GALACTICOMM

Flash Protocol Programmer's Guide

by Tim Stryker

*   *   *   *   *

What is the Flash Protocol?

The Flash Protocol is designed for use in situations where multiple computer users are online to a central device (the "host" computer), and where the need exists for rapid ("real-time") interaction between the users. Examples of such situations are: multi-player real-time games, multi-user real-time "chat" systems, multi-user real-time teleconferencing.

The Flash Protocol is an implementation of something I call the key-by-key (KBK) process. Here is the concept:

> In any multi-user real-time interaction, there is some kind of shared underlying entity (the "model") which is the same for all participating users. User input affects this model in various ways, but the essential thing is that all users access the same model at any given point in time. The traditional approach to this issue has been to maintain the model in the host computer, and to use the raw input data coming into the host from the users (typically keystrokes on computer keyboards) to perform in the host the operations determining how the user input affects the model; the results of these operations are then transmitted to each user's computer or terminal, for display. In the KBK process, the model is maintained in each user's computer (so that there are as many "copies" of the model running as there are users), and the separate copies of the model are kept equivalent to one another by ensuring that each user's computer is informed of the raw input created by all participating users, in the exact order in which the input is received by the host.
>
> Several points may need clarification: (1) all users need not have the same information displayed, under either the traditional process or KBK. Any aspect of the situation that is not precisely the same among the users is by definition not part of the "model". In the KBK process, each user's computer displays that information which its own particular user is entitled to see, based on its own internal representation of the model. This may be all of it (with or without additional, non-model information), selective portions of it (which portions may usefully differ from user to user), or, in unusual situations, none of it. (2) In order for the application of user input to have the same effect on all user copies of the model, all user

computers must in some sense "start" with the same specific parameterizations of the model. The KBK process does not depend on the details as to how this is brought about -- it might be by simultaneous invocations of the model-maintaining programs at each user site, or by any of a variety of synchronizing mechanisms, which may include the ability to hold certain users in a kind of "holding pool" (a sub-model in its own right) until proper conditions for synchronization can be established. (3) The KBK process does not depend on all user input being broadcast by the host to the user machines, only that portion that affects the model. Similarly, it does not depend on only user input information being broadcast by the host to the user machines: other data, both model-related and not, may accompany the user input, and may or may not be the same for all users. The essential condition is this: any data which has an effect on the model must be broadcast the same, and in the same order, to all participating users.

The KBK process does not depend on the types of computers used, the method of interconnection, or the protocols used in the interconnection links. Its essential starting conditions are: a host computer, multiple user computers, a means of conveying information bidirectionally between them, and a means of synchronization that can be relied upon to bring their copies of a "model" into equivalence at appropriate points. The process itself comprises the transmission of raw or minimally predigested input data from the users to the host, and the "broadcast" from the host back to each participating user's machine of the input data, each piece of the input data being tagged as to which specific user's machine the input came from, such that all users' machines are able to update their internal representations of the model in an equivalent manner.

The advantages of this are (a) virtually unlimited display-update speed, (b) independence of game or application particulars from the connectivity mechanism (i.e. Sysops need not update any BBS software in order for users to be able to run new games), (c) elimination of the need for standards in output codes (i.e. high-resolution bit-mapped graphics multi-player games and all manner of sound effects, etc. become possible without any change or impact of any kind on the protocol), (d) the size of BBS memory does not need to constantly increase over time in order to support more and more games, and (e) a unique style of "copy protection" becomes possible which is trouble-free to legitimate users yet restrictive to "pirates".

The disadvantages of this are (a) separate user-side software must be developed for each hardware platform wishing to participate in a given game or other application, (b) the need for startup synchronization constrains game design, and (c) certain aspects of game "scoring", such as cumulative totals and top-ten listings, which are trivial to implement under the traditional architecture, become so problematic under KBK that we recommend not doing them at all for the time being.

Marketing Considerations

Of special interest to 3rd-party marketeers are the financial dynamics of the Flash game marketplace, in contrast to traditional 3rd-party add-on software. Rather than your prospects being Sysops, of which there are maybe a few hundred, total, running entertainment-oriented systems, your prospects become the users of those systems, of which there are hundreds of thousands. So, rather than writing a great game and selling maybe 40 or 50 copies to Sysops at $400 each, you can write a phenomenal game and sell maybe 4000 or 5000 copies at $25 each. And that's with today's numbers -- as the concept expands in popularity and the world becomes more modem-aware, million-dollar games are not far off.

You will find existing and prospective Sysops of The Major BBS Entertainment Edition very supportive of your efforts... there is a unique 4-way symbiosis here, between you, the Sysops, their users, and Galacticomm. The better your games are, the better the Sysops will like it, because people will be using their systems more and more in order to play the games. The Sysops don't have to do any extra work to make this happen, either: no costs for software licenses, no bothering with source code and so forth, and most importantly, no increase in memory requirements to run each new game that comes out. There is no limit on the number of Flash games that an Entertainment Edition BBS can support, since the body of each game runs in the users' computers, not in the BBS!

Galacticomm has a distinct motive for your success too, both because of the $1-a-copy royalties and because the more users your games attract to our Sysops, the more lines and modem hardware they will wish to install. It's a 4-way win-win-win-win situation.

We recommend that you offer both a "shareware" and a "non-shareware" version of your first few titles. Once this Flash technology is well established, people will be willing to buy game software sight unseen, on the basis of the reputation of its designers. But in the early going, you will have much higher sales if you offer a scaled-back or "demo" copy of each game (perhaps with a limited playing time, or number of plays) for free, with the full-fledged version available only after the user pays you. The Flash Protocol is able, in Entertainment Edition Release F and later, to distinguish between shareware and non-shareware copies of the same game, and to enforce the restriction that no more than one person may use a given non-shareware copy on one system at one time.

The Sources

Here is an overview of the files supplied:

MAKEFC	BAT	...makes Flash Chat V1.0 from scratch using Turbo C 2.0
FC	C	...Flash Chat mainline
FCNOISE	C	...Flash Chat C-level interrupt-driven sound effects
FLCOMM	C	...general purpose user-side Flash Protocol utilities
FLKICK	C	...general purpose "rtkick" utility for Flash games
GENUTL	C	...standard "BBS Tools"
DOSFACE	C	...standard "BBS Tools"
FCNOISE	H	...Flash Chat #defines for sound effects
FLCOMM	H	...general purpose user-side Flash Protocol globals
DOSFACE	H	...standard "BBS Tools"
FKCODE	H	...standard "BBS Tools"
PORTABLE	H	...standard "BBS Tools"
SERCOM	ASM	...general purpose interrupt-driven serial I/O manager
NOISES	ASM	...general purpose interrupt-driven sound effects driver
FLUTIL	ASM	...general purpose assembly-level utilities
MODEL	MAC	...general purpose Microsoft/Turbo C "large" model EQUs
SCN2ASM	EXE	...general purpose utility to make screens EXE-resident
FLSCNBGN	ASM	...general purpose screens-begin assembly source
FLSCNEND	ASM	...general purpose screens-end assembly source
MFCSCNS	BAT	...Flash Chat screen-table maker
FCHAT	SCN	...Flash Chat main screen image
FCHELP	SCN	...Flash Chat chat mode help screen image
FCHELPT	SCN	...Flash Chat terminal mode help screen image
SERCOM	OBJ	...result of assembling SERCOM.ASM
NOISES	OBJ	...result of assembling NOISES.ASM
FLUTIL	OBJ	...result of assembling FLUTIL.ASM
FCSCNS	OBJ	...result of running MFCSCNS.BAT
MBBST	LIB	...standard "BBS Tools"
LFC	BAT	...Flash Chat TLINK command
LFC	LNK	...Flash Chat TLINK control file
FC10	EXE	...Flash Chat V1.0 standalone EXE file

Flash Chat is supplied here primarily as an example of the use of the Flash Protocol, so that you can see how the various utilities fit together. It is kind of fun in its own right, though, and you may wish to "port" it to hardware/software platforms other than IBM-PC/MS-DOS. If you wish to port Flash Attack to other platforms, we are asking that you do Flash Chat first, as a demonstration of proficiency (and also because 95% of the work of porting Flash Attack will be finished, once you have ported Flash Chat!).

There is room for significant improvement of Flash Chat, though, as a straight chat facility: buffer "scrollback", private "whispers", a "log to disk" option, and so forth. Feel free to use this code as a starting point for enhancement in any direction you choose.

Initialization

The subroutine init() in FC.C demonstrates the 3 central actions necessary to overall system initialization:

inicom(argc,argv)	...sets up interrupt-driven serial communications using COM port from command line or user menu
ininoi()	...sets up interrupt-driven sound effects, and "fast ticker" auto-increment at 145.6 Hz
setspc(spckys)	...sets up special (non-printable) key codes for transparent pass-through by Flash Protocol

The first two are pretty straightforward to understand, but setspc() may take some explanation. The Flash Protocol is optimized for speed. You may configure up to 11 different keystroke codes such that both the code itself and the indication as to which user issued it are condensed into a single byte when broadcast by the host. You may also have up to another 19 key codes of keys which would normally take two bytes (the IBM "extended" keys, like cursor arrows and function keys) and encode them down to one data byte each (these then take 2 bytes when broadcast, rather than 3). Any CTRL characters you will want to handle, such as carriage-return (13) or backspace (8) will need to appear in this list as well.

You pass the key codes you want treated these special ways to setspc(). The argument to setspc() is expected to be the address of an integer array, the first eleven elements of which are the the "most common" keys (in Flash Attack, for example, these are the 8 tank-movement keys, the letters M and P, and function key F1), and the remainder of which, up to a total of 30 items, are any other non-printable ASCII or two-byte extended key codes you will be passing to ecoutp() later. (Recall that subroutine getchc() in MBBST.LIB returns two-byte key codes in integer format when extended keys are hit, as #defined in file FKCODE.H).

SOLO Operation

One of the options open to the user in inicom() is that of going SOLO, without a COM port. If this option is selected, then the global variable "solo" (declared external in FLCOMM.H) is set, and no COM port stuff is initialized. However, the error-corrected I/O routines ecoutp() and ecinp()

in FLCOMM.C sense the solo flag, to simulate the effect of data forwarded to the host being picked up upon broadcast.

Your program can of course do whatever it likes with the SOLO case. We recommend making it do little more than give the user a faint taste of what he or she will see when online... in other words, drawing a sharp distinction between a SOLO game and a one-player game played while online to a BBS. It is very desirable to set up a computer-animated opponent for the user to play against if alone, but if you enable this for SOLO operation, you have nothing more than a conventional single-user computer game on your hands, of which there are already thousands. By reserving the computer-animated opponent feature for single-player online operation, you create the basis for online competition: one user will SCAN another user in the game and join up, and pretty soon you have a group of players going, exchanging word-of-mouth advertising and building fun momentum.

Note that since there is no point to "terminal mode" in the SOLO case, your program will bypass the initial LINEUP, so the "names" array will not be filled in. You can just make an arbitrary assignment, as Flash Chat does, or prompt the user manually.

Terminal Mode

A key to the advent of the Flash concept was the realization that the game programmer does not have to write a sufficiently powerful terminal emulator/autodialer to convince users to use it instead of their own. Users can dial into the BBS using whatever package they are accustomed to, and then "shell out" to a Flash game. Naturally, if you feel inclined to write a sophisticated front-end to your Flash games that competes on an even footing with ProComm, Telix, and so forth, go right ahead. But this will tend to (a) increase the size of your EXE file, making people more reluctant to upload/download it, (b) make your code more difficult to port to other platforms, and (c) take up months of your programming time that could be more productively spent writing amazing new games rather than re-inventing the wheel.

A rough-and-ready form of terminal mode that can be used in both shelled-out and standalone scenarios appears in function termnl() in FC.C. The passed parameter is an introductory string or information message to be displayed to the user once the screen is in place. The first call to this routine clears the terminal screen; subsequent calls will pick up where the previous one left off (even if the screen has held something very different in the meantime).

The functions locate(), setatr(), printf(), rstloc(), and setwin() are all from MBBST.LIB and are documented in The Major BBS Programmer's Guide. Assembly source to these routines is not available.

Onset of Flash operation is triggered by the call to flashm() in case F9 of the switch in termnl(). Parameters passed to flashm() are the game name and version code, and the 32-bit "serial number" of the copy, or 0L (don't forget the 'L'!) if shareware. When the BBS digests the results of this, it sends 6 NULs followed by the initial LINEUP command, which gets parsed by the call to parsln(). If parsln returns 0, then some sort of attempt, innocent or otherwise, has been made to subvert the normal startup process, and we recommend treating this the same as a "NO INITIAL LINEUP" condition.

If the LINEUP is successful, then "npyrs" contains the number of people now in the current "Flash pool", "names" is an array of pointers to their User-IDs, and "rseed" is a number suitable for use as a seed for a common pseudo-random number generator (all of these globals are declared external, for your convenience, in FLCOMM.H). The idea behind "rseed" is that all machines in a pool must make the identical "pseudo-random" decisions in order for their models to remain identical. A sample pseudo-random routine is given here as the function rnd() in FC.C. If you have a game involving certain random elements, at the start of each game you could set rndnum=rseed and the game would proceed, seemingly randomly but totally "in sync" among all machines from that point forward.

The LINEUP, by the way, contains useful information that parsln() discards. If the pool is taking place in a private channel, and the User-ID of the owner of that channel is present in the LINEUP, then that User-ID is preceded by an asterisk. You can use this to give the owner of a channel special privileges (like, to knock obnoxious people out of the pool, or to assign people to teams, etc.). Also, the "rseed" value is the value of now() on the host system (see DOSFACE.C), so you can have time-of-day dependent activity in any game. A "night game" of Flash Golf, for example, might be quite different from one in the daytime!

Flash Mode

Once the various shenanigans toward the tail end of function termnl() in FC.C are complete, subsequent calls to ecinp() will return one of several things (as #defined in FLCOMM.H):

-1	...nothing happening
-TICK	...a tick of the 18 Hz master clock in the host has elapsed
-REMOVE	...one of the users in this Flash pool has gone away; call ecinp() again and subtract '0' to find out which one, and reassign the last guy to this number if it wasn't the last guy himself that got removed
-LINEUP	...another user just joined this Flash pool, call parsln() to update the "names" array and npyrs, and be careful not to assume that anything in our own machine matches that in the new user's machine from this point forward

except the LINEUP-derived info, or the results of actions that we specifically henceforth arrange to happen

-NODATA	...contact with the host has been lost
<other>	...a keystroke code, from player number "pyrn"

(-TICK, -REMOVE, and -LINEUP are discussed below at some length.)

As far as local keyboard activity is concerned, you can just hand over anything that comes in via getchc() to ecoutp() and forget about it. Your program shouldn't be processing these keystrokes before hearing them back from the host via broadcast (this rule is not absolute but you should think long and hard about potential implications before violating it).

To leave Flash mode, we recommend calling lvpool() twice. The BBS will take the user out of the pool upon receipt of 3 CTRL-X's, so you should send 6 just to be on the safe side (the excess will be ignored). If you wanted to be elegant about it, your program could wait to see its own REMOVE command fly past before moving on, but this has not proven to be necessary in practice.

The -TICK Command

The reason for -TICK (and the suggested prcrtk() call resulting from it) is that this way you can time events and arrange for delayed actions in a way that is completely congruent among users. If you did time delays based on a ticker local to each user's computer, then yes, they would all agree about the order of various activities <u>most</u> of the time, because their internal crystals are all very accurate and they wouldn't drift with respect to one another by more than a few percent per year. But when the whole future of a simulation hinges on whether Cinderella does or does not manage to hop out of the coach before it turns back into a pumpkin at the stroke of midnight, "most of the time" is not good enough, and we wish to be certain that all machines in the pool will be making the identical decisions as to the exact order and timing of events. Using rtkick() delay values in units of 18 Hz "ticks" and calling prcrtk() based on -TICK returns from ecinp() makes this possible.

The -REMOVE Command

-REMOVE is fairly straightforward. The only trickery here has to do with conservation of player numbers: we want the people in the pool to always have player numbers 0 to npyrs-1, so if a user is removed from the middle someplace, we reassign the guy at the end to the vacated number... the viability of this hinges on the BBS doing the identical thing at the identical time, which it does! (Note that if the removed user was already the highest-numbered user, no reassignment is necessary or desirable.)

The -LINEUP Command

The receipt of -LINEUP is potentially the most troublesome. In Flash Chat itself there is no problem, because the "model" is an amorphous, utterly forgiving thing that has no particular history to its essentials, so that a full chat window and an empty one can be treated as essentially equivalent. However, in a more normal game environment you will have to think through very carefully how you will handle new entries into the pool during a game in progress. Remember that in general the new entrant won't have any idea what the game starting conditions were, or what keystrokes were issued by the players to change things since that time. In particular, the new entrant won't even know, a priori, whether or not a game is "in progress"!

There are many ways to deal with this, and I'm not violently pleased with any of them. Rather than hand you a pre-packaged method, let me throw out a few thoughts and perhaps you can come up with a better way. One option is to make the assumption upon initial entry that a game is always "in progress", so that the end of each game is implicitly the beginning of another, each machine being expected to emit a special "game change" pseudo-keystroke code upon detecting the transition. A new entrant, then, would remain in a kind of limbo until receiving one or more of these pseudocodes, and the machines already playing would know not to accept input from the newcomer in the interval between receipt of the LINEUP command and the first receipt of a broadcast pseudocode. Another method, the one used in Flash Attack, has all machines "sound off", when the LINEUP is received, as to whether they are in the game or just in chat. A complicated process of accepting or ignoring input from the various machines then ensues, with more fun as arbitrary patterns of participants and non-participants getting REMOVEd and re-installed come into play.

There is a key consideration in all this that we have not talked about yet, and that is the fact that the Flash Protocol is asymmetrical. Data transmitted from the host to each user's machine is error-protected, but keystrokes and whatnot going the other direction are not! The tradeoff here is one of error protection vs. responsiveness: given that it is implicit in KBK that model-affecting actions be broadcast back from the host to each machine before they are processed, it is vital to make the round-trip of a user's keystrokes as snappy as possible. Error-protecting a byte with any margin for safety requires at least 3 bytes total, so if it were 3 bytes out and 3 bytes back, we're talking 6 byte-times minimum between hitting the key and seeing the result, which at 1200 bps is 1/20th of a second, already at the fringe of human perceptibility. Given the practicalities of actual fact, many scenarios yield longer turnaround time, so we would have serious perceived sluggishness if not for cutting the 6 byte-times to 4 by sending each byte from the user machines to the host unprotected. If it gets clobbered, hey, at least it will be broadcast back to all machines in the identical clobbered form, so all machines will stay in sync. And most of the time it simply won't get clobbered, or will get clobbered to a value that all machines ignore, etc.

The implication of Flash Protocol asymmetry in the handling of LINEUPs received while a game is in progress is that you must take into account that your "game-change" or "sound off" pseudocodes can get clobbered on their way out the door. A mechanism for monitoring the returning broadcast stream for one's own pseudocodes is implied, with re-emission of them if not detected on the rebound within a certain period of time. Like I say, I'm not exactly thrilled with it. Let me know if you come up with something better!

Game Duration

The requirement for initial synchronization between all users' copies of the "model" constrains game design. The most straightforward way of dealing with this is to keep games short. This way, everybody can easily get in sync at the beginning of each game, and if no game takes more than a few minutes to play, nobody that enters the pool in the middle of one has to wait very long.

Another approach, if you want to have longer games, is to schedule them at set times. Everybody in the pool at 8 P.M., say, gets to play Flash Murder-Mystery for the next two hours or until it's solved, whichever comes first. The next game starts at 10 P.M., like a movie. Those attempting to enter outside these guidelines are politely told to come back later (your game software can itself enforce this, without modification to the BBS, by just looking at the now() value in the LINEUP random-number seed and calling lvpool() etc. if the time does not satisfy your tests).

A third approach would be to actually transfer game status data from a machine with the game in progress into each newcomer, by way of the broadcast pipeline! You could set it up so that machine #0, whichever it happened to be, would have the responsibility of spitting out, through some kind of pseudo-keystroke code sequence, enough game-status data that a new entrant to the Flash pool could pick up playing a game in the middle. This is straightforward in principle but tricky to implement when the possibility of transmission errors (not to mention changes in the status over the course of the finite time necessary for transmission) is taken into account.

Sound Effects

By calling ininoi() at the beginning of your program, and finnoi() at the end, you are able to take advantage of the technique shown in FCNOISE.C to generate interrupt-driven foreground and background sound effects.

The essential concept is that you arrange for your C language routine to be invoked every 1/145.6th of a second, regardless of what else may be going on in the machine at the time. Your C language routine then computes an integer "period" for the speaker oscillator and returns that as its value. A value of 100 is almost inaudibly high-pitched, a value of 2000 is mid-range, and a value above 50,000 or so degenerates into a series of clicks. Very smooth "glissandos" and whatnot can be achieved by changing the values you return by small increments each 1/145.6th of a second.

To the routine noise() in FCNOISE.C, sound codes below 100 are background codes. This means that they go on for a substantial interval but may be interrupted briefly for foreground sounds. Foreground sounds, when finished, return to the background sound they interrupted; background sounds, when done, terminate in silence. Of course they need not terminate at all, like the base CONDITION RED siren in Flash Attack which goes on and on until overridden by an explicit subroutine call with the background code for silence.

X.25 Considerations

Our position at present is that the Flash Protocol is not supported over X.25 packet-switch networks. The main reason for this is that delays of up to several seconds are not uncommon on these networks, and this turns the whole responsiveness issue discussed above on its head. The way in which ecinp() decides to report back the -NODATA code is by way of the "fstick" test at the end of prcrin() in FLCOMM.C. "fstick" is cleared each time rdser() returns an actual character, so when it builds up to a value of 150 it means that more than a whole second has gone by without the receipt of a single byte from the COM port. This happens very often when trying to play Flash Anything over X.25.

Feel free to experiment with larger numbers than 150 as a NODATA cutoff point, but be aware that frequent delays of more than a second in the echoing of a user's actions to his or her screen will not be well-received by the public. A thought: Flash Chat is unique in that the "model" does not include the exact contents of people's chat windows, so you could process the local user's keystrokes "on the way out", and specifically refrain from doing so upon receipt of their re-broadcast, meanwhile treating <u>other</u> users' keystrokes as usual! Then you could stretch out that <u>150 as far</u> as you like and the only drawbacks would be (a) delay in recognizing legitimate loss of contact, and (b) a certain "choppiness" in other people's apparent typing habits.

Limitations

The Flash Protocol is limited to 6 users per pool for shareware editions, and 10 users per pool for non-shareware. The number 6 is arbitrary but seems useful for practical purposes. Note that as supplied, Flash Chat can only exist as shareware; in order to have a non-zero serial number you would have to deal with the possibility of ten users in the pool somehow (there is no room for another chat window on the screen, as it stands).

You may impose a lesser maximum if you like, by having your program look at "npyrs" after its first LINEUP. If this number is higher than you can accept, your program could voluntarily remove itself by calling lvpool() a couple of times, then termnl(). The code dealing with the case of a user already in the pool, and seeing the overflowed LINEUP go by would simply ignore it in the knowledge that the excess user was about to REMOVE itself. Your message string passed as the argument to termnl() could explain to the user why his attempt at entry was being denied.

Note also that, except for the 30 specials passed to setspc(), only 7-bit ASCII data is permitted. This is because the parity bit is used in a special way in the low levels of the Flash Protocol. If you vitally need full 8-bit data for some reason, you can put front-ends on ecinp() and ecoutp() to "escape" the high bit, but most of the time (such as for foreign language support) the capacity for up to 30 special key codes will suffice.

Anti-Piracy

The way the duplicate-detection feature works is that the Flash game identifies its own serial number (if any) to the BBS as part of its attempt to enter Flash mode. The BBS then looks at the number and asks itself, "Is there anybody else on this system in Flash Mode using this same game right now with this same serial number?" If there is, it denies the user entry.

To combat the possibility of people manually entering the Flash mode entry sequence with a different serial number than the one they actually have (or any of a variety of other modes of subterfuge), the BBS includes the serial number of the copy it thinks it's talking to in the initial LINEUP back to that copy, and the copy itself checks to make sure that they match.

This takes care of most forms of casual piracy, but be aware that if people disassemble your EXE file they may be able to figure a way around this. A first-level measure of protection against this is to embed several copies of the serial number in several spots in the EXE file, each of them encrypted using a different method (e.g. XORing with certain data, rotating bits and then XORing, interspersing junk bits with the real ones, etc.). Your code can then check all of the copies against each other at run time before proceeding.

The most difficult patch-pirate to guard against is the one that goes after the code that does the checking or transmitting, and patches it so that the check automatically passes or the shareware cue gets transmitted every time even though the serial number is nonzero. You can have your code verify a CRC over its whole executable self to guard against this, but the patch-pirate may be able to locate the code that does <u>this</u>, and disable it. The only protection against this guy is to write the code in some terminally-obscure way so that he can't follow it, or to rely on legal remedies.

Since writing obscure code harms only you in the long run, it is vital to retain your ultimate legal protection, the Copyright to the code. Be sure that every copy of your program leaving your premises has your Copyright notice on it, in the form:

Copyright ‹year› ‹author or sponsor›

You can use the abbreviation "Copr." instead of "Copyright", but do not use "(C)", except as a supplement. A letter C between parentheses has no legal standing. (A letter C inside a circle does have legal standing as a valid substitute for the word "Copyright" but most computer character sets do not include one of these.) If you leave out either the year or the name of the Copyright owner, it also has no legal standing. (There is supposedly an international intellectual property law to which the U.S. has just become party, which eliminates the absolute requirement for these notices, but the issues have not yet been fully tested in court so why take any chances?)

Areas to Explore

There are any number of fantastic single-player games on the market that could be made into phenomenal multi-player games with almost no effort. Any game that pits the user against a computer-controlled entity or entities with the same "powers" as the player, is fair game. In the bowels of any such game there is probably a "move generator" construct, which takes input from the player's keyboard when it is the player's "turn", and input from an algorithm when it is the computer-controlled entity's "turn" (even in real-time games, players do take turns, it's just that the turns are very quick and most of the time they consist of "pass"). By modifying this to take everybody's input from the broadcast receive-buffer, and to then update its own display based on what its own user is supposed to see, a game like this can be made multi-user almost trivially. Any existing computer games company may be able to transform its product line virtually overnight into a panoply of sensational bit-mapped graphic, multi-player-by-modem games.

The biggest possibilities for innovation, though, are in the areas of game design which have multiple players in mind from the word go. Consider asymmetric games, in which people with different kinds of roles access the "model" from very different points of view (e.g. air-traffic controller and pilots, submarines and surface ships, navigator/pilot/gunner teams, etc.). 3-D perspective-view chase or maze or racing games, high-res animated gambling halls, and of course, D&D-style multi-player graphic adventures are a few more of the possibilities that come to mind.

As communications technology evolves, higher speed modems will allow more voluminous input: either more people per Flash pool or greater input volume per user... joysticks today, "power gloves" tomorrow? As the quest for Virtual Reality unfolds, the abundance of opportunity is staggering.