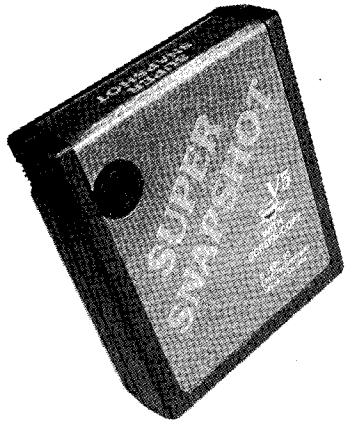
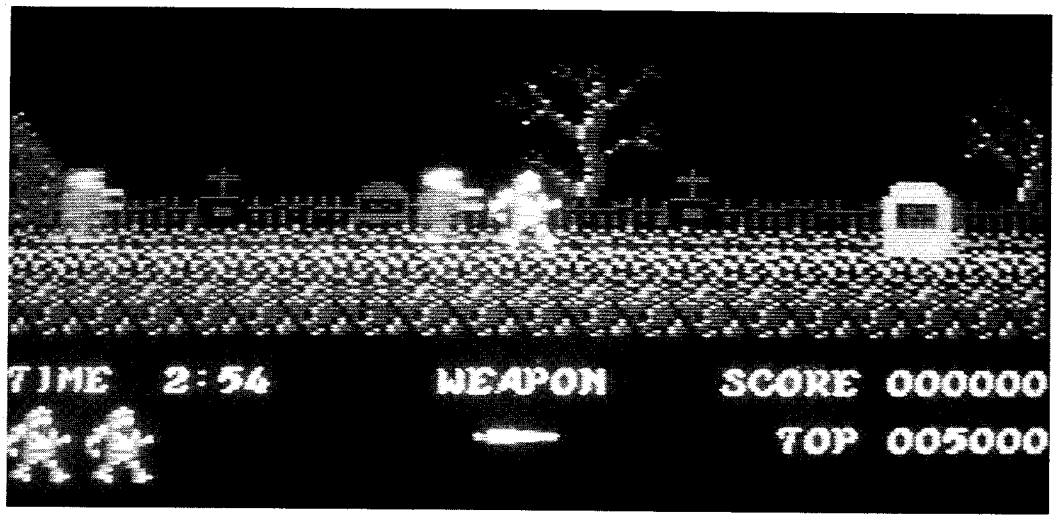
Teil 1

# Super-Snapshot, der Alleskönner



**Angenommen Sie sind stolzer Besitzer einer »Super-Snapshot« und haben sich Grundkenntnisse über den Befehlssatz des 6510-Prozessors angeeignet. Wenn Sie jetzt das Modul optimal nutzen wollen, dann sollten Sie unbedingt weiterlesen!**




**Wir betrachten den Bildschirm von Ghosts'n Goblins etwas genauer. Für die Darstellung der Leben und der aktuellen Waffe wurden Sprites verwendet, die Zeitziffern hingegen bestehen aus Zeichen.**

von Christian Dombacher

Snapshot bietet im Freezer eine Reihe von Funktionen: »Sprite-Kill« oder »Infinite-Lives« u.a. Was aber, wenn wir auch eine Infinite-Weapon-Funktion brauchen, um unser Spiel zu schaffen? Dann müssen wir uns wohl oder übel mit dem Monitor an die Arbeit machen.

Wir wollen alle Schritte an einem praktischen Beispiel demonstrieren. In unserem Fall handelt es sich dabei um »Ghosts'n'Goblins«.

Zuerst laden und starten wir das Spiel, das wir bearbeiten wollen. Nun taucht die erste Frage auf: Wie beginnt man die Suche nach der Speicherzelle, die der Waffenart, den Leben, usw. entspricht? 65536 Speicherzellen durchzusehen, ist wahrscheinlich nicht der richtige Weg. Wir wollen eine Schwäche der Spiele ausnützen, um zu unserem Ziel zu gelangen: den Bildschirmspeicher. Alle Spiele müssen die Anzahl der Leben, die aktuelle Waffe, etc. anzeigen. Es muß also irgendwo im Programm eine Verbindung zwischen dieser Anzeige und unserer Speicherzelle geben.

Befassen wir uns also mit dem Bildschirm. Bei Ghosts'n'Goblins werden die Leben und die aktuelle Waffe in Form von Sprites dargestellt. Nun betrachtet man das gesamte Spiel. Dabei stellt sich schnell heraus, daß mehr als acht Sprites sichtbar sind. Daraus schließen wir, daß eine Rasterzeilen-Interrupt-Routine »im Spiel« ist. Die Rasterstrahltechnik beruht darauf, je nach Position des Rasterstrahls bestimmte Werte in bestimmte Register des VIC zu schreiben. Das Endergebnis ist eine optische Täuschung. In unserem Fall befinden sich die Sprites gerade dort, wo der Rasterstrahl die Bildpunkte auf dem Monitor auffrischt. Kaum ist er damit fertig, sind die Sprites auch schon wieder weg. Was eignet sich besser als eine Interruptroutine, um diese schnellen Umschaltungen vorzunehmen?

Nun betätigen wir den Snapshot-Taster und steigen in den Monitor (<5>,<1>) ein. Um den Anfang der Interruptroutine zu finden, müssen wir uns erst darüber klar werden, welchen Interruptvektor wir auslesen. Falls das Kernel eingeblendet ist (Bit 1 von $01 gesetzt), genügt es, den normalen Interruptvektor ($0314 und $0315) auszulesen. Andernfalls müssen wir die Hardwarevektoren ($FFFE und $FFFF) zu Rate ziehen. In unserem Beispiel müßte nach Eingabe der Befehle folgendes auf dem Bildschirm zu sehen sein:

```
m 0001
:0001 35 17 02 00 91 08 0E 07
m FFFE
:FFFE E5 31 2F 35 17 02 00 91
```

Aus dem Inhalt der Speicherzelle $01 erkennen wir, daß die ROMs ausgeblendet sind und folgern, daß uns der Hardwarevektor $FFFE den Einsprung der Interruptroutine, hier $31E5, verrät. Nun disassemblieren wir den Teil ab $31E5 und stellen fest, daß das Programm an den Inhalt der Speicherzellen $3446 und $3447 springt. Jetzt machen wir uns auf die Suche nach Zugriffen auf diesen Sprungvektor. Dazu benützen wir folgenden Befehl:

```
H 0000 FFFF 46 34
```

Die Ausgabe verrät uns, daß alle Zugriffe zwischen $31A9 und $3420 erfolgen. Bevor wir diesen Bereich disassemblieren, sollten wir uns darüber klarwerden, an welcher Stelle der Bildschirmspeicher zu finden ist. Wir steigen deshalb aus dem ML-Monitor aus und gehen in den Charset-Monitor (Befehl X,<5>,<4>). Uns interessiert die Bank. In unserem Fall liegt der Bildschirmspeicher in Bank 3, d.h. zwischen $C000 und $FFFF.

Wir verlassen den Charset-Monitor und steigen wieder in den ML-Monitor ein (<E>,<5>,<1>). Was wissen wir bis jetzt? Zuerst einmal, daß der Bildschirmspeicher an einer Position zwischen $C000 und $FFFF an einer durch $0400 teilbaren Adresse liegt. Logischerweise muß ein Offset von $03F8 addiert werden, um die Position der Speicherzellen, die das Aussehen der Sprites bestimmen, festzustellen. Mit diesem Wissen disassemblieren wir den Speicherbereich zwischen $31A9 und $3420 und entdecken öfter den Zugriff auf $F7F8. Also ist der Bildschirmspeicher mit größter Wahrscheinlichkeit ab $F400 zu finden.

Nun haben wir über die Interruptvektoren die Adresse des Bildschirmspeichers gefunden. Jetzt taucht natürlich die Frage auf, ob das nicht bei jedem Programm anders sei. Der größte Unterschied zwischen den Programmen besteht in der Interruptroutine, die wir disassembliert haben. Man sollte bei Rasterzeileninterrupts mit den Inhalten der VIC-Register allerdings vorsichtig umgehen. Die durch den IO-Befehl angezeigten Werte dieser Register gelten nur für den im Screen-Copy (Punkt 2 im Freezer) richtig angezeigten Teil.

In unserem Beispiel müssen wir nur noch den richtigen Zugriff auf $F7F8 finden. Wir ersetzten deshalb alle STA-Befehle, von denen wir glauben, sie könnten dafür in Frage kommen, durch BIT-Befehle. Dann setzt man das Programm mit G fort, und kontrolliert, ob sich der gewünschte Sprite verändert hat. Nun kehrt man in den Monitor zurück und ersetzt den BIT-Befehl durch den originalen STA-Befehl. Diesen Vorgang wiederholt man solange, bis die gewünschte Befehlsfolge gefunden ist. Trotzdem können viele dieser Befehlssequenzen durch Überlegung ausgeschlossen werden. Bei Ghosts'n' Goblins entspricht die Anzahl der dargestellten Sprites den Leben, d.h. vier Leben, vier Sprites. Deshalb kommen Schleifen, die alle acht Sprites mit einem Wert initialisieren oder einzelne LDA-STA-Befehlskombinationen weniger in Frage. Wenn man den vorher be-

schriebenen BIT-Test durchführt, tritt eine Schleife in den Vordergrund (sie liegt von $32C9-$32D6), die das X-Register von 0 beginnend hochzählt, bis der Wert in der Speicherzelle $359A erreicht ist. Mit dem Befehl

```
M 359A
```

lesen wir diese aus und erkennen, daß es sich hier wahrscheinlich um die gesuchten Leben handelt. Da es kein Spiel für zwei Spieler ist, begnügen wir uns anfangs mit der Suche nach einem DEC-Befehl. Falls dieser nicht fruchtet, sucht man nach DEC,X oder DEC,Y. Bringt auch dies keinen Erfolg, sucht man nach Zugriffen auf die gewünschte Speicherzelle. Manchmal kann es allerdings passieren, daß wir es nur mit einem Zwischenspeicher zu tun haben, und man das Endziel noch gar nicht erreicht hat. Dann genügen meistens ein paar D-Befehle, um Licht in die Sache zu bringen. Nun zurück zu unserem Beispiel, wir geben ein:

```
H 0000 FFFF CE 9A 35
```

Und wirklich, ein DEC $359A wird gefunden. Es können auch mehrere sein. Wir ersetzen wie gewohnt den DEC-Befehl durch einen BIT-Befehl. Eine Abfolge von NOP's ist in den meisten Fällen unpassend, da hinter dem jeweiligen DEC-Befehl Branch-Befehle (BEQ, BNE) stehen, die sich am Flagregister orientieren. Dehalb benutzen wir den BIT-Befehl, um die Flags richtig zu setzten. Voller Erwartung stürzen wir uns mit G ins Spiel. Und wirklich, alles läuft wie geplant.

## Need more weapons

Da wir schon viel Vorarbeit geleistet haben, fällt uns die Suche nach den Endloswaffen schon leichter. Nun suchen wir nach einer einzelnen STA-Kombination. Die meisten Interruptroutinen sind so organisiert, daß jede Subroutine einen Teil des Bildschirms verwaltet. Falls das Programm nicht so sauber wie Ghosts'n'Goblins programmiert ist, kann es auch vorkommen, daß sich die einzelnen Subroutinen direkt durch Modifizierung der Interruptvektoren in der Routine selbst abwechseln. In unserem Fall kennen wir den Teil, der die Anzeige der Leben und Waffen verwaltet, bereits. Er befindet sich nämlich bei der Schleife für die Leben. Wieder können durch Überlegung viele Möglichkeiten ausgeschlossen werden (die LDA #$FE - STA $F7xx - Kombination kann für die Darstellung verschiedener (!) Waffen nicht verantwortlich sein, außer der Autor bedient sich der äußerst unsauberen Programmiermethode der Codemodifikation). Die einzig mögliche Kombination befindet sich ab Adresse $32E0. Die Speicherzelle $3528 wird als Index für eine Tabelle verwendet. Der BIT-Test beweist äußerst schnell, daß wir auf dem richtigen Weg sind. Also lassen wir uns mit

```
M 3528
```

den Inhalt von $3528 ausgeben. Nun unternehmen wir den nächsten Schritt: Wir erhöhen den Inhalt dieser Speicherzelle und setzen das Spiel mit G fort. Nun können wir mit Hilfe des ML-Monitors jede Waffe verwenden. Trotzdem sollte man mit den Werten, die man eintragen will, vorsichtig umgehen. Es kann passieren, das nur jede gerade Zahl verwendet wird und die ungeraden zum Absturz führen. Auch sollte man darauf achten, daß die Ziffern nicht zu groß werden. Angenommen, im Spiel gibt es fünf verschiedene Waffenarten, denen die Zahlen 0-4 zugeordnet sind, kann das Eintragen einer 7 zum Absturz führen.

Eines sollte man allerdings bedenken: Die Aufgaben, die der Interrupt zusätzlich übernommen hat (z.B. das Auswerten der Waffenart per Tabelle), können auch außerhalb der Interruptroutine erledigt werden. Man schreibt die Ergebniswerte einfach in einen Zwischenbuffer und überträgt diesen in der Interruptroutine in die VIC-Register. Dies geschieht meistens aus Zeitgründen. Deshalb sollte man die kleinen, unauffälligen Schleifen nicht so einfach beiseite schieben.

## Immer diese Hetze

In vielen Fällen sind Leben oder Waffen als Zeichen dargestellt. Auch bei Ghosts'n'Goblins findet sich etwas, das in Zeichen dargestellt ist: die ablaufende Zeit. Man muß sich allerdings klar sein, daß die Zeitziffern nicht während des Interrupts in den Bildschirmspeicher geschrieben werden müssen, da keine optische Täuschung notwendig ist. Trotzdem werden die Daten meistens in der Interruptroutine ausgewertet, da man sich nicht mehr darum kümmern muß. Einen solchen Fall hatten wir bereits: Die Tabellenauswertung der Waffenart. Man setzt einen bestimmten Wert und schon wird der Sprite mit dessen Farbe automatisch angezeigt.

Nun muß die Position des Textes durch Abzählen der Zeilen und Spalten festgestellt werden. Danach multipliziert man die Zeilen mit 40 und zählt die Spalten dazu. Man rechnet das Ergebnis in HEX um und zählt die Startadresse des Bildschirmspeichers dazu. In unserem Fall ergibt die Rechnung $F6FF als Position der Ziffern im Speicher. Mit dem Befehl

```
M F6FF F777
```

lassen wir uns einen Teil des Bildschirmspeichers ausgeben. Wenn wir die ASCII-Darstellung auf der rechten Seite betrachten, können wir allerdings nur wenig Ähnlichkeit zu dem feststellen, was sich wirklich auf dem Bildschirm befindet. Die Lösung des Rätsels heißt »geänderter Zeichensatz«. Bei näherer Betrachtung kristallisieren sich doch einige Zeichen heraus, z.B. das oft verwendete Leerzeichen hat den Wert $66. Testweise kann man den Wert an der errechneten Position erhöhen und im Screen-Copy das Ergebnis anzeigen lassen. Erweist sich die errechnete Position als richtig, kann man mit der Suche nach einem Zugriff auf diese Position im Bildschirmspeicher beginnen. In unserem Fall sieht der Befehl folgendermaßen aus:

```
H 0000 FFFF FF F6
```

Wieder wird nur eine Speicherzelle ausgegeben: $0F6B. Wenn wir das Umfeld des STA-Befehls näher betrachten, erkennen wir neben der Anzeigeroutine für die Zeit (STA $F6FF,$F701,$F702) auch den Zähler, der mit SBC-Befehlen realisiert wurde. Wir scrollen nun solange nach oben, bis wir den ersten SBC #$01-Befehl erreicht haben. Nun verhindern wir das Ablaufen der Zeit, indem wir den Wert, der subtrahiert wird, auf Null setzen. (pk)

### Teil 2

In der nächsten Ausgabe gehen wir unter anderem auf das integrierte Terminalprogramm, auf den Monitor und auf den eingebauten 8 KByte RAM-Baustein ein.