The Gory Details of Cassette Storage

Peter Boyle
1337 Adams
Denver CO 80206

The $35 audio cassette recorder has become the standard medium for program storage on small microprocessor systems. Initially, using audio recorders to store programs was an afterthought. Many systems did not have any tape interface at all and those that did used it in the most trivial way, usually just copying memory out to the tape byte for byte. It was enough that the scheme worked at all. This attitude is changing rapidly. The audio cassette recorder has become a member in good standing of the class of mass storage devices, along with digital tapes and floppy disks.

By using higher speeds, controlling the stop-start switch and using multiple drives one can use audio cassette recorders for many of the tasks normally performed by more expensive devices. All that is required is that the subject of cassette software be taken seriously. This is what I propose to do.

Specifically, the topic here will be restricted to the subject of saving the contents of areas of memory on cassette tape, and later copying the tape back into memory.

General cassette tape usage for other types of data is a subject which needs development, but that will have to be covered another time.

Terminology

For convenience, here are a few terms. I will call the program to handle cassette tape the *cassette handler*, even though this program is not a *device handler* in the strict sense. The handler has two parts. The part which writes out the memory content is the *saver* and the part which reads this data back in is the *loader*. The collection of data on the tape which the saver creates is one type of cassette tape *file*. A file of this type would be called a *save* or *memory image* file.

In some systems, where audio tape was originally conceived as the basic mass storage medium, there may exist a special form of the loader, called a *bootstrap loader*. The bootstrap loader is a minimal version of the full loader which is used to read in the entire handler. Since the form of a bootstrap loader should be determined by the details of the complete handler (rather than the other way around), the specifications of the bootstrap loader are best left until the handler details are clearly defined.

Most microcomputers have some form of *monitor* program. This is a program which is resident (in memory) at all times. It provides the basic user interface. Usually the cassette tape handler is part of this monitor program. Naturally, the monitor program, or perhaps only the bootstrap loader part of it, can be in read-only memory (ROM), and hence immediately available when a system is turned on.

The Problem

The cassette handling routines that come with commercial systems tend to be rather trivial. They are usually written for a very specific saving and loading task, with apparently no thought at all given to versatility.

In order to develop some thoughts on better cassette read-write software, I will review some of the software I've seen and then go into the alternatives. After that there will be some comments on the implementation details of a more correct set of cassette read and write routines.

Since I own a KIM-1 which is based on the MOS Technology 6502, I will draw some examples from the 6502. However the discussion will apply equally to other microcomputer systems.

The Digital Group System

In many ways, particularly in some hardware aspects, the Digital Group systems are a real pleasure. Their 6502 monitor program, however, is an excellent example of the limitations typical of cassette handling software. Their tape-read routine is on a ROM and it assumes that the file it loads will go into memory starting at location H0 (I will use a preceding "H" to differentiate numbers that are written in hexadecimal). The program loader further assumes that the program it loads will start at H500. This means that the ROM routine cannot be easily used to load some other part of memory. What is worse, programs have to be written with a hole at H500 so that a jump to the actual starting address can be placed there. That's not all. The timing is software loop controlled and these loops are in a 1702A PROM which runs at less than full speed. Thus this timing is difficult to duplicate in any other system or in normal full speed memory.

There are three lessons to learn from the limitations of the Digital Group monitor. First, cassette software should never assume that the bytes on tape correspond to some predetermined locations in memory. The location and number of bytes in the load should be part of the data that is saved on the tape. Second, the starting address of the loaded program should be saved on the tape. Third, whenever possible, external timing standards should be used.

The KIM-1

The KIM-1 is another interesting system. The KIM at least allows saving and restoring any area of memory, and it uses an external timer for timing, but it does not provide any self-starting mechanism at the end

In the following article Peter Boyle offers some criticisms of the cassette software supplied by a couple of the more popular manufacturers . . . and also puts forth some worthwhile ideas on how this software should be written. If you're not into developing cassette software at the moment, let me suggest you read the article and pick up on some of his techniques in memory management and software memory protection. – John.

of a load. This means that the starting address must be *stored* elsewhere, such as on a handy match book cover. Naturally in the course of events the files and their starting addresses tend to get separated.

Although the KIM cassette handler is a little slower than I would prefer, the cassette interface hardware is excellent. It is capable of 1200 baud (bits per second) rates, but the KIM ROM runs it at 133 baud. As soon as some additional memory was added to the KIM, I found the slow speed intolerable.

There is a more important limitation in the KIM software, which is shared by most of the cassette handlers I have seen. The memory copied out to the tape in a single write must be one continuous block and correspond to sequential memory addresses. Thus the entire write is specified by only two addresses, the upper and lower limits of the area of memory to be written out. That means that in order to write out two or more sections of memory which are in different places it is necessary to either write multiple files or copy out the in-between sections as well.

For an example of why this is a severe limitation, consider a typical KIM program. It will consist of several sections which are widely separated in the address space. It starts at, say, H200 in order to begin above the stack. It uses H0 to HFF (page 0) to store constants and commonly used variables and it requires that the interrupt vectors at H17XX be correctly set up. Between page H0 and page H17 there is 6K of memory space. To write all that as a single file requires writing all the in-between pages, most of which are not even implemented memory on the KIM.

The primary lesson from the KIM monitor is that, for one reason or another, computer programs tend not to be contiguously arranged in the address space. Clearly, cassette software should not assume that they are contiguous.

Features of Cassette Handlers

Save Time Control

So far I have discussed the things that should be specified at the time the cassette tape file is written, and hence what sort of information besides the data bytes themselves should be written onto the tape. Basically, these were the starting address of the program and the load address information. In particular, we saw that it would be nice if this load address information allowed a tape file to load any word or words in memory, independent of the addresses of these words, and even if the addresses did not follow in numerical sequence.

There are a few other items that a good cassette handler would include in file, things like the name of the file, the type of file, and perhaps even the date on which it was created.

Load Time Control

At the other end of the process – when the file is read back in – it might be nice to be able to exercise some control. It's fairly obvious that we may need to override the original load specifications. Thus the loading program should provide a mechanism for loading into a different place in memory and for starting at an address other than that saved with the program. The option of loading into a location which is specified at load time is particularly important in simple systems where a cassette handler might be used to store data other than programs. This is because data other than programs can usually be relocated to another place in memory and still be meaningful. Programs, on the other hand, usually contain references to absolute addresses that would be meaningless if the program were moved to some other part of memory.

It would also be nice if, while reading, the read program scrupulously avoided modifying any words other than those it actually read in. If this precaution is taken, then one file may overlay the current memory content, essentially adding to it. This can be useful, for example, in loading a debug routine which is only required with other programs during development.

Another feature a loader should have is the ability to load only a part of the entire file – for example, only those routines that load into memory addresses from H2000 to H2200 from a program file that could load from H2000 to H2FFF. This feature is very handy for stealing subroutines written into one program for use in another.

Commands

At this point we have listed some of the desirable features of a cassette read and write program for program storage. Before considering the implementation details it would be wise to imagine the actual operation of such a system. Imagining the actual use of a program often shows up problems that more general thoughts about it do not. This is also a good time to introduce a possible set of commands for controlling the handler.

Let's assume that a device with Teletype-like capability is available, since that is an easy to understand reference point. Let's also assume that the cassette handler is on ROM so that the details of bootstrapping a particular machine can be ignored.

Loader Commands

So now we turn on the power. The cassette handler starts automatically. We set up the tape with the program we want to load, start the tape and type "R" for "read." The loader program reads the new program into the same locations we wrote it from and starts it up. The read command needs no further specifications since the load addresses and the starting address were read from the tape file itself.

The $35 audio cassette recorder has become the standard medium for program storage on small microprocessor systems.

Fine. Suppose we only wanted part of it? In that case, we could precede the "R" command with a command specifying the highest and lowest addresses that the loader can load into. This could be "Lxxxx,yyyy" (where xxxx and yyyy are hex addresses). The loader would then ignore any data from the tape that would have loaded into an address higher than the upper limit or lower than the lower limit.

Suppose we did not want to start it after loading; for example, if we had loaded only part of a program? To handle that case we could have a separate read command that did not start after the load, but which was in other respects the same as "R", perhaps "A" for *append*.

To provide the ability to load into a different location we could have a "Bxxxx" command which sets up a *base* address for the load. This base address might function as follows. If a load address from the tape falls within the limits set up by the "L" command, and the "B" command was used, then the base address is used instead of the address given on the tape. Bytes from the tape that normally would not be loaded into an address between the limits would be ignored.

So much for the loader part of the handler. The scenario of its use seems clear and simple. Now let's turn to the saver part of the handler.

Saver Commands

Suppose that we have a program in memory that we wish to save. It would be nice to have a command analagous to the "R" command, say "W", which simply wrote out the program in memory. Obviously, however, we do have to specify what parts of memory constitute the program and hence what needs to be written out. Ignoring for now the problem of internal representation of this information, what is the easiest way to specify it? Handled carelessly, this could be a cumbersome detail. A program can consist of bits and pieces from all over the address space. Specifying each piece to be written every time a write is made could become a nuisance.

There are some ways to avoid this nuisance. Note that the contents of memory most probably came from one or more loads from other tape files. We could preserve the information specifying which locations in memory were loaded. In the default case when only the "W" command is given we can write back out exactly those bytes we read previously.

The information saved during a load can be thought of as specifying which locations in memory are of interest. Perhaps we should call it the *memory control data* (MCD), and use it as the primary means of specifying specific areas of memory.

An MCD which was automatically updated by multiple loads would take care of copy operations, slight modifications of programs, and combining two or more parts of programs into one. This would cover most situations, but not all. There is still a need for a way to specify a write which is not just a combination of previous reads. This could be implemented by a command which modifies or extends the preserved information, specifying which bytes were loaded.

We obviously must have an "M" command, say "Mxxxx,yyyy" which would *mark* the memory between address xxxx and yyyy as important. Associated with this we will need a way to reset the MCD to a null value. Perhaps "K" for *kill*.

So far there are seven commands. These are summarized as the first seven in Fig. 1. Naturally, the computer for which the cassette handler is to be written will probably already have a system monitor program. If so, the actual syntax of the commands will probably be chosen to be compatible with the existing software.

Implementation

So, to write a cassette tape handler we must write three subprograms: the loader, which executes the "R" command; the saver, which executes the "W" command; and the command decoder which interacts with the outside world via the Teletype, accepts commands, and executes them. The loader may be split into two parts, the bootstrap and the frills.

A bootstrap for a system

R	—	Read a file into memory set bit map and start it.
W	—	Write the file specified by the bit map.
A	—	Append a file. Same as R, except do not start it.
K	—	Kill (reset) the bit map.
Bxxxx	—	Force base address for the load to Hxxxx.
Lxxxx,yyyy	—	Set upper and lower load limits.
Mxxxx,yyyy	—	Mark the pages between Hxxxx and Hyyyy on the bit map.
Pxxxx,yyyy	—	Protect the pages between Hxxxx and Hyyyy.
Oxxxx,yyyy	—	Open (unprotect) the pages between Hxxxx and Hyyyy.

Fig. 1. The keyboard commands that could be used to control the handler described in the text.

with this type of cassette handler would consist of the basic loader subroutine and a simple routine to call the loader and then jump to the starting address of the program loaded. Probably the bootstrap would not contain any of the optional control functions, but would allow for them after the entire loader had been input.

The command decoder is quite straightforward, so we need not go into it here. The other parts will depend upon the representations, both on the tape and in memory, that we choose for the loading data. That is what we will consider next.

Memory Segments

How can a thing like the MCD be implemented? The obvious approach would be to keep a table of memory *segments*. Each table entry would consist of two bytes for the lower address of the segment and two bytes for the upper address. This scheme has two problems. The first is that the "M" command described above is messy to implement with a segment table arrangement. This is because a new addition to the MCD would have to be checked against the existing table entries and overlaps or duplications resolved. The second problem is purely aesthetic. As my friend Paul is wont to say, "There are only three nice numbers in computing, none, one, and as many as you like." The number of memory segments allowed would be none of these, because the space we allocated for the memory segment table would be limited. However large we made it, it would still be possible to have too many memory segments. Imagine the annoyance that a "too many memory segments" error would cause!

Bit Maps

If not a memory segment table, what then? Here is a solution that I like.

Limit the memory segments that can be written to multiples of 256 bytes (1 page). For each possible page, assign a bit as a flag which, when on, indicates that the corresponding page of memory should be written out when a "W" command is given. Since there are at most 256 possible pages we need only 32 bytes to store these 256 bits. This block of 32 words I will call the *bit map* of the current memory usage.

The bit map scheme allows any conceivable segmentation arrangement and is easy to update to reflect multiple loads or added segments as specified by the "M" command. All we have to do is set those bits which correspond to the pages which are affected.

This arrangement for the MCD is similar to one which is used in the PDP/8 operating system. They call it a *core control block.*

The section of the code of the handler which deals with the bit map requires a little thought. One way to proceed is to write a subroutine which takes a memory page number in a register, say the accumulator, and returns two things: The number of the word in the bit map which contains the corresponding bit, and a mask word with the appropriate bit set. Fig. 2 is a flowchart of a routine which performs this task. It returns the number of the word in X and the bit mask in the accumulator. Now in order to mark a page in the bit map we call the subroutine and "OR" the mask with the appropriate bit map word. To test to see if a particular page is marked we call the subroutine and "AND" the mask with the bit map word. If the "AND" gives a nonzero result, then the page was marked.

Memory Protection

In the actual implementation of a cassette handler there is a programming difficulty that will have to be dealt with. The problem is this: There are some words of memory which the cassette loader must not be allowed to overwrite. In machines where I/O is implemented as memory addresses, the most obvious locations we must protect are the I/O ports. Since the loader is reading in from the cassette port it obviously must not write into that port. The other I/O ports probably should not be overwritten either.

In any case, the entire bit map must not be overwritten because the load time bit map will be different from the write time bit map whenever the current load is an overlay to earlier loads. If the bit map were overwritten, it would be changed to the value that existed at the time of the write, and this could be incorrect.

Besides areas of memory which must not be read by the loader there are other areas which need never be saved or reread. Memory addresses corresponding to nonexistent memory, read-only memory, and the stack area are examples.

For simplicity the entire problem could just be ignored. The user would then be expected to know what was going on and never mark certain areas in the MCD bit map. This is not a very nice solution since it requires knowledge of exactly where in memory the bit map and other special things are kept.

It would be much nicer to take care of this in the handler code itself. One way to do this would be to introduce another bit map. This map would keep track of those pages of memory that are to be *protected,* that is those pages that the user wishes to automatically exclude from any read or write.

Of course, there will have to be some commands to manipulate the protection bit map, since this feature is quite general and would have applications other than the obvious one of preventing errors produced by overwriting certain memory addresses. If the protection bit map reflects the currently implemented memory, it could be useful in a variety of other programs for run-time storage allocation. By comparing the protection bit map and the MCD bit map, a program could even determine which areas of memory are currently unused. The protection bit map could also be used to specify complex partial reads of files. The possibilities seem endless.

In keeping with the other commands we could have a "P" command similar to the "M" command suggested earlier. "Pxxxx,yyyy" would protect the pages between address Hxxxx and Hyyyy. Similarly "Oxxxx,yyyy" would *open,* that is unprotect, the pages between Hxxxx and Hyyyy.

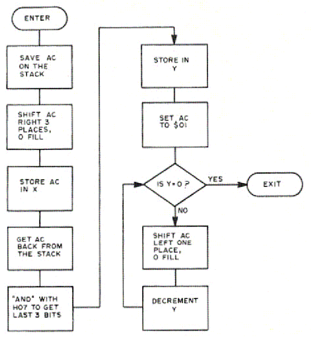
ENTER
SAVE AC ON THE STACK
SHIFT AC RIGHT 3 PLACES, 0 FILL
STORE AC IN X
GET AC BACK FROM THE STACK
"AND" WITH H07 TO GET LAST 3 BITS
STORE IN Y
SET AC TO $01
IS Y=0 ?
YES
EXIT
NO
SHIFT AC LEFT ONE PLACE, 0 FILL
DECREMENT Y

Fig. 2. This routine takes a memory page number in the accumulator and returns the corresponding bit map offset in X and the bit mask in the accumulator.

Different systems and even different programs on the same system may need to protect different areas of memory. Since it is inconvenient to reset the entire protection map, it should be possible to load it from an existing file. Here is a potential problem to avoid. In general the MCD bit map will be protected while the protection bit map itself will not be protected. Since the protection scheme described so far works only on whole pages, it follows that the MCD bit map and the protection bit map will have to be stored in different pages of memory.

File Structure

At this point I would like to briefly touch on the question of the format of the file that the saver creates. The arrangement of the MCD in memory as a 32-byte block does not mean that the way to format the actual tape file is as a 32-byte MCD followed by the corresponding pages of data bytes. I made this mistake in my first version, so I know!

These are the reasons why this is a bad idea. First, it takes a bit of computing time to unpack a bit from the MCD. If the blocks on the tape are directly contiguous, there may not be enough time during the stop bit of the last character of a 256-byte page to set up for the next page. The second reason it is a bad idea is that it is an unnatural (if not impossible) way for an assembler or compiler program to format its output. The bit map scheme cannot be used unless you know, before you start the write, which pages you are going to write. Since you don't always have this information, a system standard cassette file format should not require that you do.

The format to use instead is something of an open question, and there is much to be said on the subject. Perhaps the simplest format is one used by most paper tape loaders: some number of records where each record consists of an address to begin the load at, a byte count, the actual bytes, and then a check byte or word. This format is not all encompassing, but, whatever the detail of the format, the MCD should be decoded into the actual page addresses and these actual addresses written onto the tape instead of the bit map itself. ■