HOW TO HACK on the ZX Spectrum

A complete guide to creating POKEs on the Spectrum, featuring full examples. Devised and written by Richard P Swann

CONTENTS

WHAT YOU WILL NEED

If you want to use this book successfully, you will need the following:

* An understanding of Spectrum BASIC. If you are total beginner, you will almost certainly find this book too complicated, and hence useless to you. If you can understand most of the Spectrum manual, then you should be all right. If you don't understand a word of the section on using +3 DOS in machine code in the +3 manual, don't worry, 'cause I don't either!

* A disassembler or a monitor program. You can, in theory, hack anything without one of these, but in practice, it would be impossible for a beginner, and next to impossible for someone experienced. My personal choice is Hisoft's DEVPAC, but if you have the programs STK and 007 Disassembler (which were on the covertapes of YS #75 and #77), that'll do fine for this book.

* Some games to hack. Obviously business software is out of the window because it's illegal to hack it, and you don't need to anyway. Strategy and adventure games are pretty unlikely to demand a POKE (although it's possible, and I'll show you how I hacked an adventure later on), so you're left with arcade and arcade adventure games. Most of the time you'll want to find infinite lives and/or energy, but there are always exceptions. This book concentrates on games that you're likely to have, which are games which have been on magazine covertapes, or best-sellers.

* Patience and determination. This is the most important thing of all. Finding POKEs is challenging without any knowledge of how the program works, or even a knowledge of the language it's written in! So be prepared to do some serious THINKING!

* A Multiface or other "magic box" device. This is not essential, but it is extremely useful, and a few sections in this book require you to have one (but as few as possible, I might point out!)

PART ONE - The Basic Idea

Before we get down to some serious hacking, it will be a good idea to know what we are looking for. Okay, I know that some of you may think that this is pointless, but if you don't know what you're aiming at, you'll never get anywhere. Basically, we are going to examine a program, and change it so that certain instructions are altered so that the game becomes more easier to the user. This may be in the form of infinite lives, infinite energy, immunity, infinite bullets etc. The only way we can hope to do this is to understand what is going on in the game. And because most games are written in machine code, we're going to have to understand that as well. Please don't be put off by the thought of learning a new language; in fact you need know very little machine code knowledge to hack most games.

To start with, we'll just look at infinite lives. Lives are normally a small number, between 3 and 9 (although some games may have more), and the common thing with all games that have lives is that somewhere in the game, the amount of lives are set as a definite number, and somewhere else, the amount of lives goes down by one. In order to get infinite lives, we would have to remove the command in the program which decreased a player's lives.
You probably have and idea about how a lives system would work in BASIC. Say if, for argument's sake, a BASIC game had three lives, you would expect to see something like...

```
100 LET LIVES=3
```

...contained in the program. Then, a bit further on, you'd expect to see...

```
500 LET LIVES=LIVES-1
```

In order to get infinite lives, we would simply remove line 500 altogether and RUN the program.

A similar idea appears in machine code, but the way it's done is slightly different. For starters, machine code doesn't have any variables! You might therefore wonder how on earth the computer can store anything. In actual fact, the computer can store information anywhere in RAM, as you may well know, and this is exactly what happens in machine code.

In Chapter 24 ("The Memory") of the Spectrum manual, there's a detailed description of the Spectrum's memory. The best way to visualise the memory, I think, is to imagine 65,536 boxes, each one containing a piece of paper with a number from 0 to 255 written on it. Therefore, it is no problem for a computer to store the number of lives in machine code, since it can just put it as a byte in a memory location, leave it there, and come back to it later. You should be aware that in machine code, commands are also stored in memory locations as bytes; so if you get commands and data mixed up in memory, the computer could easily try and execute the data, thinking it's a command, and trying to execute it. Unlike BASIC, there are no errors in machine code, and the computer can execute anything it finds, so in this case you will get a crash. So most programs keep program data and program commands separately. Anyway, in order to store three lives in machine code, we'd put the number 3 in a memory location. Unfortunately, we can't do this straight away in machine code, and we can only do it by using what are called registers. Registers store information in the same way as memory locations. They are bit more versatile though, as you can perform
calculations with them. All the same, they are not the same as variables in BASIC, and are more like a pair of hands used for counting.

The main register we'll look at for now is the A register (sometimes called the accumulator), which can store a single byte and have sums done to it. What we would do to store the number three in a memory location is to put the number 3 into the A register, then put the contents of the A register (which are of course, 3) into a memory location.

The actual way of writing this in machine code is:

```
LD A,3
LD (#8000),A
```

Actually, strictly speaking, the above isn't machine code at all! Machine code as the computer sees it is, as I explained earlier, consists of many bytes in memory, which are pretty meaningless to humans. So some people invented assembly language (which is what the above is), where each instruction carried out by the computer is given a name (called a mnemonic). The above program features one command used in two different ways. The command is LD, which is pronounced "load". This has nothing at all to do with loading a program from tape. It basically is a transfer of information from one place to another.

The comma in the instructions is read as "with", so the whole instruction is read as "Load A with 3". It now seems obvious that this instruction is putting the number 3 into the A register. The second command is also LD, but the way its used is slightly different. The brackets mean "the contents of", so the whole instruction is read as "load the contents of #8000 with A". (Think of a bracket as a byte in memory, where everything in the bracket is part of that byte) Therefore, this instruction would make the computer take whatever value is in the A register, and store it in memory location #8000 (of course, it could be any other memory location as long as it is unused). So the overall result of the two commands (normally called operations) would be to put the number 3 in memory location #8000.

Congratulations! You've just learned the first way to hack. Clearly, if in a real program, we found these operations, we could change the LD A,3 to something like LD A,100 to get 100 lives!

Before we can do any "real" hacking, I'd better discuss how "real" machine code is written. Every machine code instruction contains an opcode, and some instructions need an operand. An opcode is simply the instruction the computer is going to do. Every number from 0 to #FF corresponds with a specific opcode. You can find a complete list of opcodes in Appendix A of your Spectrum manual.

The operand is used whenever there is ambiguity over something. If you look in Appendix A of the Spectrum manual, you will see that the opcode #3E is "LD A,n". "n" in this case can mean any single byte number i.e.: a number from 0 to 255. But it's possible to put any number within this range e.g.: LD A,3 or LD A,#40 or LD A,#80 etc. The computer has to somehow know what data it is dealing with, and this is where the operand comes in. In machine code, whenever the computer comes across the opcode #3E, it looks at the byte after the opcode, and assumes it's the data needed. So, if the computer came across the bytes #3E and #40 in succession, it would put the value #40 in the A register. In this example, #3E is the opcode, while #40 is the operand. After executing the instruction, the computer goes to the byte after the operand, and starts running code from there. In the second instruction of our example, the opcode is #32, i.e.:

"LD (NN),A".

The ambiguity is in the address where we are going to store the value of the A register. In this case, the operand takes up two bytes, hence the "NN", which again comes after the opcode. You should note that this is an address in memory, and is always referred to as two bytes. So you might expect the machine code equivalent of LD (#8000),A to be #32 #80 #00. Except it isn't! For some odd reason, in machine code, all two byte operands are written the wrong way round, so the actual machine code equivalent of LD (#8000),A is #32 #00 #80. There is no hard and fast reason why, it's just vitally important that you remember this.

In short, the program of the previous page is written as...

```
#3E #03 #32 #00 #80
```

...which takes up five bytes.

Not all instructions require operands. For example, DEC A (short for "decrease A", which subtracts one from the A register) has no ambiguity at all. There is only one way to decrease the value in the A register, so the instruction only takes up one byte, in this case #3D. No operands are needed.

Right, time for your first bit of hacking! From what we've discussed above, we want to find, somewhere in the game's code, a set of instructions which put the number of lives into a byte in memory. So if the game had three lives, we could expect to see the bytes...

#3E #03 #32

...unfortunately, we don't know where in memory the number of lives is going to go, so we can't work out the operands for the second instruction (#32). But in fact, we don't need to, for if we find the above sequence of bytes in a program, we can simply examine the two bytes after this sequence to find out where in memory the number is being put.

So let's put this into practice and hack an actual game. For convenience I've chosen Sweevo's World, which featured on a YS Covertape Issue 60 December 1990. I would have chosen a more recent game, but they all have some form of protection on them. Besides, obtaining infinite lives is relatively easy. Getting the game in memory without it running is easy. If you've typed out a fair few POKEs in your time, this will be no problem. Just MERGE the BASIC loader and put a STOP statement before the RANDOMIZE USR statement. Then RUN the program and wait for it to load, until the OK message appears. Now we have the game in memory, we can load STK and examine it. You have to be a bit careful here, because STK occupies 6K of the Spectrum's memory. Although it can be anywhere in memory, it possible to overwrite the all-important lives code in the game with STK, so we have to be careful. The best places to put STK are in the graphics, map or sound data. There's no easy way to tell where this is, so you'll have to try pot luck. But it helps if you know where the game loads to, so load up STK at any address, and press J and then Caps+9 to read in a header a couple of times. Then read the headers of the three blocks of code after the BASIC. They are:

Bytes: S 4000,1B00
Bytes: M FB90,043D
Bytes: P 60E0,82B0

The first block is the loading screen. The second block is in actual fact the game's music (but you wouldn't be expected to know this), and the third block is the actual game itself. Therefore, we can put STK anywhere above (60E0+82B0)=E390. For argument's sake, let's put it at EA60, which is 60000 decimal. When we've finished hacking, we can reload "Bytes: M" and run the program. Load up STK (after having stopped the BASIC and returned with a STOP statement) at address 60000.

Now at last, some hacking. Bearing in mind that you start the game with five lives. Press Q to search for a byte. Enter the address we want to start searching at as 60E0, because that's the start of the game. Now type in the following:

#3E #05 #32

These are the search bytes we described on the previous page. Keep pressing N for "next" until all of the memory has been searched. You will get the following addresses: 905C and EEDC. You can ignore the one at EEDC, because it's in the middle of STK and outside the main game code, so that leaves just the one at 905C. Press E for edit and type #905C. You'll see the following bytes:

3E 05 32 1A 61

So this tells us that the computer puts the number five in box number 611A (remember that 2 byte numbers are reversed!) Now to hack the game, we simply change the number 5 at 905D to any

number of lives we want (the maximum is #FF). Now, get out of STK, reload the "Bytes: M" file and RANDOMIZE USR 24800 (the original command in the BASIC), and hey presto - you will have whatever amount of lives you wanted!

All that remains to do is to write a proper hack for it. 905D is 36957 in decimal, so your hack would be something like "MERGE the BASIC loader, and insert POKE 36957,n before the RANDOMIZE USR statement, where 'n' is the number of lives." And it's as simple as that. If you can understand what we've done so far, you're doing well, so stick at it. Any unprotected game like Sweevo's World is hacked in the same way, except you will probably have to reload STK to a different address, and you may find that in your search for #3e <lives> #32, you may come across several locations in memory outside STK that have this pattern. In this case, you'll have to use trial and error to work out which one holds the number of lives.

It is, of course, perfectly possible for you now to find 'number of lives' POKEs for any unprotected game under the sun, and in the early days of Spectrums ('82-'84), this would have been perfectly adequate. Of course, what your really need is the real nitty gritty - INFINITE lives. This is done by taking things just a step further. What you basically need to do is find out which memory location the number of lives are being put in. Then you need to search for parts of the program which put the value of that memory location
in a register, subtract one from it, and put the new value in the register back. Then you have to rewrite the code slightly so that the computer "forgets" to decrease the value in the register, and simply puts the old value back in again.

Coming back to Sweevo's World, we already know that the number of lives are stored at memory location 611A. I normally refer to this memory location as the "lives store" for obvious reasons. All we have to do is search for all the occurrences of the address of the lives store. So, search for #1A,#61, and you will find it referred to at the following addresses:

779B 8160 81A9 905F EEDC

You can ignore the one at EEDC because it's in STK, but you can also ignore the one at 905F, since that's part of the lives setting routine we discussed earlier. So the routine to decrease the number of lives must lie at one of the other locations. You should note, that any instruction that involves the lives store will begin at the byte before, because 1A and 61 are two byte operands (see page 3 for more about operands).

So for starters, press E to edit an address and type in #779A. You'll see the following:

779A - 3A 1A 61 C3 61 99

If you look up 3A in Appendix A of your Spectrum manual, you'll see it corresponds to the instruction LD A,(NN). This is simply a reverse of the LD (NN),A instruction, in that the value of a memory location is put into the A register. This is important, because subtraction of any sort can only be done in a register, and usually in the A register. After the computer has executed the three byte instruction 3A 1A 61 (which is LD A,(611A in mnemonics), it executes the instruction C3. If you look up C3 in Appendix A of your Spectrum manual, you'll see it corresponds to the instruction JP. JP is short for "jump", and is in a sense like GOTO in BASIC. What the computer does is to jump to a location in memory. As you can see, there is ambiguity as to where it is going to jump, so we need a two byte operand. Like the ones me have met before, the bytes are written the wrong way round. So C3 61 99 means JP 9961. In this case, the computer would go to address 9961, and start executing code from there. It is possible that the code to decrease the number of lives is at 9961, but is unlikely, because it pointless to have to jump to a completely different area of memory. So we'll leave this part of memory, and go onto the next instruction, at 8160. Press EDIT to leave the editing procedure, and edit address 815F. You'll see the following:

815F - 21 1A 61 35

21 is the instruction LD HL,NN. HL is another register like A,
but its main difference is that it can store two bytes at once. So LD HL,NN requires a two byte operand, whereas LD A,N only requires one. So here the instruction 21 1A 61 means LD HL,611A. The next instruction, 35 doesn't need any operands, and is the instruction we've been looking for. 35 means DEC (HL). You've already come across brackets meaning "the contents of", so as you might have guessed, DEC (HL) decreases whatever is at the memory location with the same number as HL by one. In this case, we know that HL is 611A, because we've just set it in the last instruction. So DEC (HL) will decrease the value of whatever is in memory location 611A by one in this case. But we already know that the number of lives is stored at memory location 611A. So clearly DEC (HL) is going to decrease the number of lives by one! What we need to do to make an infinite lives POKE is to somehow overwrite the DEC (HL) so that the computer doesn't decrease the number of lives. There are two things that can be done. Firstly the address containing DEC (HL) can be replaced by 0. The number 0 relates to an instruction called NOP. NOP is short for "No operand", and in short means absolutely nothing! When the computer encounters the instruction NOP, it will do nothing and execute the next instruction. So if we overwrite DEC (HL) with NOP, the computer won't decrease the number of lives, but do nothing instead. The vast majority of POKEs have the format POKE address,0 for the reasons described above. If you run Sweevo's World changing the DEC (HL) to NOP, you'll find you only get one life instead of five! In this case, you should overwrite the DEC (HL) with OR (HL). OR (HL) is a single byte instruction, B6. Don't worry about what it does, because it isn't important. What is important is to remember to do this if you only get one life. Rerun Sweevo's World, replacing the DEC (HL) with OR (HL), and you'll have your infinite lives! The DEC (HL) is at address 8162, which is 33122 decimal, while B6 is 182 decimal, so the POKE would go something like "MERGE the loader, and put POKE 33122,182 before the RANDOMIZE USR statement, RUN the program and restart the tape."

Now we've covered the rudiments of machine code involved in hacking, we can look at more detailed ways of finding POKEs.

PART 2 - Conventional Hacking Techniques

By now, you should have an idea of how simple machine code works. Now we're going to look at the usual techniques of hacking. There are five techniques, which are called Forwards Trace, Backwards Trace, Stack Trace and Interrupt Trace, in that order of difficulty. Of these, Forwards Trace and Backwards trace
are the only techniques you can use without a Multiface, and are the most reliable methods. You should only really have to use the others in exceptional circumstances, which are, on the whole, games which don't have a lives system or a GAME OVER message. Very few games use unusual routines, since it's just a hindrance to the programmer.

We'll start with Forwards Trace. You've already had a go at this from the last part, so hopefully it won't be too hard to understand. Then we'll look at the backwards trace, followed by some practical examples.

FORWARDS TRACE:

For the forwards trace, you start with the lives initialisation routine, and then work forwards from there. The first thing you need to do is to find where the number of lives are defined. We've already seen how numbers are put into memory locations, and this is exactly what happens in most games. (From now on, all the machine code programs discussed here will be written as hexadecimal bytes, and as mnemonics.)

3E XX LD A,XX (where XX is the number of lives)
32 XX XX LD (XXXX),A (where XXXX is a memory location, which we will refer to as the "lives store".)

You have already come across these instructions, but just to resume, the number of lives is put in the A register, which in turn is put in the lives store, the overall result being the number of lives being put in the lives store.

To find this, use the search function on your disassembler (QUEST 16384 on STK), and search for 3E <number of lives>. So if you had five lives, you'd search for 3E 05. You may find that this instruction occurs a few times, but you will probably find that only one actually puts the value of A in a memory location straight away. If there appear to be two occurrences where the number of lives is put into a memory location, then you may have to test all of them. You can, in general, rule out memory locations in which the number of lives is put in twice. If you want to confirm you've found the lives store, you need a Multiface. Load the game up, and while playing the game, alter the value in what you think is the lives store. If the number of lives varies, you've found the lives store. If not, try another possible location. If you can't find any memory location which looks like a lives store, you won't be able to forwards trace, I'm afraid, in which case you should try the backwards trace instead.

Once you've found the lives store, you'll have to search for occurrences of it. You are basically looking for code which takes the value out of the lives store and puts it in a register (normally the A register), decreases the number by one, and puts the new result back in the lives store. To get infinite lives, you have to overwrite the decrement instruction with a blank instruction so that the number of lives is left intact. The code you want normally takes one of two formats:

1st type:

3A XX XX LD A,(XXXX) where XXXX is the lives store
3D DEC A (we want to remove this)
32 XX XX LD (XXXX),A (putting the new value in the lives store)
2nd type:

21 XX XX LD HL,XXXX where XXXX is the lives store
35 DEC (HL) We want to remove this

You may find that there are other instructions between these. Don't worry about what they mean, just ignore them.

In order to remove the decrement instruction, we have to do one of two things. The most common is to replace the DEC A or DEC (HL) with a NOP (code 00). If this gives you only one life, replace the DEC A with OR A (code B7), or replace the DEC (HL) with OR (HL) (code B6). Don't worry why!

BACKWARDS TRACE:

Also known as "backtracking", a backwards trace starts from the GAME OVER message, and goes back from there to the lives routine. It is normally used when a forwards trace does not work or is unsuitable (e.g.: when finding infinite energy), the easiest alternative is to use a backwards trace.

The first thing you need to do is to find out what message is printed on the screen when you die. Nine times out of ten it's "GAME OVER", but there are exceptions. For GAME OVER you have to search for the following bytes :

47 41 4D 45 32 4F 56 45 52.

This is GAME OVER in ASCII (look up the codes in Appendix A of the Spectrum manual if you like). Sometimes, you may not find it, in which case just search for 47 41 4D 45 (GAME in ASCII). If that search fails, then the GAME OVER text is not in an ASCII format (this is rare, because printing routines are considerably smaller when ASCII is used), and you can't do a backwards trace.

Once you've found the GAME OVER message, you need to find out which part of the program refers to it. Normally, a print routine will load a register with the address of the GAME OVER routine, then print it. One possible register is HL (as we have met before), but also two other registers called BC and DE.

Normally, all of these three registers are referred to as "register pairs", because they are two registers, each the same size as the A register, working together, so they can address all of the memory. You want to look for the following:

01 XX XX LD BC,XXXX (where XXXX is the address of the GAME OVER MESSAGE)

11 XX XX LD DE,XXXX (as above)

21 XX XX LD HL,XXXX (as above)

(Remember that XXXX will be written "backwards" in the game's machine code!)

You should only find one occurrence of any of the above instructions. If you can't find any, repeat the search but take one away from the address the GAME OVER text is. (This is because sometimes there are Spectrum ASCII control codes such as 16 XX for PAPER XX before the actual text, and these are referred to as the start of the message). If that doesn't work, repeat the search taking one away again. Keep doing this and you'll soon find one of the above instructions.

When you've found the instruction, note down where it occurs. The part of the code you are now in is part of the GAME OVER printing routine. What you now need to do is go back through the code until you find either the code C3 XX XX (which is JP XXXX), or C9 (which is RET, which behaves exactly the same way as RETURN in BASIC.) Alternatively, you may find a blank area of memory (00 00 00 00 00 00 etc.) The address after one of these instructions is the START of the GAME OVER routine.

When you've found the start of the GAME OVER routine, you can find out which part of the code calls it. Then, to get a cheat, you can remove all parts of the code which branch to the GAME OVER routine. Search for the address of the start of the routine. You will probably find some of the following:

CD XX XX	CALL XXXX
CC XX XX	CALL Z, XXXX
C4 XX XX	CALL NZ, XXXX
C3 XX XX	JP XXXX
CA XX XX	JP Z,XXXX
C2 XX XX	JP NZ,XXXX

...where XXXX is the start of the game over routine. JP is similar to GOTO in BASIC, while CALL is similar to GOSUB (so that a RET instruction will return to the instruction after the call - except in some protection systems, but more about that later).To cheat, simply poke all three bytes of the instruction you find with 00 (so as to disable the CALL or JP). And there you have it!

Using the techniques of forwards trace and backwards trace, you should be able to hack most old, unprotected games!

PART 3 - EASY LOADING SYSTEMS

So far, you've worked out the all important basics of hacking. However, there is another, equally important facet of hacking games that you should know about.

Few games these days are unprotected. They feature "protection systems" which prevent you from breaking into a program and fiddling about with it. The difficulty level varies, but in general they use two concepts - headerless loading and decryption.

Before we do anything, I should point out that you're going to need a disassembler from now on. The machine code listings in this book use Devpac's notation, but 007 Disassembler's notation is almost identical, except it uses decimal instead of hex. Hopefully, you shouldn't get lost.

Anyway, for now, we'll forget about decryption and concentrate of headerless loaders, since they're common to all protection systems.

A headerless loader will look something like this:

```
DD 21 XX XX    LD IX,XXXX
11 XX XX       LD DE,XXXX
3E FF          LD A,#FF
37             SCF
CD 56 05       CALL #0556
```

...where XX can be any number from #00 to #FF. IX is another register similar to HL, but has slightly different properties, which you don't need to worry about right now. The value put into IX is always the start address of the block to be loaded, and the value put into DE is always the length of the block to be loaded.

So the routine works exactly like loading and saving bytes in BASIC. The only differences you should ever find are that the CALL is to a different address (#0556 is the ROM loading routine, so other CALLS are to turboloaders in RAM), the LD A,#FF has some other value loaded into A instead, or is missing, or the SCF is missing. Basically, if you see DD 21 XX XX 11 XX XX in a protection system, you can be pretty sure it will be used to load something.

Now we know how a headerless loader works, let's try and hack a real one. As an example, I've chosen Ethnipod, which was on the May 1991 YS Covertape.

First of all, load up STK at any address (I'd suggest 32768, but you don't have to) and press Z to BLOAD in the BASIC. Then use STK to list the basic, and you'll get the following:

```
10 BORDER 0: PAPER 0: INK 7: CLEAR 24999: LOAD "" CODE
65000: RANDOMIZE USR 65000
```

Therefore, we should type in CLEAR 24999:LOAD "" CODE 65000 and restart the tape. When the OK message appears, stop the tape, load up your disassembler, and have a look at address 65000 (#FDE8). Here's a complete disassembly of the code you'll find there.

```
FDE8 21 00 40   LD HL,#4000
FDEB 11 01 40   LD DE,#4001
FDEE 01 FF 1A   LD BC,#1AFF
FDF1 36 00      LD (HL),#00
FDF3 ED B0      LDIR
```

LDIR is a command we haven't met before, but it's easy to understand. It's a copying routine. The start address of the block you want to copy is put in HL, the length of the block you want to copy is put in BC, and the start address of the area of memory you want to copy it to is put in DE. So, in the example above, the area of memory from #4000 is copied to #4001 for #1AFF bytes. In short, this routine is overlaying each address in this area of memory with the byte of the previous address. The LD (HL),00 means that byte #00 is put into address #4000. Therefore, the whole of the memory from #4000 to #5AFF is filled with 0. In case you didn't know, the whole of this memory is the screen memory, so this bit of code is what makes the screen black when loading the game normally. If you want, you can change the byte at #FDED to #00 to give LD DE,#0001, so the contents of the screen memory are copied into the ROM (except that they aren't because the ROM is a read-only memory and you can't write anything into it.) This will stop the screen going black. You don't actually need to do it at all, but there we go. Continuing the disassembly:

```
FDF5 11 00 1B        LD DE,#1B00
FDF8 DD 21 00 80     LD IX,#8000
FDFC 3E FF           LD A,#FF
FDFE 37              SCF
FDFF CD 56 05        CALL #0556
```

This portion of code loads in a block of code, with the start #8000 and the length #1B00.

```
FE02 3E 00     LD A,#00
FE04 D3 FE     OUT (#FE),A
```

This part of the code includes an OUT instruction, but OUT in machine code is exactly identical to that in BASIC. So, this routine is basically the equivalent of OUT 254,0 in BASIC. If you don't know what that does, it sets the border to black.

```
FE06 11 00 40   LD DE,#4000
FE09 21 00 80   LD HL,#8000
FE0C 01 00 1B   LD BC,#1B00
FE0F ED B0      LDIR
```

This is another LDIR, and it moves the code from #8000 to #4000 for #1B00 bytes. In other words, it copies the screen picture into the screen memory so you can see it.

```
FE11 11 60 9D        LD DE,#9D60
FE14 DD 21 B4 5F     LD IX,#5FB4
FE18 37              SCF
FE19 3E FF           LD A,#FF
FE1B CD 56 05        CALL #0556
```

This part of code loads another block, with start 5FB4 and length 9D60.

```
FE1E C3 C7 61        JP 61C7
```

This part of the routine jumps to the game itself once it is loaded.

To hack the game, replace the C3 at FE1E with C9. This will put a RET at the end of all the code, so the loader will return to BASIC when all loading has finished.

When the OK message comes up, you can hack the game as you've done with unprotected games. If you load STK into address #6000 (24576 decimal), and hack the game using a forwards trace, you'll

eventually find that changing #EF09 to 0 gives you infinite lives for player one. Then to start the game, type

RANDOMIZE USR 25031

(61C7 in decimal), and bingo!

To write a hack, we need to rewrite the BASIC loader, but make the modifications so we can put POKEs in:

```
10 CLEAR 24999
20 LOAD "" CODE 65000
```

This comes directly from the BASIC loader and loads the small headerless loader code.

30 POKE 65054,201

This means that control will return to BASIC when all the headerless code has been loaded

40 RANDOMIZE USR 65000

This starts off the headerless loader.

50 POKE 61193,0

This is the infinite lives POKE

60 RANDOMIZE USR 25031

This starts the game itself. Easy when you know how!

Now we know how a simple headerless loader works, let's crack a turboloader. There are loads of YS covertape games which have a suitable loader, but I'm going to choose Pixy the Microdot 2, although you'll find that any YS game which uses blue, black and magenta stripes when loading is almost identical.

First of all, load up STK at address 58550 (you'll find out why later on), to find out what the BASIC loader has to say, using the same method as with Ethnipod. It starts running at line 0.

```
1 BORDER 0:PAPER 0:CLEAR 64999:LOAD "" CODE
2 RANDOMIZE USR 65146
20 CLEAR 64999:LOAD "mc" CODE:LOAD "pixldsy" CODE:SAVE "t:":SAVE "PIXY" LINE 1:SAVE
"x"
CODE 65146,200:LOAD "screen" SCREEN$:RANDOMIZE USR 65000
```

The BASIC starts at line 1. The commands should be obvious to you. Type CLEAR 64999:LOAD "" CODE, start the tape, and load in the first block of code. Stop the tape when the OK message comes up.

Now load your disassembler and examine the code at 65146, which is FE7A hex.

```
FE7A F3          DI
```

DI is short for "disable interrupts". What are interrupts, I hear you ask? Well, imagine you're watching TV when suddenly, someone says "We interrupt this program to give you an important newsflash!" Then, after the newsflash, the program you were watching resumes. Well, computer interrupts work in exactly

the same way. In fact, every fiftieth of a second, a program is "interrupted" by the computer, which then checks to see if you're pressing any keys, and resumes the original program. The command DI simply stops this happening, and your program continues without any interruption! This makes the program run faster. However, you CANNOT get back to BASIC by a RET command, because the computer won't be checking the keyboard, and so it has effectively locked up. To get round this, you must execute the command EI (enable interrupts) first, so control can be resumed. Don't worry about doing this now, though.

FE7B 31 60 61 LD SP,6160

This is a new instruction. SP (short for "stack pointer") is a 16-bit register, like BC, DE and HL. However, it's far more important as far as BASIC is concerned. In machine code, there are two ways of storing numbers. The first, using memory locations, we've already come across. However, there is another method by storing numbers on what is called a stack. Think of a stack as a big spike on which you can push pieces of paper with information on. Then, later on, you can take them off the stack and use them. If you think about it, if you put the numbers 1, 2 and 3 on the stack, in that order, you'll have to take 3 off first, then 2, then 1 (think about it). And it's the same in machine code. There are instructions which enable values of registers to be put on the stack, and which enable the value on the top of the stack to be taken off and put in a register. The stack, like everything else, has to go somewhere in memory. The SP (stack pointer) register gives the address of the top of the stack. So LD SP,6160 will mean that the stack is to start at address 6160.

This is bad news if you want to return to BASIC, because the Spectrum's ROM program puts lots of information on the stack, so if you change the stack pointer, it's going to receive garbage when it takes all the values off what it thinks is the stack. And that, of course, will mean a crash. So the general rule is LEAVE THE STACK POINTER ALONE!
You can change the value of the stack pointer using CLEAR from BASIC. If a machine code instruction has LD SP,XXXX, you can type CLEAR (XXXX)-1. So here, we should CLEAR (#6160)-1 = #615F. Bear in mind that the value will have to be in decimal, which is 24927. So exit from STK, CLEAR 24927, and go back into it again. This will mean that later on we can do the EI / RET as described above. Then you have to remove the LD SP instruction, which is most easily done by changing FE7B to 21, so it reads LD HL,6160. This is harmless in this case. Carrying on through the code....

```
FE7E DD 21 00 40      LD IX,4000
FE82 11 00 1B         LD DE,1B00
FE85 CD 97 FE         CALL FE97
```

As you can probably see, this loads a headerless block, the loading screen, in fact. However, you'll notice, as I said earlier, that some of the other commands (LD A,#FF and SCF) are missing, and the CALL goes to a different address. This is because it's a turboloader.

```
FE88 DD 21 60 61      LD IX,6160
FE8C 11 4B 83         LD DE,834B
FE8F CD 97 FE         CALL FE97
```

This loads another block, start 6160 and length 834B. This means, that all the memory from 6160 to E4AB will be overwritten. Fortunately, you loaded STK into address 58550, which is E4B6 hex, so it won't be overwritten. Clever, eh? Meanwhile....

```
FE92 30 F4            JR NC,FE88
```

Something I haven't told you yet is that after a headerless load, a JR NC will result in that JR if there is a tape loading error. So, if there was a tape loading error in loading this game, the JR NC,FE88 would be executed (so that computer would try and reload the block).

FE94 C3 60 61 JP 6160

This starts the main program running.

To crack the loader, therefore, POKE FE94 with FB (for EI) and FE95 with C9 (for RET), along with the modifications I've already told you about. Then RANDOMIZE USR 65146, and restart the tape (it is possible that you didn't stop the tape quickly enough the previous time, so you'll miss the turboload header, in which case wind back just before it). When the game has finished loading, an OK message will appear.

And that's it! Well, actually it's not, because the game is actually compressed, and needs to be unpacked first. Don't worry, because it's easy to hack. Go into STK again, and look at address 6160. You're looking for a JP instruction to the game, which is what is executed when the game is unpacked. You'll find it at 61A3. So POKE 61A3,FB and 61A4,C9 (for an EI / RET), and RANDOMIZE USR 24928. Wait a few seconds until BASIC returns. And there we are - you've cracked the loader!

You might be wondering how you can tell that the game is compressed. Well, there are two things. Firstly, the JP from the loader (6160) is to a very low address in the usable RAM (which only starts at 5B00). But more noticeable, you won't be able to do a forwards trace or a backwards trace until you run the decompressor. In fact, in general, if you think you should be able to forwards trace or backwards trace a game for infinite lives, and haven't overloaded any important code with a disassembler, but nothing happens, its worth looking at the start of the code executed and seeing if there's a JP a bit later on to a completely different address.

So now, perhaps, we should write a complete hack for the game.

10 CLEAR 24927:LOAD "" CODE

This is from the BASIC loader and loads in the first block of code. We've changed the CLEAR though, so the stack is in the right place.

20 POKE 65147,33

This changes the LD SP,6160 into LD HL,6160 so the SP isn't tampered with.

30 POKE 65172,251:POKE 65173,201

This changes the JP 6160 to an EI / RET so control will return to our hack once the game has loaded.

40 RANDOMIZE USR 65146

This starts the game loading

50 POKE 24995,251:POKE 24996,201

This changes the JP 86CE to an EI / RET so control will return to our hack once the game has decompressed

60 RANDOMIZE USR 24928

This starts the game decompressor.

70 POKE 28402,0

This is the infinite lives POKE, which you'll find out when you do a forwards trace on the uncompressed game.

80 RANDOMIZE USR 34510

This is the start of the game.

Now that you've done that, why not crack another game which uses the same loader? They're nearly all the same, except some of the JP addresses will be different. And then when you've done that, why not have a look at some other headerless loaders - most games by Codemasters use them.

You will find, however, that you will sometimes have to overwrite some of the memory with your disassembler. There's no easy way to tell where it should be, I'm afraid, so you'll have to take pot luck. If your forwards trace and backwards trace are both unsuccessful, try loading the disassembler elsewhere in memory, or look to see if the game is compressed.

PART 4 - DECRYPTERS

By this point, you should be familiar with headerless loaders, which take up the bulk of most protection systems these days. However, there is one other important aspect of protection that you need to know about if you are to crack the more complex protection systems. It's encryption.

Encryption works like a secret code. You start off with something unintelligible, then you use the "secret rules" to change it into something that makes sense. So, IBDLFST doesn't make much sense, but if you take the previous letter in the alphabet each time, you get HACKERS, which makes perfect sense. Encryption works in roughly the same way. We've already seen that a loading system occupies a small area of memory. An encrypted loading system will appear as a block of code which makes absolutely no sense
whatsoever. There will also be a short program which changes all this nonsense into workable code so the loading system can be run. Some really tough loading systems, have more than one decrypter, such as the Alkatrazz loading system which has 250 of the damn things; in practice I've got through about 25 before going mad and hacking something else (don't worry, there's another way of hacking them which I'll tell you about later.)

To make it easier for you to understand decryption, its best to have a look at a real loading system. As an example, I've chosen Impossaball, which was on the YS#75 covertape. Load up STK at address 50000 (it's a safe address), and see what the BASIC has to say for itself by BLOADing it.....

```
10 CLEAR 64530: LOAD "" CODE:RANDOMIZE USR 64531
20 LOAD "" CODE 16384
21 FOR i=1 TO 40 STEP - RANDOMIZE USR 50000
```

Fair enough - so enter CLEAR 64530:LOAD "" CODE and start the tape. The first code block loads in, and then - it crashes! What's going on? Don't worry, you have just fallen victim to the first type of encryption - BASIC encryption.

In BASIC, any numerical constant (technotwaddle for "number") between 0 and 65,535 is stored in the memory as follows. First, there is the number in it's ASCII form, which takes between 1 and 5 bytes. So, the number 1234 will appear as #30,#31#,#32,#33 here. Then, come the numbers #0D,#00 and 00 (this is always the case). Then, there is the number stored in it's two byte form (as in machine code). What happens it that the computer prints the ASCII form (which is what you get when you LIST a program), but
uses the two-byte form in calculations and expressions. The upshot of all this is that what you see isn't always what you get! As an example, type in this:

```
1 PRINT 1
```

Now type POKE 23760,50 (which changes the ASCII value of the number 1 from "1" to "2"). Now if you LIST the program, you will see 1 PRINT 2, but if you RUN the program, the computer will print the number 1 instead! Needless to say, protection systems use the same idea. Sometimes, the numbers listed are obviously fake (if you list a program and you get something like 0 CLEAR 0:RANDOMIZE USR 0 it's obviously got encrypted BASIC), but some programs, like the Impossaball are not obvious at all until you try executing what you see.

There was a program called *List printed ages ago in YS, but if you haven't got that, I've reprinted it here. So type it in, SAVE it to tape, RUN it and reload the Impossaball BASIC loader.

```
loading LINE 10 LEN 81
```

```
10 CLEAR 25599: LOAD "" CODE: RANDOMIZE USR 64512
20 LOAD "" CODE 16384
21 FOR i=1 TO 40 STEP ????....
```

Now that's what you really get! Type CLEAR 25599:LOAD "" CODE and load in the first block of code. When that's done, load up your disassembler, and disassemble address 64512 (FC00 hex) to have a look at the decrypter.

```
FC00 01 99 02    LD BC,#0299
FC03 21 13 FC    LD HL,#FC13
FC06 11 14 FC    LD DE,#FC14
```

This part of the program sets up all the initial values for the decrypters in some of the registers.

```
FC09 1A          LD A,(DE)
```

We've come across brackets before, but briefly what happens here is that the contents of the address with the value of the DE register (which starts of as #FC14) are put into the A register. So now the A register could contain any byte.

```
FC0A AE          XOR (HL)
```

We haven't seen XOR before, so I'll explain what it does. It is in technical terms a Boolean Operation. You may have seen XOR gates if you studied (or are studying!) Physics, Electronics or Computer Science at school. An XOR gate has two inputs, which can each either be 0 or 1, and one output, which can be either 0 or 1 as well. If the two inputs are the same (0 and 0, or 1 and 1), the output is 0, otherwise it is 1. In machine code, you XOR the A register with a number or contents of a register, and what happens is that each bit in the A register is XORed with the same bit in the number or register contents, and the result is stored in the A register. If you're confused, look at the example below.

```
Contents of the A register      00100101
Number to be XORed with         01010011
Result                          01110110
```

(Notice that all the numbers are in binary - see your Spectrum manual for more information about this).

If you still don't understand, just remember that an XOR will change the contents of the A register. That's all you need to remember for now.

Continuing the disassembly...

```
FC0B 77          LD (HL),A
```

This puts the value of A (which has just been changed) into the bytes at the address with the value of the HL register (which starts off as #FC13). So, the routine has basically taken a byte out of a memory location, changed it a bit, and put the altered value back again. This is decryption in its most obvious form - the changed values make up a working machine code program.

```
FC0C 23          INC HL
FC0D 13          INC DE
FC0E 0B          DEC BC
```

I don't think I've mentioned INC before, but it's basically the opposite of DEC in that it increases the value in whatever register. So the values in the HL and DE registers are incremented, and the value in the BC register is decremented.

```
FC0F 78LD A,B
FC10 B1        OR C
FC11 20 F6     JR NZ,#FC09
```

This is a standard piece of code, and it essentially means "If BC isn't 0, then jump to #FC09". In other words, another byte will be decrypted until the value of BC is 0 (which happens when everything has been decrypted), and the decrypter ends.

Continuing the disassembly...

```
FC13 5C        LD E,H
FC14 9F        SBC A,A
FC15 A5        AND L
FC16 5B        LD E,E
FC17 13        INC DE
FC18 5B        LD E,E
....
```

Hang on - this code doesn't make sense! It has no relation to the code above, and the instruction LD E,E is pointless anyway. Well, you'll remember that the initial value of decryption is #FC13, which means that all the code from there onwards has to be decrypted. So we'll have to crack the decrypter to go any further.

Sometimes, it is possible to put an EI/RET instruction directly after the decrypter, but this is not possible here, as you will see. So instead, we'll have to move the decrypter somewhere else in memory, and put the EI/RET on the end (in actual fact we don't need the EI because there is no DI command in the decrypter). This is easily done by using the LDIR command. Type in the following program:

```
1 FOR N=23296 TO 23310:INPUT A:POKE N,A:NEXT N
```

Now RUN it and enter the following numbers:

```
33,0,252,17,128,91,1,1,18,0,237,176,54,201,201
```

The program you have just typed in is this:

```
LD HL,FC00
LD DE,5B80
LD BC,0012
LDIR
LD (HL),C9
RET
```

Now RANDOMIZE USR 23296 and the decrypter will be copied to 5B80 and a RET will be stuck on the end. Just RANDOMIZE USR 23424 (5B80 in decimal) to run the decrypter. When the OK message comes up, restart the disassembler and look at FC13 again:

```
FC13 C3 3A FE  JP #FE3A
```

This jumps to #FE3A, to start the loading.

```
FE3A F3        DI
FE3B 21 00 58  LD HL,#5800
FE3E 11 01 58  LD DE,#5801
FE41 01 FF 02  LD BC,#02FF
FE44 36 00     LDIR
```

As you already know, this makes the screen black.

```
FE48 CD 81 FE  CALL #FE81
FE4B CD 00 80  CALL #8000
```

If you look at the code at #FE81, you'll see it's a headerless loader. The routine at #8000 prints the loading screen.

```
FE4E CD 81 FE  CALL #FE81
FE51 CD 00 64  CALL #6400
```

This loads another block (the main game), and prints another screen (the border for the game).

```
FE54 F3 DI
FE55 31 FF FF  LD SP,#FFFF
```

This disables interrupts and changes the stack pointer, so we'll have to change that in the final hack.

```
FE58 21 00 6D  LD HL,#6000
FE5B 11 00 5B  LD DE,#5B00
FE5E 01 00 8F  LD BC,#8F00
FE61 ED BO     LDIR
```

This is another LDIR command, but if you know how the Spectrum's memory is organised, you will see that the BASIC system variables are overwritten, which means we can't return to BASIC when the game has loaded. Fortunately, there's a short routine to get round this, which I'll explain in a mo.

```
FE63 21 71 FE  LD HL,#FE71
FE66 11 00 F0  LD DE,#F000
FE69 01 10 00  LD BC,#0010
FE6C ED B0     LDIR
FE6E C3 00 F0  JP F000
```

This moves the code from FE71 to F000, and jumps to F000. Obviously, then, the code at FE71 is important.

```
FE71 21 00 FC  LD HL,#FC00
FE74 11 01 FC  LD DE,#FC01
FE77 01 FF 01  LD BC,#01FF
FE7A 36 00     LD (HL),0
FE7C ED B0     LDIR
FE7E C3 00 80  JP #8000
```

This routine wipes out all the memory from #FC00 to #FC01, but in actual fact you don't need to do this. Then it jumps to #8000, which is the start of the game. So, we can put a machine code routine to put a POKE in, and then jump to #8000. However, we've got to find the POKE first, and there's the problem that we can't use BASIC because it is overwritten. Luckily, there is a short routine you can use which will cause a NEW to a certain address. Put it into the above program by typing in the following:

```
1 FOR N=65137 TO 65144:INPUT A:POKE N,A:NEXT N
```

Then RUN it, and enter these numbers in turn:

243,175,17,0,95,195,203,17

The program you've just typed in is the following:

```
DI
XOR A
LD DE,#5F00
JP #11CB
```

DI disables interrupts, XOR A loads A with 0 (think about what would happen if you XORed the value of the A register with itself, LD DE,#5F00 means we want to NEW up to #5F00 (this value can be changed from about #5D00 to #FFFF), and JP #11CB starts the NEW.

Now RANDOMIZE USR 65082 (#FE3A in decimal), and restart the game tape. When the computer resets, you can load STK into address 24320 to find POKEs.

Phew! And that's about all of that protection system cleared up. If you can get your way through that lot, I think you're probably ready to have a go at a "commercial" protection system. First, though, we'll write a complete hack for the game.

```
10 CLEAR 25599:LOAD "" CODE
```

This is from the BASIC loader, and it loads in the first machine code block

```
20 FOR N=23296 TO 23310:READ A:POKE N,A:NEXT N
```

This line POKEs in the machine code program to move the decrypter.

```
30 DATA 33,0,252,17,128,91,1,19,0,237,176,54,201,201
```

And here's the actual machine code itself

```
40 RANDOMIZE USR 23296
```

This line calls the decrypter and returns to BASIC

```
40 FOR N=65137 TO 65143: READ A:POKE N,A:NEXT N
```

This line POKEs in our hacking program

```
50 DATA 175,50,??,??,195,0,128
```

And here's the hacking program, which loads ???? with 0 (which is the infinite lives POKE), and jumps to #8000.

```
60 RANDOMIZE USR 65082
```

This starts the whole loading system off, with the POKE firmly in place.

And that's about it. A bit of a long piece of work, but it was worth it!

PART 5 - Advanced Hacking Methods

Remember in Part 2, when I said there were other ways of hacking, apart from forwards and backwards tracing? Well, you can find them here. But you need a Multiface to be able to use them. The two we're interested with here are what I call a stack trace and an interrupt trace.

STACK TRACE:

We've already come across the stack as a means of storing numbers. What you haven't come across yet is how the stack is used. Well, in a CALL to a subroutine, what actually happens is that the return address from the subroutine (which is the address after the CALL instruction) is stored on the stack, and with a RET, the top value of the stack is taken off and jumped to. With a Multiface, the value on the top of the stack is the return address to the program. and subsequent values refer to return addresses in subroutines. To do a stack trace, load and play the game, and wait until the "death effect" occurs - this may be a beep, a flashing border or something else recognisable. Now quickly press the Multiface button during this effect - if you're too slow, you won't get the values you're looking for (so return to the game and die again). Now, look at the value of the stack pointer (your Multiface manual will tell you how to do this), and write down all the values on top of the stack for the first ten bytes. All numbers are stored in the normal reversed two-byte form, so if the bytes on the top of the stack were #00,#80,#80,#70,#90,#60, the values would be #8000,#7080 and #6090. Have a look at all of these addresses - you should find that some of them are addresses right after CALL instructions. Now for the hacking bit - go to one of these address and write down the two bytes there. Then change them to the magic codes #18 and #FE (this is the machine code version of JR -2, which is an endless loop, a bit like 1 GOTO 1 in BASIC). Restart the game, and hopefully, you'll find that the game pauses as soon as you do something which would normally result in you losing a life! (If not, replace the #18FE with the original two bytes, look at another address on the "hit list" and repeat the whole procedure). Once you've found a target address, try putting a RET (#C9) at the start of the subroutine. If this just cancels the death effect, but you still "die", activate the Genie Disassembler if you have it (or use the NEW routine in Part 4 at any address, then load in STK or Devpac somewhere far away from the area of memory you're looking at), and search for CALLs to this routine. Then go back from this CALL until you find a RET or a JP, and search for the address of the instruction after this (if nothing comes up, search for one more than this, then two more etc.). You will hopefully either see one of these:

JP Z
JP NZ
JP C
JP NC

(The JP may be a JR or a CALL instead)

Simply overwrite this instruction with NOPs to get immunity or something similar. On the other hand, when searching from the CALL address, you may find a JP Z or JP NZ, etc. Change this to an unconditional JP to get immunity.

INTERRUPT TRACE:

This involves looking at the interrupt routine in the game. Since the whole routine must be executed in 1/50th of a second, the routines are usually quite short, especially if there is a LDIR or something similar. Most of the time you'll find infinite time in this routine (because interrupts work in real time, so its an ideal place to put a time routine), and you need a Multiface to find it. Load the game and start playing as normal. Then activate the Multiface, and have a look at the I register. If the value is #3F, there are no special interrupts, so forget about an interrupt trace altogether (but you can use a stack trace which will make the clock loop round to 99 or whatever when it reaches 0). If it is between #80 and #FF (and if it's

not in that range and not #3F you've probably crashed the computer!), go to the address #100 time that of the value in the I register (so if the value of I is #F0, look at #F000). You will see an area of memory filled with the same number. Go to this address (if this area of memory is filled with #FE, go to #FEFE etc.) There will either be a jump to the interrupts routine, or the interrupts routine itself. Have a look at the routine, and somewhere you will see the commands to decrease the timer - just remove the DEC instruction to get infinite time.

PART 6 - Commercial Protection Systems

If you can understand everything we've done so far, you can now probably crack just about any budget game or YS covergame that's thrown at you. And indeed, you can probably get featured in "Practical POKEs" month in month out (like a load of anonymous hackers, I might add!) by using the knowledge you've got. However, if you want to become a hacking legend, you should have a go at some of the numerous commercial protection systems, which have been written by freelancers for software houses. I think we're getting seriously into Multiface territory now, but I'll try and do as much as possible with only STK and 007 Disassembler.

I think I should also say that cracking commercial protection systems is NOT easy. The code is deliberately badly structured and obscurely coded to put you off, so you'll have to persevere. You really do need something like a Multiface or Devpac to crack some of these systems, because they can overwrite the system variables in BASIC, and you sometimes need to know the values of certain registers, which is impossible to determine using BASIC. You'll also need some games other than YS stuff to hack, but fortunately, protection systems such as Speedlock or Alkatrazz are so common, you're bound to have a game with one of them on. I'll be doing examples specific to one game, but you'll find that another game with the same protection is pretty much the same, except you're likely to find that some of the addresses will be different.

Before we start, I'd just like to point out that I'll be referring to the term "breakpoint" a lot. This is simply a small bit of code which will stop the program dead in its tracks. Using DEVPAC, you just press the W key. On a Multiface, you do the same as a stack trace, by writing down the two bytes at the place you want to put a breakpoint, then replacing them with #18 and #FE. If you are using 007 disassembler and/or STK, you'll need to put a jump to the start of the program (#C3 #00 #40 for 007 disassembler; STK varies depending on where you put it).

So let's start with something relatively simple....

BLEEPLOAD

"Bleepload" first appeared on Firebird games around March 1987, and was used by them on every release by them from then on until their demise in 1989. It emulates a BBC loading system in that each file loads in a series of blocks, which are numbered in hexadecimal. The hardness is not because it uses non standard
code, it's just that it jumps around so much in memory you need to put in an awful lot of software patches. I'll be hacking Bubble Bobble as an example.

First of all, load in *Hack, and load in the BASIC loader.

Bubble LINE 10 LEN 179

```
10 REM
20 CLEAR 50000
30 BORDER 0:PAPER 0:INK 0:CLS
40 PRINT AT 1,7;PAPER 1;INK 7;"BUBBLE BOBBLE"
50 LOAD "Bobble" CODE 52480
60 RANDOMIZE USR 52480
```

There is absolutely nothing difficult about this BASIC loader, so just type CLEAR 50000:LOAD "" CODE and start the tape to load the first block of code. Stop the tape when it's loaded, and load in your disassembler into address 32768 (it's a safe one), and have a look at the code at CD00.

```
CD00 3A 5C 5B  LD A,(#5B5C)
CD03 32 00 60  LD (#6000),A
```

This takes the byte at #5B5C and puts it in #6000. #5B5C is the system variable for the 128K page number, in case you're interested.

```
CD06 3E 02     LD A,#02
CD08 CD 01 16  CALL #1601
```

This is a standard ROM routine, and all it does it to tell the computer we want to print something on the screen.

```
CD0B AF        XOR A
CD0C 32 6B 5C  LD (#5C6B),A
```

As you may be aware, poking #5C6B (23659 decimal) with 0 will cause the computer to crash if you press BREAK or return to BASIC. So POKE CD0E,0 which changes it to LD (#006B),A; this is harmless.

```
CD0F CD CE CE          CALL #CECE
```

The routine at #CECE prints the message "Searching" on screen.

```
CD12 10 09     DJNZ #CD1D
```

We haven't come across the command DJNZ before. It basically means "decrease the value in the B register, and jump if B is not zero."

```
CD14 11 08 FF  LD DE,#FF08
CD17 16 00     LD D,0
CD19 CD 1A CE  CALL #CEA1
```

This routine prints the number 00 on screen.

```
CD1C 3E 08     LD A,#08
CD1E 32 15 FF  LD (#FF15),A
CD21 CD 74 CD  CALL #CD74
```

This routine loads in a block of code from tape (in actual fact the start address is #FE00 and its length is #100 bytes).

```
CD24 3E 00 FA  LD A,(#FE00)
CD27 FE 64     CP #64
CD29 20 F6     JR NZ,#CD21
```

This routine loads A with the value at #FE00. The CP instruction compares the value in the A register with something, in this case the number #64. If there is no match, the routine jumps back to #CD21, otherwise it continues. This routine actually checks to see if the block is found.

```
CD2B 3A 01 FF  LD A,(#FF01)
CD2E BA        CP D
CD2F 28 05     JR Z,#CD36
```

This routine checks to see if the block has been loaded successfully. If so, it jumps to #CD36, otherwise it continues.

```
CD31 CD 84 CE CALL #CE84
CD34 18 EB    JP #CD21
```

This routine prints up the "loading error" message, and attempts to load the block again.

```
CD36 CD 30 CE CALL #CE30
```

The routine at #CE30 is a decrypter (have a look - do you see why?), which decrypts the block loaded in i.e.: from #FE00 to FEFF. You don't need to crack it yourself.

```
CD39 BE       CP (HL)
CD3A 28 05    JR Z,#CD41
```

This routine reloads the block if the value of A equals the value at (HL). Don't ask me why.

```
CD41 CD 5F CD CALL #CD5F
```

This routine moves the code at #FE00 to where it should really be in memory.

```
CD44 CD 5D CE CALL #CE5D
```

This routine prints the "loading" message on the screen, but this should in actual fact be "loaded", because the block has just been read in at this point of the code.

```
CD47 21 04 FF  LD HL,#FF04
CD4A 7E        LD A,(HL)
CD4B 23        INC HL
CD4C 3D        DEC A
CD4D 20 FC     JR NZ,#CD4B
CD4F 23        INC HL
CD50 23        INC HL
CD51 23        INC HL
CD52 23        INC HL
CD53 7E        LD A,(HL)
CD54 2B        DEC HL
CD55 2B        DEC HL
CD56 2B        DEC HL
CD57 E6 07     AND #07
CD59 3C        INC A
CD5A 32 15 FF  LD (#FF15),A
CD5D 14        INC D
CD5E E9        JP (HL)
```

This routine starts off with HL as FF04, then does a lot of sums, and comes out with a value in the HL register, which it jumps to after its loaded the block. This is what we need to hack. So POKE CD5E with C9 and RANDOMIZE USR 52480 - you'll find out it loads in one block and then stops. However, this isn't
much use as you can't find out the value of the HL register. So put this routine somewhere, such as #5B00.

```
5B00 CD 00 CD  CALL #CD00
5B03 22 10 5B  LD (#5B10),HL
5B06 C9        RET
```

This routine simply loads the first block, and puts the value of HL in #5B10 so we can find out what it is from BASIC. Now rewind the tape before the first Bleepload block again, and RANDOMIZE USR 23296. When that's finished, type PRINT PEEK 23312+256*PEEK 23313. You should get the answer 65293, which is #FF0D. Disassemble this address.

```
FF0D C3 21 CD  JP #CD21
```

This will go back and load the next block from tape. We can crack it in the same way as the first be changing our routine at #5B00.

```
5B00 CD 00 CD  CALL #CD00
5B03 CD 21 CD  CALL #CD21
5B06 22 10 5B  LD (#5B10),HL
5B09 C9        RET
```

Now wind the tape back to the first Bleepload block again, RANDOMIZE USR 23296 and start the tape. When the OK message comes up, type PRINT PEEK 23312+256*PEEK 23313, and you should get 65286 which is FF06 hex. Disassemble this address.

```
FF06 C3 21 CD   JP #CD21
```

This goes back and loads another block. By now, you might have guessed that the value of HL will always contain the address of a JP #CD21 instruction - except for the last block which will jump elsewhere. Now we can write a routine which will load any block as long as it jumps to #CD21 at the end. I'm putting the routine at #CCEC, because it's right next to the loading system, and hence is unlikely to be overloaded (although it could be, in which case we'd just put the routine elsewhere). The routine goes like this.

```
CCEC CD 00 CD CALL #CD00
```

This is just loading the first block

```
CCEF CD 21 CDCALL #CD21
```

This loads in a block from tape.

```
CCF0 23        INC HL
CCF1 7E        LD A,(HL)
CCF2 2B        DEC HL
```

This routine loads the A register with the value of (HL+1). This will be #21 if another block is to be loaded.

```
CCF3 FE 21     CP #21
CCF5 20 02     JR NZ,#CCF8
CCF6 18 F7     JR #CCEF
```

This compares the value in the A register to 21. If there is no match, then the routine jumps to the end to preserve the value of HL, and to return to BASIC. Otherwise, it goes back to load another block.

```
CCF8 22 FE CC  LD (#CCFE),HL
CCFB <breakpoint>
```

This puts the value of HL in address #CCFE, then returns to control of the disassembler.

Now, run this routine (RANDOMIZE USR 52463), rewind to the first Bleepload block, and start loading. The program will now load blocks 00-2D, and return to control of the disassembler. The value at #CCFE is #FF06, so disassemble this address.

```
FF06 C3 00 5B    JP #5B00
```

Now disassemble #5B00, which is the real meat of the loading system!

```
5B00 DD E5       PUSH IX
5B02 CD 74 CD    CALL #CD74
5B05 CD 30 CE    CALL #CE30
5B08 28 07       JR Z,#5B12
5B0B 06 00       LD B,#00
5B0D CD 84 CE    CALL #CE84
5B10 18 F0       JR #5B02
```

This routine loads in another block of code, and will jump to 5B12 when it has been successfully loaded.

```
5B12 F3          DI
5B13 E1          POP HL
5B14 2E 00       LD L,#00
5B16 ED 5B E7 FE LD DE,(#FEE7)
5B1A 1A          LD A,(#DE)
5B1B AE          XOR (HL)
5B1C 24          INC H
5B1D AE          XOR (HL)
5B1E 25          DEC H
5B1F 12          LD (#DE),A
5B20 2C          INC L
5B21 IC          INC E
5B22 20 F6       JR NZ,#5B1A
```

This routine is a decrypter, which decrypts some of the code we just loaded in from tape.

```
5B24 ED 5B E7 FE LD DE,(#FEE7)
5B26 21 40 5B    LD HL,#5B40
5B2B 1A          LD A,(DE)
5B2C AE          XOR (HL)
5B2D 77          LD (HL),A
5B2E 1C          INC E
5B2F 2C          INC L
5B30 20 F9       JR NZ,#5B2B
```

This code decrypts some more code loaded in from tape, but it puts it at #5B40, which is right in the middle of the code we are working on at the moment. So put a breakpoint at #5B32 (the first instruction after the decrypter), and jump to #5B00 (because we haven't executed any of the code from #5B00 onwards yet!)

```
5B32 21 00 00    LD HL,#0000
5B35 22 B0 5C    LD (#5CB0),HL
5B38 2E 02       LD A,#02
5B3A 32 6B 5C    LD (#5C6B),A
```

This puts the value #0000 into #5CB0, but I'm not sure why, because #5CB0 is an unused system variable. It then changes the value of #5C6B to #02, which is what it was originally before it was changed to protect the loader.

```
5B3D ED 5B E7 FE    LD DE,(#FEE7)
5B41 2A E9 FE       LD HL,(#FEE9)
5B44 1A             LD A,(DE)
5B45 AE             XOR (HL)
5B46 77             LD (HL),A
5B47 23             INC HL
5B48 IC             INC E
5B49 20 F9          JR NZ,#5B44
5B4B 3A EC FE       LA A,(#FEEC)
5B4E BC             CP H
5B4F 20 F3          JR NZ,#5B44
```

This is another decrypter, which works in exactly the same way as the others.

```
5B51 31 FF 60   LD SP,#60FF
5B54 21 00 CF   LD HL,#CF00
5B57 11 00 40   LD DE,#4000
5B5A 01 00 1B   LD BC,#1B00
5B5D ED B0      LDIR
5B5F 21 00 EA   LD HL,#EA00
5B62 11 00 61   LD DE,#6100
5B65 01 00 10   LD BC,#1000
5B68 ED B0      LDIR
```

This code moves all the decrypted code to where it should be. This includes the loading screen (as you can see by the reference to #4000.)

```
5B6A 3E 65      LD A,#65
5B6C 32 00 5B   LD (#5B00),A
5B6F 21 0F 14   LD HL,#140F
5B72 22 01 5B   LD (#5B01),HL
5B75 21 0F 00   LD HL,#004F
5N78 22 03 5B   LD (#5B03),HL
5B7B CD 00 FA   CALL #FA00
```

This code loads the next Bleepload block, from 00 to 87, but will return to 5B7E when it's finished.

```
5B7E 21 00 40   LD HL,#4000
5B81 11 01 40   LD DE,#4001
5B84 36 00      LD (HL),0
5B86 01 FF 1A   LD BC,#1AFF
5B89 ED B0      LDIR
5B8B 3E 66      LD A,#66
5B8D 32 00 5B   LD (#5B00),A
5B90 21 0A 0A   LD HL,#0A0A
5B92 22 01 5B   LD (#5B01),HL
5B96 21 0D 0A   LD HL,#0A0D
5B99 22 03 5B   LD (#5B03),HL
5B9C CD 00 FA   CALL #FA00
```

This code blanks out the screen and loads some code into it. Some Bleepload games do not have this code, and it is only used on games who's game code overwrites the loading system at #FA00.

```
5B9F 21 00 40    LD HL,#4000
5BA2 11 00 FA    LD DE,#FA00
5BA5 01 00 06    LD BC,#0600
5BA8 ED B0       LDIR
```

This moves the code loaded from the screen to #FA00 (where it should be).

```
5BAA 3A 00 60  LD A,(#6000)
5BAD 32 5C 5B  LD (#5B5C),A
5BB0 31 A7 61  LD SP,#61A7
5BB3 CD 8E 02  CALL #28E
5BB6 28 1D     JR Z,#5BD5
```

This routine restores the value of #5B5C that was stored in #6000 right at the very start. It then sets the stack to #61A7, and calls the ROM key check routine. If no key is pressed (and there shouldn't be), the routine jumps to 5BD5. In fact, it must jump there, otherwise it would attempt to load a normal headerless block, and there are none!

```
5BD5 C3 BC F5  JP #F5BC
```

This is what we've all been waiting for - the JP to the game itself. You can simply put POKEs on the end of #5BD5, and follow them with a JP #F5BC to load the game. For now, though, it might be a good idea to put the NEW routine up to #61A7 there, instead, and JP to #5B32 (where we left off). Then load the rest of the game, which will reset at the end, enabling you to load in STK, Devpac or whatever.

Now we've gone all the way through Bleepload, perhaps we should write a hack for the complete game. However, I'm going to put most of the hack in machine code, rather than have long lines of decimal DATA statements. You should be able to convert the machine code into DATA statements and get a short program which reads them in and POKEs them into memory. The only thing that has to be done from BASIC is the CLEAR 50000:LOAD "" CODE 52480 from the BASIC loader. The machine code hack will consist of the first routine we wrote, followed by a few patches to the main loading system, so that the JP to the game is overwritten with our POKEs. I'll be putting it at #CC80, because it's a safe place in memory.

```
CC80 3E C9     LD A,#C9
CC82 32 5E CD  LD (#CD5E),A
```

This puts a RET in place of the JP (HL) at #CD5E so we can CALL the loading system.

```
CC85 CD 00 CD CALL #CD00
```

This loads in the first Bleepload block.

```
CC88 CD 21 CD CALL #CD21
```

This loads in another Bleepload block.

```
CC8B 23        INC HL
CC8C 7E        LD A,(HL)
CC8D 2B        DEC HL
```

```
CC8E FE 21      CP #21
CC90 20 02      JR NZ,#CC94
CC92 18 F4      JR #CC88
```

This checks to see if all the Bleepload blocks have been loaded, and jumps ahead if they have, otherwise it jumps back to load the next block.

```
CC94 3E C3      LD A,#C3
CC96 32 32 5B   LD (#5B32),A
CC99 21 A1 CC   LD HL,CCA1
CC9B 22 33 5B   LD (#5B33),HL
CC9E C3 00 5B   JP #5B00
```

This puts the instruction JP #CCA1 at #5B32 so the loader decrypter will return to our hack at #CCA1 when finished.

```
CCA1 21 B2 CC LD HL,#CCB2
CCA4 11 D5 5B LD DE,#5BD5
CCA7 01 08 00   LD BC,#0008
CCAA ED B0      LDIR
```

This copies the final part of our hacking routine to #5BD5, where it will be executed once the whole game has been loaded.

```
CCAC 21 00 00   LD HL,#0000
CCAF C3 35 5B   JP #5B35
```

The LD HL,#0000 instruction is important, because it's the instruction we overwrote with out JP back to the hack. Therefore, we've got to execute it, otherwise the loading system may crash. Then it resumes loading at #5B35 with the POKEs firmly in place.

```
CCB2 3E B6      LD A,#B6
CCB4 32 5F AB   LD (#AB5F),A
CCB7 C3 BC F5   JP #F5BC
```

This is the hacking routine which will be copied into the loading system. AB5F,B6 is the POKE for infinite lives (which can be worked out by a forwards or a backwards trace), and JP #F5BC jumps to the game.

And that's about it for Bleepload! Hopefully, if you were hacking a different game, you still managed to do it (they're all virtually identical anyway).

ULTIMATE LOADER

Remember Ultimate? They were one of the finest software houses of all time. Most of their games from 1983 to 1987 had the same type of loader (but a few were Speedlocked - more about them later). On the face of it, it just looks like a totally unprotected BASIC loader, but the appearance is deceptive. The five blocks it loads are the loading screen, the game itself, a decrypter at #5B80, and two very short blocks of system variables. The system variables are, in actual fact the BASIC clock, and determine how many 50ths of a second the computer has been switched on for. The decrypter works using this system variable. The upshot of all this is that if you stop the program for even 1/50th of a second, you'll mess up the decrypter. You can get round this with a Multiface by loading in the first three blocks of code, then replacing the code at #5B80 with #F3,#18 and #FE. This disables interrupts, so the system variable

doesn't get updated, and causes an endless loop. Load in the last two blocks, activate the Multiface, and find out what the system variable should be. Then you can put this into the decrypter automatically.

MIKRO-GEN LOADER

This loading system appeared on just about every game released by the software house Mikro-Gen (oddly enough!) from about mid-84 to their demise in 1987. They come in two varieties, and you'll need a Multiface to hack some of the later ones, unfortunately. The first type are recognised by black and white loading stripes, which loads in a screen block, and then the main game block separately. I'll be doing Pyjamarama as an example, but any Mikro Gen game which fits the above description will do.

So the first thing to do is to *Hack the BASIC loader.

PYJAMARAMA LINE 0 LEN 504

```
0 BORDER 7:PAPER 7:INK 0:BRIGHT 0:FLASH 0:CLS:PRINT AT
12,12;"LOADING":RANDOMIZE USR (PEEK 23627+256*PEEK 23628+6)
20 POKE 23756,0:POKE 23757,0:SAVE "PYJAMARAMA" LINE 0:RANDOMIZE
USR 33040
```

The BASIC loader actually features much more than what we can list. If you're old enough to remember the ZX81, you'll recall that the best place to put a machine code program is in a REM statement. And that's almost the case here, except the machine code comes after the ASCII code #0D (newline), so you can't list it. But it's there. It's activated by the RANDOMIZE USR command. Type PRINT (PEEK 23627+256*PEEK 23628+6) and you'll find out the start address of the code. I made it 23984, which is 5DB0 hex (but you might find it to be something different), so disassemble this address.

```
5DB0 F3        DI
5DB1 31 00 00  LD SP,#0000
5DB4 2A 4B 5C  LD HL,(#5C4B)
5DB7 11 1C 00  LD DE,#001C
5DBA 19        ADD HL,DE
5DBB 11 16 80  LD DE,8016
5DBE 01 E7 00  LD BC,00E7
5DC1 ED B0     LDIR
5DC3 C3 16 80  JP 8016
```

Hopefully the DI and the LD SP,#0000 should be familiar. The next line loads HL with the two byte value starting at 5C4B. I made it 5DAA. This then has 1C added onto it, making it 5DC6. The rest of the code is a simple LDIR command, which puts the loading system to where it should be. In our hack, we can simply use a headerless loader to load the code into place. We know that 5DC6 goes to 8016. BASIC always
starts at the value in #5C53, which is #5CCB in this case. We know that the length is 504, or #1F8 hex bytes long, and the start address is (#5CCB-#5DC6)+8016 = #7F1B. So, run the following routine.

```
5B00 DD 21 1B 7F  LD IX,#7F1B
5B04 11 F8 01     LD DE,#01F8
5B07 3E FF        LD A,#FF
5B09 37           SCF
5B0A CD 56 05     CALL #0556
5B0D 30 F1        JR NC,#5B00
5B0F C9           RET
```

I've put a JR NC,#5B00 in, so that the computer ignores the BASIC header, and will only return on loading the main BASIC block. You should also note, that in the final hack, we'll have to add a DI and a LD SP,#0000 sometime. For now, disassemble #8016

```
8016 DD 21 00 40        LD IX,#4000
801A 11 01 1B           LD DE,#1B01
801D CD 4F 80           CALL #804F
```

This code activates the turboloader, which loads in the title screen.

```
8020 21 00 40    LD HL,#4000
8023 01 00 1B    LD BC,#1B00
8026 CD 3F 80    CALL #803F
```

This code verifies that the screen has loaded in properly (the routine at #803F adds up all the memory with start HL and length BC, and compares it with the byte after this block), and resets the computer if it hasn't.

```
8029 DD 21 00 82        LD IX,#8200
802D 11 A0 7A           LD DE,#7AA0
8030 CD 4F 80           CALL #804F
8033 21 00 82           LD HL,#8200
8036 01 9F 7A           LD BC,#7A9F
8039 CD 3F 80           CALL #803F
```

This is exactly the same as with the previous code, except it loads and checks the main game instead of the loading screen.

```
803C C3 89 FC           JP #FC89
```

Put a breakpoint over this instruction. Now POKE #8012 with F3, #8013 with #31, #8014 with #00 and #8015 with #00 (because we didn't execute the DI:LD SP,#0000 from the BASIC loader, and the game will not load otherwise), JP #8012 and start the tape. When the main game's loaded, disassemble #FC89.

```
FC89 21 EF B4   LD HL,#B4EF
FC8C 11 00 40   LD DE,#4000
FC8F 01 00 1B   LD BC,#1B00
FC92 1A         LD A,(DE)
FC93 AE         XOR (HL)
FC94 77         LD (HL),A
FC95 23         INC HL
FC96 13         INC DE
FC97 0B         DEC BC
FC98 78         LD A,B
FC99 B1         OR C
FC9A 20 F6      JR NZ,#FC92
FC9C C3 EA BE JP #BEEA
```

This decrypter uses values in the screen memory, so you'll have to put a breakpoint at FC9C, put a JP #FC89 at #8029, JP to #8012 and reload the loading screen before you can run it. Then disassemble #BEEA.

```
BEEA 31 00 00           LD SP,#0000
BEED CD CC BE           CALL #BECC
```

```
BEF0 C3 00 82        JP #8200
```

This code puts the stack pointer back at #0000, CALLs another decrypter, and JPs to #8200, which is the start of the game. Change the #8200 to a suitable place to put POKEs; finish them with a JP #8200 to start the game.

Here's the final hack, and I've put it at #5B00, because it doesn't get overloaded, apart from the byte at #5B00 itself, which is no longer needed by that time. Also, I've executed the DI:LD SP,#0000 directly, as well as the code from BEEA to BEF2.

```
5B00 DD 21 1B 7F     LD IX,#7F1B
5B04 11 F8 01        LD DE,#01F8
5B07 3E FF           LD A,#FF
5B09 37              SCF
5B0A CD 56 05        CALL #0556
5B0D 30 F1           JR NC,#5B00
5B0F 21 1C 5B        LD HL,#5B1C
5B12 22 3D 80        LD (#803D),HL
5B15 F3              DI
5B16 31 00 00        LD SP,#0000
5B19 C3 16 80        JP #8016
5B1C 21 25 5B        LD HL,#5B25
5B1F 22 9D FC        LD (#FC9D),HL
5B22 C3 89 FC        JP #FC89
5B25 31 00 00        LD SP,#0000
5B28 CD CC BE        CALL #BECC
5B2B AF              XOR A
5B2C 32 ?? ??        LD (????),A
5B2F C3 00 82        JP #8200
```

The other type of Mikro Gen loader is almost identical, except the whole game loads in one long block. Then end of the BASIC loading system is missing to start with, and is only loaded right at the end of the main headerless block. You can find out the missing code by loading the game as normal, then stopping it with a Multiface in the pause between the game loading, and the game starting (approx. 3 seconds), and hack it in the same way as Pyjamarama.

POWERLOAD

This protection system appeared first around the start of 1984, and was written by "Tag" (Phil Taglione) for Incentive Software. However, it's been used by quite a lot of other software companies as well, including Beyond, Mirrorsoft, Prism and Ariolasoft. It can be recognised by the screen turning black, accompanied by a few ascending beeps. It then loads one short headerless block, and then a longer headerless block, which includes the attribute file for the game coming up "backwards" i.e.: right to left, starting from the bottom. The game also stops loading just before the end of the long headerless block.
The only thing I know of that YS have put on the cover tape that has Powerload is the Graphic Adventure Creator, but that's pointless hacking, so instead I'll be hacking Dynamite Dan. Of course, most other Powerload games are identical apart from some addresses, and, in fact, the BASIC loaders are all identical.

Before we start, I need to explain a little more about the stack, because Powerload uses it a lot. There are four commands which use the stack, and they are:

PUSH X (where X is any register) - this takes the value in a register, and puts it onto the stack. The stack pointer then decreases by two (to be in the right place to store another value).

POP X - this takes the two byte value at the stack pointer (i.e.: the top of the stack), and puts them in a register. This also increases the stack pointer by two.

CALL XXXX - when you CALL a subroutine, the return address (i.e.: the address after the call) is PUSHed onto the stack, and the subroutine is JPed to. The stack pointer also decreases by two.

RET - when a RET instruction occurs, the computer takes the value on the top of the stack, and JPs to it. The stack pointer increases by two.

Now we've cleared that up, let's start hacking. *Hack the BASIC as usual.

D.D. LINE 0 LEN 496

```
0 REM
10 CLEAR 59999:POKE 23693,0:POKE 23624,0:POKE 23697,0:CLS:POKE
23659,0:FOR N=30 TO 36:BEEP .075,N:NEXT N:RANDOMIZE USR
24146:RANDOMIZE USR 0
100 REM
```

The POKEs in line 10 just make the screen black and prevent you from pressing break. 24146 is #5E52 hex; but a breakpoint at #5E52 and GOTO 0. This is because the stack is set up in a specific way by the BASIC commands.

```
5E52 F3 DI
5E53 21 00 00   LD HL,#0000
5E56 39         ADD HL,SP
5E57 22 F2 5D   LD (#5DF2),HL
```

This code simply puts the value of the stack pointer into address #5DF2, so it can be retrieved later.

```
5E5A 31 95 5E   LD SP,#5E95
5E5D 26 5E      LD H,#5E
5E5F E5         PUSH HL
5E60 21 68 5E   LD HL,#5E68
5E63 E9         JP (HL)

5E68 3E 12      LD A,#12
5E6A 32 93 53   LD (#5E93),A
5E6D E1         POP HL
5E6E E5         PUSH HL
5E6F D1         POP DE
5E70 C9         RET
```

Put a breakpoint at 5E70 and JP to 5E52. At 5E70, the value on the top of the stack is #5E76, so a RET will JP to there.

```
5E76 C1         POP BC
5E77 7E         LD A,(HL)
5E78 ED 44      NEG
5E7A 77         LD (HL),A
5E7B 23         INC HL
5E7C 10 F9      DJNZ #5E77
```

This code is, as you might realise, a decrypter. The start value of HL is #5E12, and the initial value of B is #3A. In case you're interested, the NEG instruction turns the value in the A register into its negative form; in other words, the value in A is subtracted from #100 hex. Put a breakpoint at #5E7E and JP #5E70 (which is where we left off).

```
5E7E E1        POP HL
5E7F 22 78 5E  LD (#5E78),HL
5E82 C1        POP BC
5E83 3E C9     LD A,#C9
5E85 32 7E 5E  LD (#5E7E),A
5E88 3E 00     LD A,#00
5E8A 32 7A 5E  LD (#5E7A),A
5E8D 5D        PUSH DE
5E8E E1        POP HL
5E8F C9        RET
```

This code changes the previous decrypter slightly, and RETs to 5E77. Put a breakpoint at 5E8F and JP #5E7E.

```
5E77 7E        LD A,(HL)
5E78 ED 67     RRD
5E7A 00        NOP
5E7B 23        INC HL
5E7C 10 F9     DJNZ,#5E77
5E7E C9        RET
```

This code works with the same values as the previous one; HL=5E12 and B=3A. It then RETs to 5E12. Put a breakpoint at #5E7E, and JP #5E8F (where we left off last time).

```
5E12 21 B4 5F  LD HL,#5FB4
5E15 11 B5 5F  LD DE,#5FB5
5E18 01 B8 88  LD BC,#88B8
5E1B ED B0     LDIR
5E1D E1        POP HL
5E1E 54        LD D,H
5E1F 5D        LD E,L
5E20 1C        INC E
5E21 C1        POP BC
5E22 ED B0     LDIR
```

These two LDIR commands wipe all the memory that isn't being used by the loading system. To get round this, you should change #5E1B, #5E1C, #5E22 and #5E23 to #00, to stop them being executed. Put a breakpoint at #5E24 and JP #5E7E (where we left off).

```
5E24 06 1E     LD B,#1E
5E26 E1        POP HL
5E27 7E        LD A,(HL)
5E28 EE A3     XOR #A3
5E2A 77        LD (HL),A
5E2B 23        INC HL
5E2C 10 F9     DJNZ,5E27
```

The value in HL for this decrypter is #5E2E, which is right after the decrypter. To crack it, therefore, move the code from #5E24 to #5E2D somewhere safe (such as #5B00), put a breakpoint on the end, and

run the code from there. When that's done, put a breakpoint at #5E2E and JP to #5E2E (so that you're back in the right place in the loading system). Carrying on with the loader....

```
5E2E E1        POP HL
5E2F 22 02 5E  LD (#5E02),HL
5E32 E1        POP HL
5E33 22 05 5E  LD (#5E05),HL
5E36 37        SCF
5E37 3E 07     LD A,#07
5E39 CD 00 5E  CALL #5E00
```

This code takes some values off the stack, and puts them into the subroutine at #5E00, which is then CALLed. Put a breakpoint at #5E39 and JP to #5E2E.

```
5E00 DD 21 40 9C  LD IX,#9C40
5E04 11 90 1      LD DE,#190
5E07 14           INC D
5E08 08           EX AF,AF'
5E09 15           DEC D
5E0A 3E 0F        LD A,#0F
5E0C DB FE        OUT (#FE),A
5E0E CD 62 05     CALL #0562
5E11 C9           RET
```

This routine is a headerless loader. The start is #9C40 and the length is #190. Also the value of A is 7, and the carry flag has been set. In effect, we could have used a standard CALL #0556 headerless loader. Put a breakpoint at #5E3C and JP to #5E39. Start the tape and load in the first short headerless block. Then continue disassembling.

```
5E3C D2 01 00     JP NC,#0001
```

This code resets the computer if there was a loading error from the first headerless block.

```
5E3F 21 40 9C  LD HL,#9C40
5E42 06 FF     LD B,#FF
5E44 CD 77 5E  CALL #5E77
5E47 06 FF     LD B,#FF
5E49 CD 77 5E  CALL #5E77
5E4C F3        DI
5E4D C9        RET
```

This code decrypts the headerless file we have just loaded in. The decrypter is CALLed to, and it's the same one we had before (with the RRD). So, in actual fact, we can forget about the BASIC loader; just load in the headerless file normally and run our own decrypter. For now, just put a breakpoint at #5E4D and JP to #5E3F. The RET is to #9C40

```
9C40 21 52 9C  LD HL,#9C52
9C43 01 90 01  LD BC,#0190
9C46 16 A5     LD D,#A5
9C48 7E        LD A,(HL)
9C49 AA        XOR D
9C4A 77        LD (HL),A
9C4B 23        INC HL
9C4C 0B        DEC BC
```

```
9C4D 78          LD A,B
9C4E B1          OR C
9C4F C2 48 9C    JP NZ,9C48
```

This is a decrypter, and you can crack it in one of two ways. Firstly, you can copy if to somewhere else, put a breakpoint on the end, and run it from there, or you can replace the JP NZ,9C48 with JR NZ,9C48. Both do the same thing, but the JR NZ only uses two bytes. This means we can put a RET at #9C51 and CALL the decrypter. So change #9C4F to #20, #9C50 to #F7 and #9C51 to #C9, then put a CALL #9C40 and a breakpoint somewhere convenient (such as #5B00) and run the decrypter from there. When decrypted, the code continues at #9C52.

```
9C52 21 63 9C    LD HL,#9C63
9C55 11 45 FE    LD DE,#FE45
9C58 01 90 01    LD BC,#0190
9C5B ED B0       LDIR
9C5D 31 84 FD    LD SP,#FD84
9C60 C3 45 FE    JP #FE45
```

This moves the code we've just decrypted to #FE45, sets the stack pointer to #FD84, and JPs to #FE45. Put a breakpoint at #9C60 and JP to #9C52.

Now at #FE45, we come to the actual loading system itself.

```
FE45 3E 84          LD A,#84
FE47 11 00 18       LD DE,#1800
FE4A DD 21 00 40    LD IX,#4000
FE4E CD AB FE       CALL #FEAB
```

This code loads in a headerless file with start #4000 and length #1800 (which is the display file for the screen). Nothing too unusual about that.....

```
FE51 11 00 04       LD DE,#0400
FE54 DD 21 FF 5B    LD IX,#5BFF
FE58 CD 2A FF       CALL #FF2A
```

....except that as soon as it's done that, it loads in a block with start #5BFF and length #0400 straight away. This block is "sandwiched" right next to the other block on the tape. This particular block loads "backwards".

```
FE5B 11 E1 01       LD DE,#01E1
FE5E DD 21 1F FE    LD IX,#FE1F
FE62 CD FA FA       CALL #FEFA
```

And here's another block of the same kind, except it's loaded forwards this time. What's worse is that it's going to overwrite the code we're looking at now, so it must be a "modification" to the loading system (similar to the Mikro-Gen loader). To get round this, we would have to copy the code somewhere else, stick a breakpoint on the end, and run it from there. But remember that the loading system was copied from address #9C63, so there is, in actual fact, a copy of the code anyway. You want to put a breakpoint at #9C83 (the instruction after loading these three blocks), and JP to #9C63. Then start the tape and load in the next headerless block. The loading screen will appear, and the game will load for about four seconds, then control will return to the disassembler. Now look at the code at #FE65.

```
FE65 11 1D 9F       LD DE,#9F1D
FE68 DD 21 1C FA    LD IX,#FA1C
```

```
FE6C CD 2A FF        CALL #FF2A
```

This code loads the main game backwards. This will overwrite your disassembler, so you will have to put the NEW routine at #FE6F (but write down all the bytes you are replacing, because you'll need to restore all the original code later). Then you will have to go back and load the first block, because there isn't a header for the main game block. Change #FE4D to #01, #FE57 to #10 and #FE61 to #01 - this will make the computer try to load the three blocks into the ROM. Then rewind the tape back to the start of this headerless block, JP #FE45, and start the tape. Upon loading the whole block, the computer will reset. Load in your disassembler, and replace the code from the NEW routine to the values they should be. Now you can tackle the final part of the loading system.

```
FE6F 11 E4 12        LD DE,#12E4
FE72 DD 21 FF FF     LD IX,#FFFF
FE76 C3 70 FF        JP #FF70

FF70 3E 00     LD A,0
FF72 D3 FE     OUT (#FE),A
FF74 CD 1F FE  CALL #FE1F
FF77 21 43 FE  LD HL,#FE43
FF7A BE        CP (HL)
FF7B CA 89 FF  JP Z,#FF89
FF7E 21 48 EE  LD HL,#EE48
FF81 01 FF FF  LD BC,#FFFF
FF84 11 49 EE  LD DE,#EE49
FF87 ED B0     LDIR
```

The routine at #FE1F adds up all the memory in the screen to get a value in the D register. This is then compared with the value of the byte at #FE43. If there is no match, all the memory is blanked out, so the value in the D register must be the same as the byte at #FE43. You should find that the byte is #E6. You need to know this for later on.

```
FF89 21 A3 FF  LD HL,#FFA3
FF8C 01 45 00  LD BC,#0045
FF8F 7A        LD A,D
FF90 AE        XOR (HL)
FF92 23        INC HL
FF93 0B        DEC BC
FF94 78        LD A,B
FF95 B1        OR C
FF96 C2 8F FF  JP NZ,#FF8F
```

This is all we can disassemble for the moment, because the code from #FF89 to #FF98 decrypts the final part of the loader. Change the byte at #FF87 to #16, and the byte at #FF88 to #E6 (this is LD D,#E6, which is used in the decrypter), put a breakpoint at #FF99, and JP to #FF87. Then continue the disassembly.

```
FF99 CD 31 FE  CALL #FE31
FF9C 21 44 FE  LD HL,#FE44
FF9F BE        CP (HL)
FFA0 C2 7E FF  JP NZ,#FF7E
```

This code checks the main game, coming out with a result in the E register. However, this value is never used, so you can ignore this whole routine. Following on.....

```
FFA3 21 C0 5D  LD HL,#5DC0
FFA6 01 30 75  LD BC,#7530
FFA9 CD D4 FF  CALL #FFD4
FFAC 21 C0 5D  LD HL,#5DC0
FFAF 01 30 75  LD BC,#7530
FFB2 CD DE FF  CALL #FFDE
FFB5 21 1C FA  LD HL,#FA1C
FFB8 11 1C FF  LD DE,#FF1C
FFBB 01 1D 9F  LD BC,#9F1D
FFBE ED B8     LDIR
FFC0 21 10 A7  LD HL,#A710
FFC3 22 36 5C  LD (#5C36),HL
FFC6 01 10 DF  LD BC,#DF10
FFC9 AF        XOR A
FFCA ED 42     SBC HL,BC
FFCC 31 FF FF  LD SP,FFFF
FFCF ED 56     IM1
FFD1 C3 6F 00  JP #006F

006F E9 JP (HL)
```

This routine runs the game decrypters, moves the game into the right place, sets the stack and the interrupts, and puts the start address for the game in the HL register. Change the #006F at #FFD2 to somewhere where you can put a NEW routine (such as #5B00), because the disassembler will be overwritten. Then, change the value at #FFA1 to 16, and the value at #FFA2 to #E6 (because one decrypter uses the value in the D register), and JP to #FFA2. When that's done, you can reload your disassembler, and hack the game using a forwards and backwards trace (but you won't be able to run the code because some of it's missing!)

Now we'll write a complete hack for the game. You have to be a bit careful about where you put your hack in memory, because a lot of memory is overloaded. The first free address we can put the code at is #FA1D.

```
FA1D DD 21 40 9C  LD IX,#9C40
FA21 11 90 01     LD DE,#0190
FA24 3E 07        LD A,#07
FA26 37           SCF
FA27 CD 56 05     CALL #0556
FA2A 30 F1        JR NC,#FA1D
```

This loads in the first headerless block using the values set up in the BASIC loader.

```
FA2C 06 FF     LD B,#FF
FA2E 21 40 9C  LD HL,#9C40
FA31 7E        LD A,(HL)
FA32 ED 67     RRD
FA34 23        INC HL
FA35 10 FA     DJNZ #FA31
FA37 06 FF     LD B,#FF
FA39 7E        LD A,(HL)
FA3A ED 67     RRD
FA3C 23        INC HL
FA3D 10 FA     DJNZ, #FA39
```

This decrypts the headerless block.

```
FA3F 21 20 F7   LD HL,#F720
FA42 22 4F 9C   LD (#9C4F),HL
FA45 3E C9      LD A,#C9
FA46 32 51 9C   LD (#9C51),A
FA49 CD 40 9C   CALL #9C40
```

This changes the JP NZ at #9C4F to a JR NZ and a RET, then calls the decrypter.

```
FA4C 3E C3      LD A,#C3
FA4E 32 83 9C   LD (#9C83),A
FA51 21 5A FA   LD HL,#FA5A
FA54 22 84 9C   LD (#9C84),HL
FA57 C3 63 9C   JP #9C63
```

This puts a JP back to our hack at #9C83, and jumps to #9C63 to load the first part of the headerless block.

```
FA5A 3E C9      LD A,#C9
FA5C 32 6F FE   LD (#FE6F),A
FA5F CD 65 FE   CALL #FE65
```

This puts a RET after the code to load the rest of the game, then CALLs that loading procedure.

```
FA62 21 E6 16   LD HL,#16E6
FA65 22 87 FF   LD (#FF87),HL
FA68 3E C9      LD A,#C9
FA6A 32 99 FF   LD (#FF99),A
FA6D CD 87 FF   CALL #FF87
```

This patches in the LD D,#E6, puts a RET at the end of the decrypter, and CALLs it.

```
FA70 21 7E FA   LD HL,#FA7E
FA73 11 00 40   LD DE,#4000
FA76 01 20 00   LD BC,#0020
FA79 ED B0      LDIR
FA7B C3 00 40   JP #4000
```

This code moves our hack into the screen memory (so it isn't affected by the LDIR which overwrites it in the next bit of code), and jumps to it there.

```
FA7E 21 0B 40   LD HL,#400B
FA81 22 D2 FF   LD (#FFD2),HL
FA84 16 E6      LD D,#E6
FA86 C3 A3 FF   JP #FFA3
```

This code replaces the JP #006F with a JP back to our hack (which is in the screen memory by this time), and JPs to #FFA3. We have to include the LD D,#E6 again, because the value of DE was corrupted by the LDIR.

```
FA89 AF         XOR A
FA8A 32 C6 CD   LD (#CDC6),A
FA8D E9         JP (HL)
```

This sets the infinite lives POKE, and does a JP (HL) to start the game.

Phew! I hope you managed to get all that, because it's really hard to do without a Multiface. If you can do it, then you've definitely got the hang of things, so keep it up!

SEARCH LOADER

This loading system appears on every game ever written by Steve Marsden (who wrote the original loading system), as well as a few others. You can recognise them by their fancy front end, which consists of a countdown timer, accompanied by animated graphics and/or instructions, which appear as the game loads. The only game I've actually got at the moment that's got a Search Loader on it is Technician Ted, so I'm going to have to hack that.

So, *Hack the BASIC loader, and let's see what it's got to offer....

Chip Fact LINE 0 LEN 736

0 RANDOMIZE USR 24341

The rest of the BASIC is a load of garbage which consists of the machine code for the game. It is stored in a similar way to that in the Mikro Gen loader. 24341 is 5F15 hex, so disassemble this address.

```
5F15 F3        DI
5F16 21 00 40  LD HL,#4000
5F19 11 01 40  LD DE,#4001
5F1C 01 FF 17  LD BC,#17FF
5F1F 36 00     LD (HL),0
5F21 ED B0     LDIR
```

This code blanks out the screen.

```
5F23 CD 32 5E  CALL #5E32
```

The routine at #5E32 sets up the attributes for the screen i.e.: red banners at the top and bottom, black background with varying ink colours in the middle.

```
5F26 C3 38 5F  JP #5F38

5F38 21 AB 5F  LD HL,#5FAB
5F3B 01 59 A0  LD BC,#A059
5F3E 31 00 5C  LD SP,#5C00
5F41 3A A8 5F  LD A,(#5FA8)
5F44 57        LD D,A
5F45 1E 0B     LD E,#0B
5F47 7A        LD A,D
5F48 87        ADD A,A
5F49 87        ADD A,A
5F4A 87        ADD A,A
5F4B 87        ADD A,A
5F4C 82        ADD A,D
5F4D 83        ADD A,E
5F4E 57        LD D,A
5F4F 77        LD (HL),A
5F50 23        INC HL
```

```
5F51 0B       DEC BC
5F52 78       LD A,B
5F53 B1       OR C
5F54 20 F1    JR NZ,#5F47
```

This routine fills all of the memory above #5FAB with unexecutable code. It is, however, extremely important code, as we shall see later on.

```
5F56 CD 93 5F   CALL #5F93
```

This routine at #5F93 just messes around with the garbage a bit more.

```
5F59 CD 80 5D   CALL #5D80
```

The routine at #5D80 scrolls in the title messages for the game, accompanied by annoying clicks.

```
5F5C 3A 66 80   LD A,(#8066)
5F5F 6F         LD L,A
5F60 3A E6 60   LD A,(#60E6)
5F63 67         LD H,A
5F64 E5         PUSH HL
5F65 3A 4F FC   LD A,(#FC4F)
5F68 5F         LD E,A
5F69 3A 0F 60   LD A,(#600F)
5F6C 57         LD D,A
5F6D DD E1      POP IX
5F6F 37         SCF
5F70 3E FF      LD A,#FF
5F72 14         INC D
5F73 08         EX AF,AF'
5F74 15         DEC D
5F75 3A 66 63   LD A,(#6366)
5F78 6F         LD L,A
5F79 3A E6 63   LD A,(#63E6)
5F7C 67         LD H,A
5F7D E5         PUSH HL
5F7E DB FE      OUT (#FE),A
5F80 1F         RRA
5F81 E6 20      AND #20
5F83 F6 01      OR #01
5F85 4F         LD C,A
5F86 BF         CP A
5F87 F5         PUSH AF
5F88 3A 87 65   LD A,(#6587)
5F8B 6F         LD L,A
5F8C 3A 85 64   LD A,(#6485)
5F8F 67         LD H,A
5F90 F1         POP AF
5F91 E5         PUSH HL
5F92 C9         RET
```

This code takes values out of the garbage and puts them in certain registers. It then imitates the start of the ROM loading routine, and puts some values on the stack. At #5F92, the values of the registers are: Hl= #056B, DE=#03C3, IX=#8000, and the values on the stack are first #056B, then #8000. So, this code

will load a headerless file with start #8000 and length #0363, then will JP straight to #8000. We can do away with the BASIC loader altogether in the final hack by mimicking the headerless loader. This is done using the following program.

```
5B00 F3              DI
5B01 31 00 5C        LD SP,#5C00
5B04 DD 21 00 80     LD IX,#8000
5B08 11 C3 03        LD DE,#03C3
5B0B 3E FF           LD A,#FF
5B0D 37              SCF
5B0E 14              INC D
5B0F 08              EX AF,AF'
5B10 15              DEC D
5B11 AF              XOR A
5B12 DB FE           OUT (#FE),A
5B14 1F              RRA
5B15 E6 20           AND #20
5B17 F6 01           OR #01
5B19 4F              LD C,A
5B1A CD 6B 05        CALL #056B
5B1D <breakpoint>
```

This routine is slightly different than the one in the loader for two reasons. Firstly, I've put values into the registers directly, rather than have their values taken from bytes in memory. Secondly, you aren't allowed by law to rip off someone else's code; if you directly copied a loading system into a hack, you could be sued. In fact, someone was, once! You're probably alright copying a five byte decrypter from Powerload across, because there really isn't any other code which can do the job in the same way. In general, I would say don't copy code into your hack unless you have to. If you do, change it if you can so it does the same job in a different way. Copying 40 bytes of code directly out of a loading system is definitely out, and most magazines wouldn't print the routine anyway.

There are a few commands we haven't met in the routine. EX AF', AF' concerns the swapping of registers. In the Z80, in actual fact, there are two different sets of each register, although only one set can ever be used at once. Think of it like a TV set, although both registers (A and A' in this case) are there, you
can only see one at a time. EX AF,AF' exchanges both the A register and the contents of the flags. Don't worry any more about swapping registers for now. RRA rotates all the bits in the A register to the right. Actually, it doesn't quite do this, but we don't need to know about it. If you're not using a Multiface, the garbage routine will have overwritten the disassembler, so reload it in anywhere below #8000. Then run the routine and restart the tape from where you left off. A small part of the headerless block will load in, and control will return to the disassembler. Have a look at address #8000.

```
8000 D2 00 00    JP NC,#0000
```

This resets the computer if the previous headerless block didn't load properly.

```
8003 3E 08       LD A,#08
8005 D3 FE       OUT (#FE),A
```

This makes the border black, and sends a signal to the cassette recorder.

```
8007 D9EXX
```

EXX is a "general exchange" instruction, and changes the registers B,C,D,E,H and L for their alternate sets.

```
800E 0E 00          LD C,#00
800A D9             EXX
800B 26 00          LD H,#00
800D 06 80          LD B,#80
800F DD 21 1C 8C    LD IX,#8C1C
8013 16 05          LD D,#05
8015 CD 41 83       CALL #8341
8018 D2 00 00       JP NC,#0000
801B 06 B1          LD B,#B1
801D 15             DEC D
801E 20 F5          JR NZ,#8015
```

This code loads in five bytes of tape (the routine at #8341 loads in information off tape into the address pointed to by the IX register), starting at #8C1C. Therefore, this tape routine will start loading code at #8C1C.

```
8020 11 D8 72   LD DE,#72D8
8023 D9         EXX
8024 0C         INC C
8025 D9         EXX
8026 C3 7D 83   JP #837D

837D 2E 01      LD L,#01
837F CD 41 83   CALL #8341
8382 D2 00 00   JP NC,#0000
8385 3E CB      LD A,#CB
8387 B8         CP B
8388 17         RLA
8389 D9         EXX
838A 47         LD B,A
838B E6 01      AND #01
838D 3D         INC A
838F 11 6B 80   LD DE,#806B
8392 1A         LD A,(DE)
8393 A7         AND A
8394 C4 77 82   CALL NZ,#8277
```

This loads in a set number of bytes from tape, and then prints some sprites on screen (the routine at #8277). This produces all the men walking forwards and backwards while the game loads.

```
8397 CB 18      RRB
8399 D9         EXX
839A CB 15      RL L
839C 06 B1      LD B,#B1
839E 30 DF      JR NC,#837F
83A0 7C         LD A,H
83A1 AD         XOR L
83A2 67         LD H,A
83A3 7A         LD A,D
83A4 B3         OR E
83A5 20 CE      JR NZ,#8375
```

This updates the computer ready to do the next loading and animation sequence.

```
83A7 7C        LD A,H
83A8 A7        AND A
83A9 C2 00 00  JP NZ,#0000
83AC 11 2F EC  LD DE,#EC2F
83AF 06 EB     LD B,#EB
83B1 CD 41 83  CALL #8431
83B4 D2 00 00  JP NC,#0000
83B7 3E EA     LD A,#EA
83B9 B8        CO #0B
83BA D2 00 00  JP NC,#0000
83BD 42        LD B,D
83BE 15        DEC D
83BF 1D        DEC E
83C0 C2 B1 83  JP NZ,#83B1
```

This loads in some more bytes from tape. When these have finished, the computer will continue execution at address #83C3. The code beyond this address does nothing except fiddle about with registers, so there might as well be nothing there. The first bit of useful code will appear at #8C1C, but this hasn't been loaded yet. Instead, put a breakpoint at #83C3, move the headerless loading routine so the CALL #056B is at #7FFD, and run the code from there. Rewind the tape a bit and reload in all of the main headerless block. When finished, you'll find the following code at #8C1C.

```
8C1C 3E 5C     LD A,#5C
8C1E 21 00 40  LD HL,#4000
8C21 54        LD D,H
8C22 5D        LD E,L
8C23 EB        EX DE,HL
8C24 4E        LD C,(HL)
8C25 23        INC HL
8C26 46        LD B,(HL)
8C27 23        INC HL
8C28 EB        EX DE,HL
8C29 09        ADD HL,BC
8C2A BA        CP D
8C2B 20 F6     JR NZ,8C23
8C2D 11 92 5C  LD DE,#5C92
8C30 EE 5C     XOR 5C
8C32 28 EF     JR Z,#8C23
8C34 E5        PUSH HL
```

This routine adds up every single byte in memory to get a value in HL, which is pushed onto the stack. This value is important, because it is used in decrypting. This is what the garbage was all used for - to get the right value. It's impossible to calculate it yourself, because any programs you right will mean you get the wrong answer. The only way I can think of to get this value is to load the game as normal, and stop the game with a Multiface as soon as the timer hits 000. Then put a breakpoint at #8C35 and return. Wait a few seconds, then reactivate the Multiface and have a look at the stack. The first value will be #8C35, the second will be the value of HL you want. You should find it's #4DBD.

Carrying on the disassembly....

```
8C35 21 00 58  LD HL,#5800
8C38 11 01 58  LD DE,#5801
```

```
8C3B 01 FF 02   LD BC,#02FF
8C3E 36 00      LD (HL),#00
8C40 ED B0      LDIR
8C42 E1         POP HL
```

This clears the screen and restores the value of HL, which is used for the following decrypter.

```
8C43 11 60 8C   LD DE,#8C60
8C46 0E 29      LD C,#29
8C48 7C         LD A,H
8C49 65         LD H,L
8C4A 47         LD B,A
8C4B 09         ADD HL,BC
8C4C 1A         LD A,(DE)
8C4D AD         XOR L
8C4E 12         LD (DE),A
8C4F 6F         LD L,A
8C50 13         INC DE
8C51 1A         LD A,(DE)
8C52 AC         XOR H
8C53 12         LD (DE),A
8C54 13         INC DE
8C55 CB 7A      BIT 7,D
8C57 20 F0      JR NZ,#8C49
8C59 FB         EI
8C5A 67         LD H,A
8C5B 11 70 71   LD DE,#7170
8C5E 19         ADD HL,DE
8C5F E9         JP (HL)
```

First of all, POKE #8C40, #8C41 and #8C42 with #21, #BD and #4D respectively (so you get the right value of HL), put a breakpoint at #8C5B (nearest place possible to the JP (HL) that we can place a breakpoint), and JP #8C40. On return, the value of HL is #38F5. Add this to #7170 (which is what happens in the next two commands) to get #AA65. This is the start address of the game. So put a JP to your POKEing routine (anywhere from #5B00 to #5BA0 is fine) at #8C5B, and finish your POKEs with a JP #AA65. You will have to do a stack trace to find infinite lives in the actual game itself. There is a complete hack for this game by myself in YS #78, so why not disassemble it and have a look. It's slightly different to what we've done in that it intercepts the RET at the end of the loading system rather than mimic the first headerless loader, and puts a JP back to the hack at #83C3, but apart from that, it's more or less everything we've discussed above put together.

PAUL OWEN'S PROTECTION SYSTEM

This has been used on a few Ocean games, but is in fact a standard headerless loader in disguise. The value of A to use is always #98. Load in the BASIC loader and the first block of code, then stop it with a Multiface, and use a stack trace to find out the values of IX and DE for each block, and the JP to the game.

SPEEDLOCK

Concluding the look at protection systems, I think it's only fitting that we end in quite possible the most famous protection system of all time. Speedlock was first written by two guys called David Looker and David Aubery Jones around late 1983, although it wasn't commercially used until October 1984, on Daley Thompson's Decathlon, by which time it had reached it's third version. Since then, it has been used by

many major software companies, especially Ocean. Its also gone many modifications, and can be split into three distinct generations. I should state at this point that you need to have a Multiface to crack most of these Speedlocks, because they completely disrupt the operating system which will lock up any disassembler which relies on ROM routines. The Multiface relies on its own ROM, which isn't affected by the Speedlock code.

Type 1 - have one or two BASIC loaders, and load the main code with the infamous "clicking" leader tones (you know, instead of a steady "bleep", they go "blip, blip, blip, blip" a few times).

Type 2 - have one short BASIC loader, a long CODE block, lots of annoying beeps, then a similar loader to Type 1, minus the clicking leader tones, plus a countdown timer.

Type 3 - as for Type 2, except there is just one very long BASIC loader. The protection system crashes if a Multiface is left switched on. Mazemania on YS #77 covertape used a Type 3.

So, let's start at the very beginning (a very good place to start) with Type 1. In fact, there are about four different difficulty levels of Type 1 Speedlocks; the difficulty goes in chronological order (as you might expect).

The very first Type Ones were completely different to later ones, having the same initialisation routine, but a completely standard decrypter. The only differences between this Speedlock and an ordinary decrypting loader were its initialisation routine and its use of the IY register.

We came across index registers when we first met headerless loaders. There are, in fact, two index registers, IX and IY. In BASIC, the IX register is free for use in a machine code program run from a USR command, but the IY register must always contain the value #5C3A, which is the base address of the BASIC system variables which are wiped with a NEW command. If you return to BASIC with the value of IY anything other than #5C3A, the computer will crash, even if you use the "exit to BASIC" feature on a Multiface. The value of IY must also be #5C3A whenever a BASIC interrupt occurs. Both Devpac and 007 Disassembler run under the BASIC interrupts. They also use built in ROM routines, such as those to check the keyboard and print text; this is preferable, otherwise they'd have to waste memory rewriting their own versions of the routines. Hence the value of IY must always equal #5C3A.

The only safe way of using the IY register is to disable interrupts and write the whole program in RAM without using any built in ROM routines. And Speedlock fits this bill exactly, so it uses the IY register for most of its decrypter calculations. Speedlock code also uses a lot of undocumented instructions. In theory, you cannot split the sixteen bit IY register into two eight bit registers. But the processor doesn't understand this, and you can split the IY register into two if you want. You simply put the code #FD on the front of any instructions using H or L. There are no standard names for the two halves of the IY register, but I will refer to them as IYH (Hi part of IY) and IYL (Lo part of IY).

Now let's hack a Speedlock game. To start with, I'm hacking Knight Lore, but the following games are also suitable: Beach Head, Daley Thompson's Decathlon, Gilligan's Gold and Underwurlde. Anything released after these will be explained later.

So first *Hack the BASIC loader.

KNIGHT LINE 0 LEN 1037

```
0 BEEP 0.1,1:BEEP 0.1,2:BEEP 0.1,3:BEEP 0.1,4:BEEP 0.1,5:PAPER
0:BORDER 0:INK 0:BRIGHT 1:CLS:PRINT BRIGHT 1;INK 0;AT 9,5;
"LOADING: KNIGHT LORE";AT 12,10;"PLEASE WAIT"
0 POKE (PEEK 23641+256*PEEK 23642),PEEK 23649:POKE (PEEK 23641+ 256*PEEK
23642)+1,PEEK 23650
```

```
0 POKE (PEEK 23613+256*PEEK 23614),PEEK 23627:POKE (PEEK 23613+ 256*PEEK
23614)+1,PEEK 23628
0 POKE 23662,PEEK 23618:POKE 23663,PEEK 23619:POKE 23664,PEEK 23621
23676 "REM CLOSE #ATTR....
```

Wait a minute, there's absolutely no sign of a RANDOMIZE USR command anywhere! There's just some BASIC which beeps a bit, sets the colours, and prints a message, a whole load of POKEs, and then a load of garbage. Surely the computer will do everything, then report with an error message as soon as it reaches 23676? Well, that's not actually the case. Look at the third line 0 (the one which starts POKE [PEEK 23613+256* etc.). This system variable is known as ERR SP. What happens is that when an error occurs (and it will do here), the computer jumps to the value in this register. This value is PEEK 23627+256*PEEK 23628. PRINT this value, and there's the start of the machine code. You might get a different result to me, but I made the start address #60A8. Disassemble this address.

```
60A8 F3DI
60A9 FD 25      DEC IYH
60AB FD 7C      LD A,IYH
60AD FD AD      XOR IYL
```

The DI is very important, because otherwise the I register can't be used. Given that IY starts off as being #5C3A, the value in A will end up being #5B XORed with #3A, which is #61.

```
60AF FD 26 F3   LD IYH,#F3
60B2 FD 2E A6   LD IYL,#A6
60B5 3B         DEC SP
60B6 3B         DEC SP
60B7 01 54 FE   LD BC,#FE54
60BA FD E3      EX (SP),IY
```

First of all, this loads IY with the value #F3A6. It then decreases the stack pointer by two. By doing this, the stack pointer is now pointing to the start address of the machine code, which is #60A8. EX (SP),IY is a variation to the register exchange commands we've already come across. It basically swaps the value in the address pointed by the stack pointer with the value in the IY register. So, after this instruction, IY will contain #60A8, and the value on the top of the stack will be #F3A6.

```
60BC 21 30 F2   LD HL,#F230
60BF FD 09      ADD IY,BC
60C1 01 AC 01   LD BC,#01AC
60C4 FD 5D      LD E,IYL
60C6 FD 54      LD D,IYH
60C8 EB         EX DE,HL
```

Here, HL is being loaded with #F230. The value in BC (#FE54) is added to the value in IY (#60A8), making the value in IY #5EFC. Then BC is loaded with #01AC, and the value in IY is transferred into DE. Then the values of DE and HL are swapped. So, by the end of the code we've looked at so far, HL will equal #5EFC, DE will equal #F230, BC will equal #01AC, and A will equal #61. These values are all used in the decrypter which follows.

```
60C9 AE         XOR (HL)
60CA 12         LD (DE),A
60CB 7E         LD (HL),A
60CC 23         INC HL
60CD 13         INC DE
60CE 0B         DEC BC
```

```
60CF FD 6F      LD IYL,A
60D1 78         LD A,B
60D2 B1         OR C
60D3 FD 7D      LD A,IYL
60D5 20 F2      JR NZ,#60C9
60D7 C9         RET
```

This is a straightforward decrypter, except the value for A (which is needed throughout the decrypter) is temporarily stored in part of the IY register. The RET is to #F3A6 (the top value on the stack). To crack this, we can set up the register values manually, CALL the decrypter, and then hack the main loader ourselves. Type out this program:

```
5B00 F3             DI
5B01 21 FC 5E       LD HL,#5EFC
5B04 11 30 F2       LD DE,#F230
5B07 01 AC 01       LD BC,#01AC
5B0A 3E 61          LD A,#61
5B0C CD C9 60       CALL #60C9
5B0F FD 21 3A 5C    LD IY,#5C3A
5B13 <breakpoint>
```

Notice that we've disabled interrupts to avoid crashing, and we need to restore the value of IY to #5C3A afterwards, so your disassembler won't crash. RUN the program, and have a look at the code at #F3A6. You'll find it's just a straightforward headerless loader with absolutely no frills, and you should be able to hack it no problem. As for the final hack, load the BASIC into address #5CCB, run the decryption routine above, patch the JP in the main turboloader to your POKEs, and start running.

All other Speedlock Type 1s have the same sort of decrypter. The code for the decrypter is very complicated, with the result that I have been unable to reproduce it here. Luckily, you don't have to touch the code; you can write your own decrypter as long as you have a Multiface.

I'll be doing Tapper as an example, but any other Speedlock follows this procedure almost exactly. *Hack your game and note down the length of the code (you'll need it later). I made it 1453, which is #05AD hex. Now look at address #5EFD. The byte at #5EFD is always decrypted to give the byte #42, and the byte at
#5EFD is always decrypted to give the byte #55. The decrypter works by XORing the encrypted byte with a number taken from the R register. By inspecting the code before and after running, you'll see the XORing number starts off as #CB, and increases by #0A each time. If the result is more than #FF, the result has #80 subtracted from it. We can incorporate this into our decrypter. The start of the code is #5EFD, and the length is (PEEK 23627+ 256*PEEK 23628)-#5EFD, which is #01ED in the case of Tapper. The following code will simulate the decrypter.

```
   LD HL,<start>
   LD BC,<length>
   LD D,<initial decrypter value>
*** LD A,(HL)
   XOR D
   LD (HL),A
   LD A,D
   ADD #0A
   SET 7,A
   LD D,A
   INC HL
   DEC BC
```

JR NZ, ***

Once you've done that decrypter, you've got to do the whole lot again, starting at #5EFD. The byte there will be decrypted to either #3E or #ED - you'll have to guess which decrypting value to use. For Tapper, the start is #5F2B, the length is #1BF, and the second decrypter value is #AB. When you've done that, you'll either get the complete loading system or another decrypter. If you've got the loading system, then reload the BASIC loader, and do a stack trace to find out where it should go. You should have no problems with the loader. If you've got another decrypter, go along five bytes and find a LD DE,(XXXX). Add #2E to this value, and that's where you move it to. The length is the same as that for the second decrypter. The decrypter itself can be cracked by changing a JP Z in the code about forty bytes later (the value of this is the start of the turboloader), but the decrypter itself uses a byte which is worked out by adding all the memory together in the loading system. Since we've got an exact copy of this system elsewhere in memory, just change the value of XXXX in the aforementioned LD DE,(XXXX), and then JP to the start of the decrypter. If the first decrypting value you used was #CB, then you can just change the JP in the turboloader to your POKEs.

If the value was #CD, then you'll need to know about the Standard Speedlock patch. Somewhere in the loading system there will be the two bytes ED 53 [LD DE,(XXXX)]. Change the XXXX to the address of your POKEs (#5BA0 is normally safe), and end your POKEs with a JP to the value you overwrote. You'll have to use this patch for the later Speedlocks as well.

There was a Speedlock Type 1 MultiPOKE in YS#79. RUN the program, press BREAK and disassemble address #5B00 to find out what to do in your own hacks.

Type 2 Speedlocks feature a very easy BASIC loader, and one big block of code, which has six short decrypters and a complex moving routine. The decrypters are all easy peasy, just move them to somewhere else in memory (such as #5B00), bung a RET on the end, and CALL the decrypter from there (but watch out for the third decrypter, which checks for a Multiface and crashes if it finds it. The moving routine fiddles about with the loader. Search for 31, which means LD SP,XXXX. Hopefully, you'll find a LD SP,#0000, with perhaps a DI right before it. Write down the address and run the moving routine (you may have to restart the tape, because some of the moving routines insist on a signal coming into the tape recorder). Use a stack trace to find out where the code has gone to. Now you can move all the code from the moving routine to the end of the machine code block to where it should be, given that you know where the LD SP,#0000 goes to. Once moved, patch the loader in the same way as the first Speedlock.

Type 3s have just one long BASIC loader, with about 144 decrypters, but that's nothing to worry about. *Hack the BASIC loader, and have a look at the first bit of code which moves the rest of the code into the right place (you can then use a headerless loader to load this into the right place in memory). The tricky bit is changing a byte in memory so a CALL to the loading system at the very top of the code is changed to a CALL somewhere else once all the decrypters have been run. The only way you can do this is to change the address of the hi byte of this CALL to something else, and RUN the huge load of the decrypters. The computer will crash if you have a Multiface attached, but only after everything's been decrypted, so then
look and see what the CALL's been changed to. If it's suitable, remember the patch, position your hack around this, change the CALL to what it should be, and put in the usual Speedlock patch. Look at the start address in the turboloader. This address will be overwritten by a decrypter once loading finishes. This decrypter is nothing special, so just crack it as usual, and watch out for the game moving around. Jon North's Pokerama Tapes usually have a Speedlock Type 3 crack on it - load up the Pokerama, choose your POKE, then do a stack trace to find it and have a look at it.

Part 7 - Epilogue

Well folks, I'm sorry to have to break this to you, but I've just about told everything you should ever need to know to crack every protection system under the sun. So I'll just say some final words and credits, and then sign off, okay?

The idea of this book, its production and its writing were done entirely by Richard Swann, from February to June 1992.

Some suggestions and tips came from two people to whom this book is dedicated, Matt Corby from "down-the-road", and Niall "Mr Incredibly Technical" Daley. Thanks, guys!

Thanks to YS for putting disassemblers and the like on recent covertapes; it saved me the trouble of writing one!

Thanks to Jon North for some of the info on hacking he gave to me on disk recently - much appreciated, mate.

Thanks for YOU for buying this. There aren't many Speccy hackers around right now, so we need to make the numbers up. Good luck!

What's that? "I don't understand this bit at all!" I hear you say. Well, send me any queries that you may have about this book, stating exactly what the problem is, along with an SAE (very important that), and I'll do my best to reply to them. DON'T write to me asking me to hack a whole list of games for you - I just haven't got the time. However, I've got a big book of Multiface POKEs which you can obtain for £1.50 and an A4 SAE if you want it, so that might come in handy.

Thought for the year: Seven years ago, Spectrum and Commodore owners were at each other's throats. Spectrum owners would vow never to have anything to do with Commodore. Then why do I hear of so many Spectrum owners that have upgraded to a Commodore Amiga? Personally, I can't stand the Amiga's operating system - it's terrible!

Well, that seems to be about it, so I'll just leave you know in the hands of a glossary of terms. Happy hacking!

RICHARD SWANN - June 1992

GLOSSARY

Breakpoint - an instruction put in by a disassembler which will return control to it when it is executed. Also something to do with tennis.

Crack/cracking - writing a routine or executing some code which will get round some element of a protection system, enabling the user to put a POKE into the game.

Crash - an undesired effect which the programmer or user did not intend to happen. Classic example is the computer resetting itself to give the old "(C) 1982 Sinclair etc.etc.", or a whole load of flashing squares. Also a now defunct computer magazine.

Decrypter - a short program contained in a protection system which, when run will change garbage into runnable code. You need to crack a decrypter to write a hack for it.

Endless loop - the Multiface equivalent of a breakpoint, put in hacking a protection system or doing a stack trace. The computer will keep executing the same code over and over again, indefinitely, unless the Multiface button is pressed. See also endless loop (ho ho ho!)

Garbage - A block of machine code which does not make sense to a human. The computer will attempt to process it, but will almost certainly crash. Also what the dustmen collect in America.

Hack - a self-standing program which when run will load a game and activate certain cheats. Also the act of getting round a protection system. (See also crack). Also a lot of anonymous people from Scotland.

Headerless loader - a loader which does not contain the first "header part" of a file specifying it's name, length etc., but has it ready built into memory. This makes the program harder to hack.

Interrupt - small program which occurs every 150 of a second, regardless of what the computer is doing.

Loader - any program which reads a file off tape into memory, and executes it. This may consist of simple BASIC commands, a headerless loader, a turboloader or even a protection system.

Operating system - this the built in program into the computer to deal with all the basic things like reading the keyboard and loading software. This is BASIC in the case of the Spectrum. Most protection systems deliberately confuse the operating system or lock you out of it.

Patch - replacing a bit of code with something designed to hack it. This patch may consist of a jump elsewhere in memory, or a breakpoint. Also something that pirates wear on one eye.

POKE - the process in which a single byte of memory is changed. Originally, games were hacked by one or two POKEs.

Protection system - A block of code which tags on the front of a game's loader and prevents anyone from accessing the code it is protecting. At least, that's what it's supposed to do!

Trace - looking through a block of code with the aim of finding something specific (such as an infinite lives POKE). This may be forwards, backwards, interrupt or stack trace.

Turboloader - a loader which loads in a file off tape at a faster speed than usual. This speeds up loading. The turboloader may be hidden by a protection system.

THE END

```
10 REM *HACK BY JON & RITCHIE
20 LET T=0
30 FOR F=3E4 TO 30083
40 READ A:POKE F,A
50 LET T=T+(F-29990)*A:NEXT F
60 IF T<>544506 THEN PRINT "DATA ERROR":STOP
70 LET T=0
80 FOR F=30085 TO 30200
90 READ A:POKE F,A
100 LET T=T+(F-30075)*A:NEXT F
110 IF T<>919527 THEN PRINT "DATA ERROR":STOP
120 RANDOMIZE USR 3E4:RANDOMIZE USR 30085
130 DATA 221,33,0,80,17,17,0,175,55,205
140 DATA 86,5,48,240,221,126,239,183,32
150 DATA 236,62,2,205,1,22,33,1,80,6,10
160 DATA 126,215,35,16,251,62,202,215,221
170 DATA 70,253,221,78,252,205,43,45,205
180 DATA 227,45,221,54,253,255,62,32,215
190 DATA 62,177,215,221,70,251,221,78,250
200 DATA 205,43,45,205,227,45,62,13,215,42
210 DATA 83,92,221,46,0,195,115,8
220 DATA 62,2,205,1,22,42,83,92,229,237
230 DATA 91,75,92,55,63,237,82,124,181,225
240 DATA 200,70,35,78,35,229,205,43,45,205
250 DATA 227,45,225,78,35,70,35,229,9,34
260 DATA 254,255,225,126,254,13,32,4,35,215
270 DATA 24,212,254,46,40,8,254,58,48,19
280 DATA 254,48,56,15,68,62,14,237,117,205
290 DATA 180,51,229,205,227,45,225,24,220,254
300 DATA 32,56,2,215,126,254,234,32,8,62
310 DATA 13,215,42,254,255,24,167,254,34,32
320 DATA 12,35,126,254,32,56,2,215,126,254
330 DATA 34,32,244,35,24,183
```