list of device drivers and other reserved names to make sure a proposed filename will not be misinterpreted as a device. It's a great idea to use this routine especially if the filename 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.

Find First Matching File, Find Next

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=fnd1st(&fb,filespec,attr);	Any files in this filespec?
int yes;	1=one or more, 0=none
struct fndblk fb;	findblock structure
char attr;	attribute mask (see below)

yes=fndnxt(&fb);	Any more files in this filespec?
int yes;	1=yes 0=no more
struct fndblk fb;	same structure passed to fnd1st()

The fnd1st() and fndnxt() functions make use of the following data structure, defined in DOSCALLS.H:

```
struct fndblk {              /* used by fnd1st,fndnxt in datntim.c    */
     char junk[21];          /*                                       */
     char attr;              /* file attribute (see masks below)      */
     unsigned time;          /* time in HHHHHMMMMMMSSSSS format       */
     unsigned date;          /* date in YYYYYYYMMMMDDDDD format       */
     long size;              /* size in bytes                         */
     char name[12+1];        /* name of file "FFFFFFFF.EEE"           */
};
```

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 fnd1st() restricts the search to certain types of files (or more accurately directory entries — fnd1st() and fndnxt() are really directory scanning routines). Here are the possible bit components to the attr parameter of fnd1st(), 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 fnd1st() are 0, meaning roughly to scan for files only, not subdirectories or volume IDs, and FAMDIR, meaning to check if a given name exists as either a directory or a file. For example, if there is a file named SRC, fnd1st(&fb,"SRC",FAMDIR) will report that it exists even though you were anticipating it finding a directory by that name. To check, look at fb.attr for the FAMDIR flag. If set, it is a directory. If not set, it may be any of the other choices (volume id, hidden, etc.).

You can refer to a DOS technical manual on interrupt 21H, function 4EH, for details on the other bits.

Everything Else

cpu=cputype();	Returns 88, 186, 286, 386
int cpu;	depending on what type of processor you run it on (486s and above return 386)
setcrit();	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.
pascrit();	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).

Reliability

Debug Versions

You can build and run debug versions of the baseline and some or all add-ons during testing or when otherwise tracking down problems. There's a small performance loss, so everyone doesn't want to run debug versions of software all of the time. Once a product is released, it is generally released as a non-debugging release.

As a developer, you can take advantage of debug-only code in the debug versions of your .DLLs. In addition to writing basic debug-only code, you can make use of the debug-only tools described below. By simply building the debug version of the baseline, you can take advantage of the debug code Galacticomm included in the baseline.

See page **Error! Bookmark not defined.** for details on building the debug version of the baseline. Also, see page **Error! Bookmark not defined.** for details on making your own .MAK and .LNK files in such a way as to support a debug version. If you follow those guidelines, you can build a debug version of your own .DLL.

To write code that will only execute when it's compiled for the debug version, use the DEBUG define:

```
#ifdef DEBUG
    for (i=0 ; i < MAXLST ; i++) {
        if (list[i] == NULL) {
            catastro("NULL list entry!");
        }
    }
#endif
```

Assertions

An easy to use debugging tool is the *assertion*. Assertions are failures triggered by the ASSERT() macro. ASSERT() is used to assert that a condition is always true. When DEBUG is not defined, ASSERT() calls are not evaluated. When DEBUG is defined, ASSERT() will fail if the passed condition is not true. In the failure message will be the filename, file version, line number, and specific condition that failed. By default, assertion failures take the form of catastro() crashes. But, assertions can be altered to fail in different ways, as described below.

Using an ASSERT is an excellent way to prove that assumptions made in the code are indeed correct assumptions: parameters are within range and things are generally the way you expected them to be. Not only do the ASSERT() calls help track bugs by failing from time to time, but they also help those reading the code understand the assumptions being made by the developer.

If you would like to use ASSERT() in one of your C files, you'll first need to define the FILREV symbol at the top of it as the version number of the file. If your version control system doesn't offer the ability to automatically fill in text with version numbers, you'll have to maintain this number yourself:

```
#define FILREV "1.0.4.1"
```

To use ASSERT(), just call it like a function, passing it a condition to test:

```
ASSERT(ptr != NULL);
```

The ASSERTM() macro is the same as ASSERT(), except it allows you to specify a special message to appear in resulting failure messages in place of the condition string:

```
ASSERTM(ptr != NULL,"Developer passed NULL to rsp2read()!");
```

It's critical that you are careful not to write any ASSERT()s or other debug-only code that will alter the manner in which non-debug code will run. For example:

```
ASSERT(i++ != 3);
```

In this example, the debug version will increment i, the non-debug version won't. The following code may rely on i being incremented, and work fine as long as you're running the debug version. Once you build final, non-debug shipping disks, you'll be surprised to find a new bug. Bugs caused by debugging code are often the most painful.

Phase Transitions

Another helpful method of debugging involves using *phase transitions*. Phase transitions are used to track the basic flow of code by recording major events. The BEG_PHASE() and END_PHASE() macros are used to record the beginning and end of phases or events. The last x phase transitions are reported in catastro() and GP crashes to help understand what led to the crash. BEG_PHASE() and END_PHASE() both take a string and a long for a brief description of the phase and any relevant data. Generally, they are used to bracket some well-defined set of code:

```
BEG_PHASE("Status handling",0);
(*(module[usrptr->state]->stsrou))();
END_PHASE("Status handling",0);
```

In the above case, the phase transitions bracket a module's handling of a status code. BEG_PHASE() and END_PHASE() calls don't necessarily have to have a one-to-one correspondence:

```
BEG_PHASE("Global commands",0);
for (i=0 ; i < nglobs ; i++) {
    rc=(*globs[i])();
    END_PHASE("Global command",globs[i]);
    :
    :
}
```

In the above case, the address of the global command just executed is included as the extra parameter to END_PHASE(). This will report the address of the global command just executed along with the Global command end transition message.

Debug Options in CNF

If you create any debug-only CNF options, put them in CNF's level 95. That's the level where all debugging options go by default. A Sysop can edit those options by running CNF directly from DOS with a special command-line parameter:

```
CNF -L95
```

The Baseline's Debugging Module

The Debugging in baseline module is installed automatically as part of the Client/Server Developer's Kit. By default, it's disabled. To enable it, go to Basic Utilities (choice **7** on the Introductory Menu), run WGSDMOD (the Enable/Disable Modules utility), and move Debugging in baseline from the Expressly Disabled column to the Available column.

This module provides for a number of special debugging features. Some only take effect when running the debugging version of MAJORBBS.EXE. Others take effect under all circumstances.

Once you enable the module, you may notice various text flying by at the top of your Main Console when your server is up and running. That's one of the features of the debugging module. It can display execution addresses and phase transitions on the Main Console in case the server locks up or otherwise crashes.

Check out CNF level 95 once you've enabled the debugging module. All of its features can be turned on or off from there. The available options are briefly described in the option help messages. Two of the most notable are:

- the ability to debug memory allocations in order to detect memory under/overflow bugs; and
- the ability to alter assertion failures to be quietly ignored or logged to disk rather than catastro().

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.

- Design with vision and foresight
- Code with attention to detail
- 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 directly 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 GPs

Illegal memory accesses can happen in hundreds of unusual 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, causes an immediate GP.

B. Accessing off the front or end of an array

The high scrutiny mode available with the Debugging in baseline module vastly increases the likelihood of catching these errors with GPs.

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):

```
struct key *binary(word,tab,n)
char *word;
struct key tab[];
int n;
{
     int cond;
     struct key *low,*mid,*high;

     low=&tab[0];
     high=&tab[n-1];
     while (low <= high) {
          mid=low+(high-low)/2;
          if ((cond=strcmp(word,mid->keyword)) < 0) {
               high=mid-1;
          }
          else if (cond > 0) {
               low=mid+1;
          }
          else {
               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.          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 Major BBS Sysop wants maximum up-time for his server, he sets CNF option GPHDLR in Hardware Setup to **YES**:

GPHDLR	Continue operation after "GP" errors?	**YES**

This causes the server to attempt to recover from GP errors after reporting them in the Audit Trail (see *Sysop's Guide* 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 server 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 The Major BBS server 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 \WGSERV\GP.OUT.

Reading a GP.OUT Report

A GP is a software interrupt 13. 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:

```
THE MAJOR BBS GP @ 2d4f:031a EC 0000 (recorded 12/16/94 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

YOUR SYSTEM NAME HERE
Server Version: 1.00
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:\WGSERV\MAJORBBS.EXE
0717=C:\WGSERV\GALGSBL.DLL
073f=C:\WGSERV\BBSBTU.DLL
075f=C:\WGSERV\DOSCALLS
2a5f=C:\WGSERV\GALMSG.DLL
2cff=C:\WGSERV\GALRSY.DLL
2d3f=C:\WGSERV\GALTLC.DLL
2d6f=C:\WGSERV\GALTXV.DLL
2d97=C:\WGSERV\GALUIE.DLL
2dbf=C:\WGSERV\GALXIT.DLL
2de7=C:\WGSERV\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 _CLRXRF
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 (or CS:EIP depending on the CPU) 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 23). 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 shows information on the most recent server activity on a user's channel.

Server Version: 1.00	The Major BBS server software version
User 3	User number (usrnum, see page 37)
channel 00	Channel number (hexadecimal, page 37)
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 26)
substate 0	usrptr->substt
MSG:galtlc.mcv	the current .MCV file being read
/-1	the last message read from this file (-1=none)
Module "Teleconference"	From the usrptr->state code (5 in this example)
Input "T"	Most recent status-3 input line on the server

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 main 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 server. 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\bc4\lib\c0phdll.obj
2d4F   \WGSERV\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 data — 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 BPs 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 29), 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 call 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 BPs 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 \WGSERV\SRC
CTDLL -S MJRTLC
CD \WGSERV\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 \WGSERV\PHOBJ\MJRTLC.ASM and makes a listing file in \WGSERV\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                     ;        case 0:
532                     ;             tptr->flags¦=NOPAGE;
533                     ;
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 confirms 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

D

H

M

S

V

W

X

Y

Z