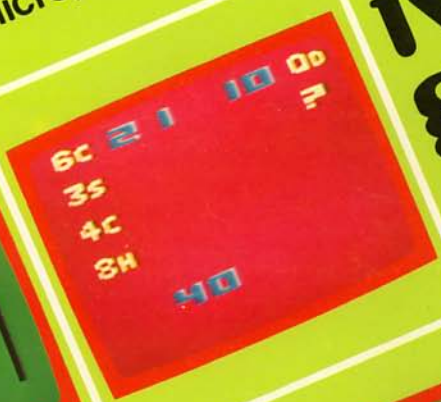
an exciting introduction to microprocessors

tv games computer



Elektor Publishers Ltd.

The TV Games Computer

an exciting introduction to microprocessors

P. Holmes

Elektor Publishers Ltd.

Printed in the Netherlands.

ISBN 0 905705 08 4

Introduction

The first acquaintance with microprocessors can be rather frightening. You are not only confronted with a large and complex circuit, but also with a new language: 'bytes', 'CPU', 'RAM', 'peripherals' and so on. Worse still, the finished article is a miniature computer and so you have to think up some sufficiently challenging things for it to do!

This book provides a different – and, in many ways, easier – approach. The TV games computer is dedicated to one specific task: putting an interesting picture on a TV screen, and modifying it as required in the course of a game. Right from the outset, therefore, we know what the system is intended to do. Having built the unit, 'programs' can be run in from a tape: adventure games, brain teasers, invasion from outer space, car racing, jackpot and so on. This, in itself, makes it interesting to build and use the TV games computer.

There is more, however. When the urge to develop your own games becomes irresistible, this will prove surprisingly easy! This book describes all the component parts of the system, in progressively greater detail. It also contains hints on how to write programs, with several 'general-purpose routines' that can be included in games as required. This information, combined with 'hands-on experience' on the actual unit, will provide a relatively painless introduction into the fascinating world of micro-processors!

Contents

Construction and operation

A closer look at the TV games computer

Writing programs

The extended version

Final notes

Appendix

Acknowledgments

We wish to thank NV Philips Gloeilampenfabrieken, who assisted with the design of the basic unit and provided the monitor software; Mr. P.J. Dickers, who helped to design the extension board; Mr. R. Pequet, who provided the original design for the EPROM programmer. On the software side, the improved text routine was suggested by Mr. G.M. Roberts; the basic idea for the disassembler routine was sent to us by Mr. M. Saliger. Several other enthusiasts, and Mr. M.V. Norman in particular, sent us comments and suggestions that have proved extremely helpful.

The printed circuit boards mentioned in this book are available from the Elektor Printed Circuit Board Service. For further information you are referred to the most recent Elektor issue. This will also contain details about the Elektor Software Service.

hardware:	**EPS number:**
UHF/VHF modulator	9967
main board	79073
power supply	79073 - 1
keyboard	79073 - 2
extension board	81143
random number generator	81523
EPROM programmer	81594

software:	**ESS number:**
test patterns, PVI programming, space shoot-out, etc.	ESS 006 (record)
jackpot, reversi, amazone, code breaker etc.	ESS 007 (cassette)
invaders, fishing, maze adventure, memory, pontoon, nim, etc.	ESS 009 (cassette)

1

μP TV games

Doing battle with chips.

'Chips', in the electronic sense, used to be transistors. Then came the Integrated Circuit, quickly followed by 'MSI ICs' (Medium Scale Integration Integrated Circuits) – quad opamps, for instance. Inevitably, these were followed by 'LSI ICs' (Large Scale Integration): complete alarm clocks, TV games and microprocessors. It's amazing how much circuitry manufacturers can squeeze onto a 'chip' nowadays!
However, microprocessors have posed an unexpected problem. We've got them, but very few amateur electronics enthusiasts seem to know what to do with them! The best approach would appear to be to simply use them as 'complicated integrated circuits', build something interesting around them, and quickly proceed to forget what component makes the unit tick. That particular approach is what we have in mind with the circuit described here. Once completed, it is a box with a keyboard and joystick controls; after playing some 'space sounds' off a tape or disc into it, it becomes a sophisticated TV games machine – with full colour, all sorts of 'object shapes', score-on-screen and sound.
At a later date, it should also create an irresistible urge to 'program' ones own games. This will prove to be relatively easy: the unit just happens to contain what microprocessor 'freaks' call 'comfortable monitor software'...

The TV games computer contains a microprocessor – μP for short. So be it. There's no reason to be scared of μPs – they don't bite! Furthermore, absolutely no knowledge of microprocessors is required to have fun building and using this unit. At the same time, it offers a unique possibility of getting to grips with μPs at a later date, simply by exploiting its capabilities to the full.
This first chapter is intended as a basic introduction to the unit. The descriptions of what it does – and how – are sufficient for those readers who want to build and use it with the TV games programs available via the Elektor Software Service (ESS). A much fuller description, for those who want to develop and program their own games, will be given later. For the moment, let's concentrate on what it does.

What it does

The unit is intended for TV games. This means that it must create a (colour) picture on a standard (PAL) TV set, corresponding to some game

– man against man or man against machine. The effect – and the fun – can be enhanced by including sound effects. Games can be won or lost, so a 'score' display is also required.
The first requirement for a 'game' display are 'objects' on the screen. In early TV games, these consisted of two vertical bars (bats) and a small square (ball). Nowadays, complete cowboys, battleships or jet aircraft are more in demand. The TV games computer can cater for all tastes.
A single 'object' can consist of up to 80 squares in an 8 x 10 rectangle (see table 2). By filling in the necessary squares, quite detailed objects can be displayed: as an example, in figure 1 a locomotive fills the centre of the screen.
The same object can be repeated in different parts of the screen; furthermore, several sizes are available: the little steam engine at the bottom of the screen in figure 1 is a scaled-down version of its big brother in the centre. To be more precise, any object can be displayed in four different sizes: 1:1, 2:1, 4:1 and 8:1.
The shape of the object can be re-programmed quite rapidly, so that the same 'section of memory' can produce several different objects (or object sizes) in the same picture. However, this is a complicated system for microprocessor specialists only; for most users it is more interesting to know that four completely different objects can be 'stored' simultaneously. A battleship, submarine, jet aircraft and 'missile', for instance; or, for a more peaceful game, two footballers, a goal (displayed twice) and a ball.
The colour of the objects can also be determined according to personal taste: a choice of eight colours is available.

Each of the primary colours (red, green and blue) can be selected independently, so that the various mixtures are also available; for instance, red plus green equals yellow. All in all, red, green, blue, three mixtures (orangy-yellow, greeny-blue and purple), white and black can be produced.
For most games, objects are not enough: a 'background' is also required. This may consist of anything from a partial boundary to a complete crosshatch pattern. Building up the desired background with the TV games computer is similar, in many ways, to building up an 'object'. Basically, the background consists of 160 squares: 10 rows of 16 squares each. Each square is defined by its upper and left-hand sides: the right-hand side and

Table 1.

The TV games computer in a nutshell

- both joystick and keyboard control;
- video output suitable for PAL colour TV receivers (UHF and VHF input);
- sound effects via built-in loudspeaker;
- microprocessor controlled, and therefore programmable for a wide variety of games;
- cassette interface included, for easy loading and storing of programs (e.g. readymade games supplied through the Elektor Software Service);
- extensive monitor software.

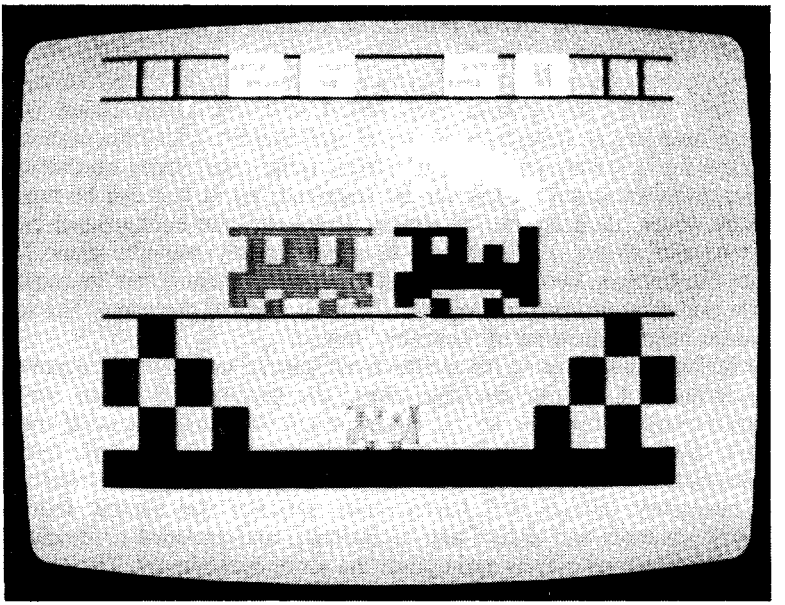


Figure 1. This photo illustrates the various display possibilities of the TV games computer.

bottom 'belong' to the adjoining squares. Any of these sides can be displayed, as required; furthermore, by 'widening' the left-hand edge it is possible to fill in the entire square.

This system can be used to create a wide range of different background patterns. A few examples are shown in figure 1: all the squares along the bottom row have been 'filled in' to produce a solid edge. The checker-board pattern in the lower left- and right-hand corners is achieved by filling in alternate squares; the cross-hatch along the top consists of the sides of the corresponding squares only. Various other horizontal and vertical lines, bars and dots are all variations on the same theme.

The colour of the general background 'between the lines' and that of the lines themselves can both be chosen independently. As before, a choice of eight colours are available. The only restriction is that if the same colour is chosen for both background and 'object', the latter will not be visible!

The final point, as far as the picture is concerned, is the possibility of displaying the score. The TV games computer offers four alternative display modes, each consisting of four digits. These digits can be displayed either at the top or at the bottom of the picture; furthermore, they can be used either as a single four-digit number (e.g. 2650) or as two two-digit numbers (26 50). In figure 1, the latter option is displayed at the top of the screen. The colour of the 'score display' is always chosen to contrast with the colour of the background lines.

So much for the picture. However, this by no means exhausts the capabilities of the games computer. As mentioned earlier, sound effects can also be included: the basic unit contains a 'programmable squarewave

generator' that can be used to produce all kinds of squawks and whistles via a small built-in loudspeaker.*

In most games, the score depends to a large extent on 'hits' and 'misses'. In this unit, collisions between each object and any other object or the background are all detected and stored individually. This information can be used for a variety of different effects, quite apart from up-dating the score. Suitable sound effects can be triggered; an object can be made to change shape, disintegrate, or simply disappear; the background can be altered; and so on. For instance, in a particularly war-like game like a shoot-out between two cowboys, each time one of them is 'hit' he could fall down and then proceed to float slowly up to and off the top of the screen – to the accompaniment of 'heavenly' music!

The facilities discussed so far are the basic 'pieces' out of which a game can be constructed. For an exciting game, the picture and sound must be modified continuously – as the players manipulate their joy-stick controls and/or keyboard. This is where the 'brain in the box' comes in: once the microprocessor has been programmed for a particular game it will set up the necessary objects and background, monitor the signals from the players' controls and detect any 'collisions', and proceed to modify the display, add sound effects and update the score accordingly. The beauty of this system is that merely running a new program from tape into the machine is sufficient to obtain an entirely different game – with the number of possible games being virtually unlimited!

Furthermore, the microprocessor is sufficiently 'intelligent' to be a formidable opponent in solo games (man against machine), provided the rules of the game are not too complicated: the memory in the basic unit is insufficient for games like chess.

Having discussed, in general terms, what the 'games computer' can do, the next step is to find out how. In other words, what makes it tick?

* The extended version, described later, includes two sophisticated 'Programmable Sound Generators'.

Table 2.

Picture composition, basic facilities

Four different 'objects', each of which:
- consists of up to 80 squares in an 8 x 10 rectangle;
- can be located simultaneously at different positions on the screen;
- can be displayed in four different sizes (x1, x2, x4 or x8);
- can be displayed in any of eight different colours.

A background, consisting of:
- a cross-hatch pattern 10 x 16 squares; any parts of the horizontal and vertical lines can be displayed or suppressed as required, and the squares can be partially or wholly filled-in; eight colours are available;
- a general background 'between the lines', once again with a choice of eight different colours.

A 'score' display:
- four digits at the top or bottom of the screen;
- displayed as two two-digit numbers (e.g. 14 92) or as one four-digit number (e.g. 1492);
- displayed in the complementary colour to the lines in the background.

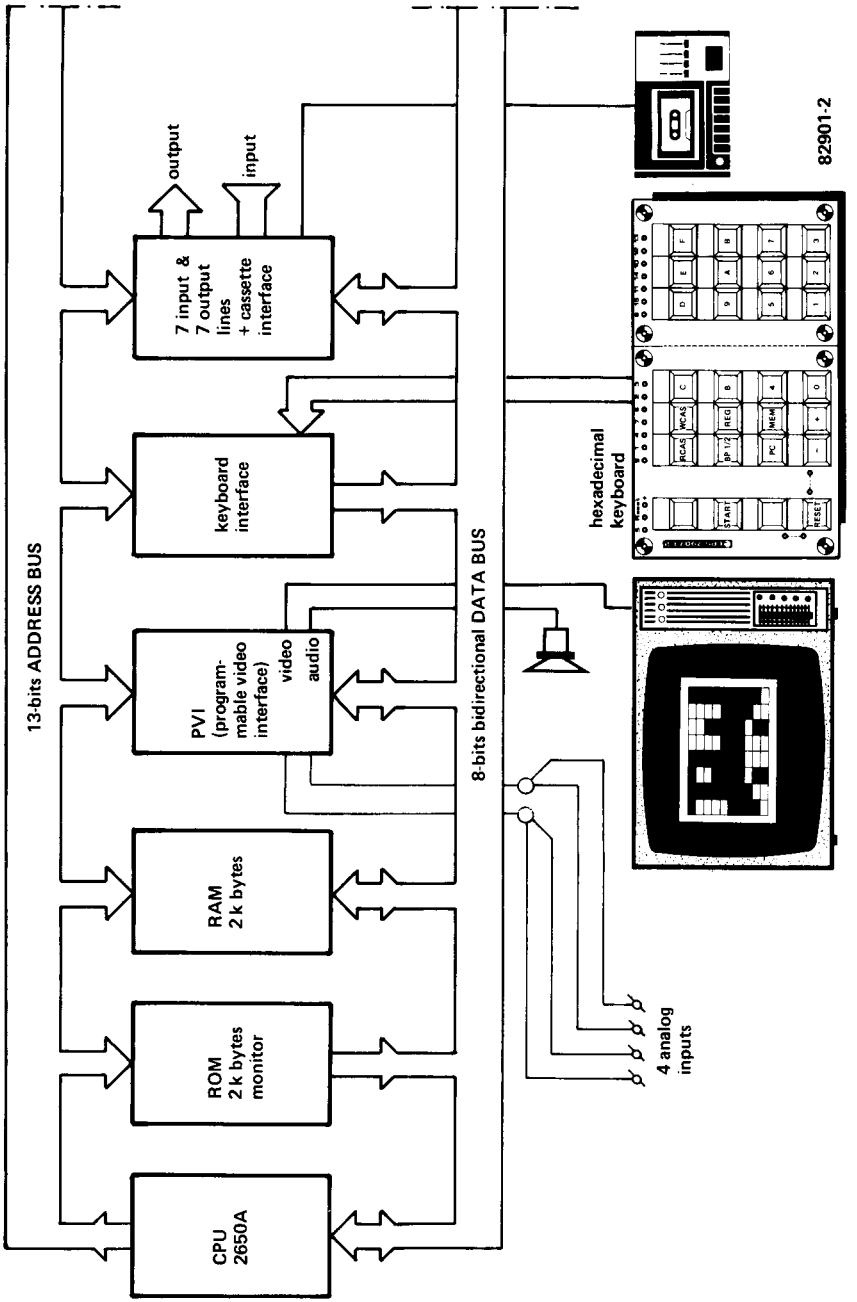


Figure 2. Simplified block diagram of the TV games computer (basic version).

How it works

At this stage, we only intend to give a very broad outline of the operating principles. Some more detailed explanations will be given later.

A functional block diagram is shown in figure 2. In the first block, 'CPU' stands for Central Processing Unit. This is the actual microprocessor chip – the 'brain', in other words. It controls all the other units, calling them up as required via the 'address bus'. Although microprocessor specialists seem to take particular delight in describing this type of process in (mistifying) jargon, it is basically a fairly straightforward circuit. Code locks and code switches are fairly well-known, and several circuits have been published over the years.

Imagine, now, that each of the other blocks contains one or more of these code switches. When the correct code appears at the input of one of these switches (i.e. power on the correct lines and no power on the wrong ones), the switch operates and the corresponding part of the block is turned on. Provided each individual circuit in the whole unit is operated by its own code switch with its own unique code, the CPU can turn on any one of them by simply putting the correct code on the address bus.

A 'brain', on its own, is fairly useless. A 'memory' is also required, and two different memories are actually available in this unit. The first is a 'ROM' (Read Only Memory); it contains all sorts of 'standard reference' information – how to read programs off a tape and how to store them on tape, for instance. The second block of memory is labelled 'RAM', which stands for 'Random Access Memory'. Information can be stored here and recalled as required; its primary use is for storing the program for a particular game, as well as information that changes in the course of each game – such as the scores.

CPU, ROM and RAM could be called the 'internal organs' of the unit. On the outside, there is a TV set, a keyboard, joysticks, a loudspeaker and a tape recorder. In computer language, these are all called 'peripherals'. None of these, of themselves, can be controlled by the CPU – a very unsatisfactory state of affairs. This failing is rectified by adding suitable conversion and switching units between the peripherals and the computer proper. Once again in computer language, this type of 'buffer unit' is called an 'interface'.

Going back now to the block diagram, the remaining sections require little further explanation. The fourth block is labelled 'PVI' (Programmable Video Interface); it is the 'interface' which provides the outputs for the TV set and the loudspeaker. It also takes care of the analog inputs from the joystick controls. The next block, the 'keyboard interface', is exactly what its name implies. Finally, an unnamed block takes care of the remaining in- and outputs – the tape or cassette recorder in particular. The other 7 pairs of in- and output lines from this block can be disregarded for the present.

The text '8-bit bidirectional databus' in the link along the bottom of the blocks is another piece of computer jargon. It signifies that output signals from the various sections can be passed along to other units – the arrows illustrate the possibilities. The CPU, for instance, can receive information from any of the other blocks, or it can itself provide information. By

contrast, the ROM and the keyboard can only provide information (on request, when called up via the address bus).
The way in which the various units work together (under central control of the CPU) is best illustrated by an example. However, before we come to that we must first build the (basic) unit.

2

Building the TV-games computer

Having seen what the TV games computer can do, the next step is to build it! After a brief description of the circuit, we will concentrate on the constructional details and calibration procedure.

One glance at the main circuit diagram is probably sufficient to scare off all but the hardiest electronics enthusiasts. However, the block diagram is not nearly as bad (figure 3) and, once it has been understood, it will serve as a guide through the main circuit.

As explained in the introduction, the 'brain' of the TV games computer is the microprocessor chip, or 'CPU'. By passing 'switching signals' along the 'address bus' (13 wires), it can operate all other sections of the unit as required. Information is passed from one unit to another along the 'data bus' (8 wires); finally, several special-purpose control signals are connected direct from the CPU to the unit(s) involved.

Without a memory, a brain is fairly helpless. Three distinct types of memory are available in this unit: The Read Only Memory (ROM), containing the pre-programmed 'monitor software'; the Random Access Memory (RAM), that is used for storing the actual 'game' program; and, finally, a cassette or tape recorder for long-term storage of as many different 'game' programs as may be desired. The type of memory that is to be used at any particular moment is selected (under CPU control) by the 'address decoder'; the exact part of that memory from which data is to be retrieved (or in which it must be stored) is selected by the CPU itself, via the address bus.

Since most tape and cassette recorders are designed for audio work, using them to store digital signals requires some careful signal handling. The digital output from the computer to the tape must be AC-coupled and filtered to remove the extreme high-frequency components; the input from the tape to the computer must be boosted in level and 'cleaned up' to produce a recognisable digital signal. These operations are performed by the section labelled 'cassette interface' in figure 3.

The sections described so far are common to virtually any computer system. 'Intelligence' – the CPU – and 'memory'. It may be noted, in passing, that precisely this part of the system is shown on the left-hand page of the main circuit diagram. However, the TV games computer has still to be connected to the 'player' controls (joysticks and keyboard), the TV set and a loudspeaker.

First, the controls. The joysticks, being potentiometers, are basically analog devices. To adapt them to the otherwise digital system, some kind of analog-to-digital conversion is required: a 'joystick interface', which is

actually part of the PVI. However, to save space only one 'interface' is actually available and so only one joystick can be dealt with at a time. For this reason, a 'joystick selector' is included, to switch to and fro between the two controls. This unit is, of course, under CPU control (via a separate connection); it is also connected to the 'keyboard interface'. The latter unit feeds the data from the keyboard onto the data bus – again under CPU control (via the address bus). So much for the inputs. The outputs – to TV and loudspeaker – are rather more complicated. Happily, most of the work is done by a single IC: the PVI, or Programmable Video Interface. This unit is comparable to a 'slave' micro-computer: under CPU control it stores and supplies data as required; it detects certain situations (e.g. inter-object collisions); based on the data stored, it 'creates' the corresponding picture and sound signals. In fact, the 'sound' in particular is simplicity itself. A single buffer stage, connected to the corresponding PVI output, can drive a small loudspeaker.

To create the 'picture' the PVI must enlist the aid of a few ancillary circuits. A crystal-controlled oscillator produces the basic timing signals. These are fed, through frequency dividers, to another useful IC: the 'Universal Sync Generator' (USG). This unit takes care of all the sync signals required for a modern colour TV set, and produces some additional synchronising signals for other parts of the circuit almost as a by-product.

One set of outputs from the USG is fed to the PVI, to tell it what part of the picture is actually being 'written' at any particular moment. Based in part on this information, the PVI produces a group of output signals that determine what colour must be displayed at that point in the picture, in order to reproduce the required display of objects, background and score. These outputs are fed, via a 'gating' circuit controlled by the USG, to the final section: the 'digital video summer'. This unit does exactly what its name implies: it sums the outputs from the crystal oscillator, the USG and the PVI to produce the total video output. 'Summing', in this case, should be taken in the broadest possible sense: it includes frequency division, gating and level matching.

Having a video output is all very well, but most TV sets only have UHF or VHF inputs. This problem is quickly solved, by adding a UHF/VHF modulator.

The circuit

Having discussed the block diagram, we can now risk a quick look at the main circuit diagram (figure 4). However, there is little point in going into great detail. The main point to note is that the layout corresponds, by and large, to that of the block diagram: the various sections are drawn in the same relative positions.

The CPU (IC1) is at the left; the address and data busses run along the top. The address decoder for the memory and input/output selectors (IC6 and IC7), the ROM (IC2) and the RAM (IC13... IC28) require little explanation. The only point to watch is that 2112's used in the RAM must be 450ns

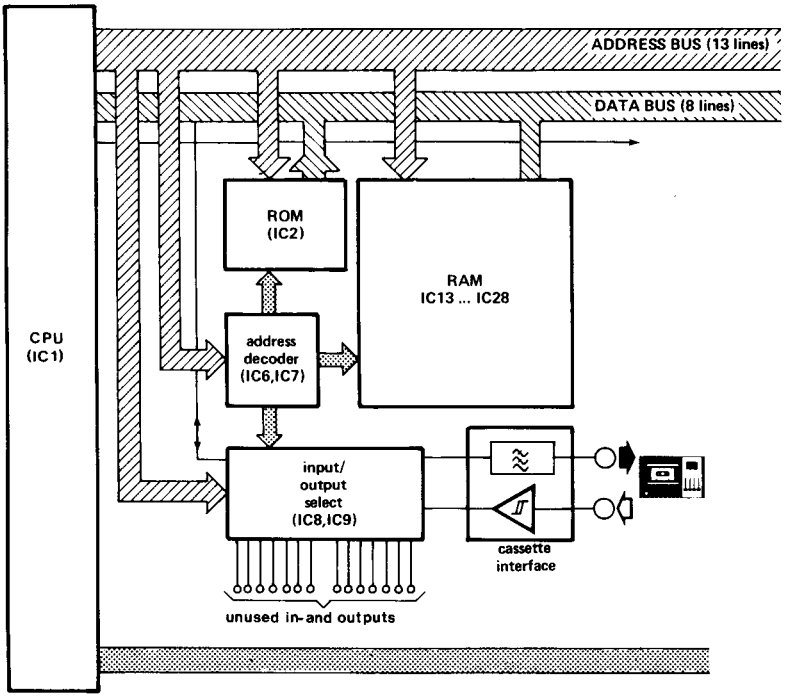


Figure 3. Block diagram of the TV games computer. The CPU, memory and cassette interface are shown above; the joystick input and TV and sound output to the right.

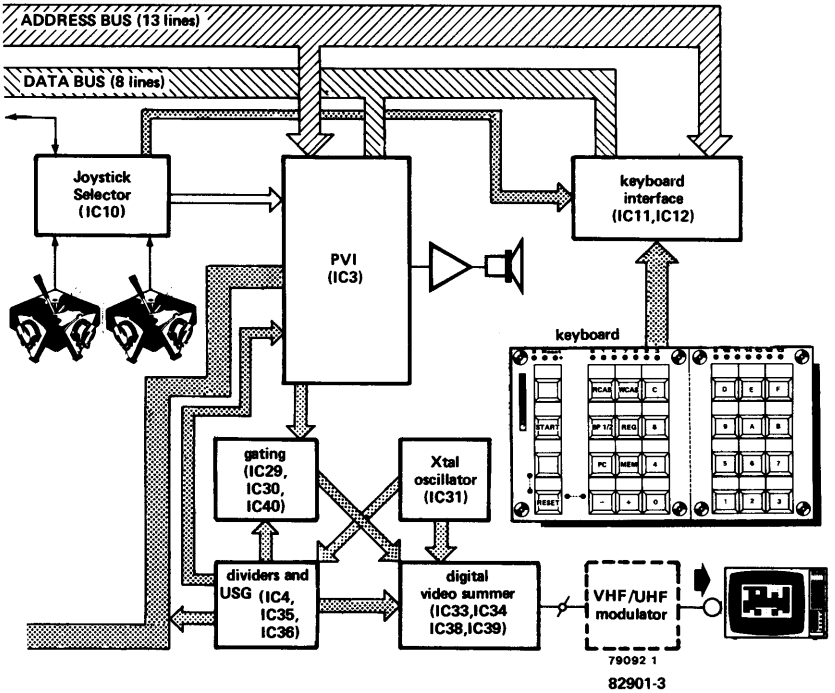
versions (or faster). The input- and output selectors (IC8 and IC9, respectively) offer eight 'serial' inputs and eight outputs. However, only one of each is used in the basic unit; the other seven in- and outputs are available for other external devices if required. The 'output filter' in the cassette interface consists of three resistors and two capacitors; the 'input buffer' uses an opamp to boost the signal to TTL level.

So much for the first half of the circuit! This 'thumbnail' description should suffice to give a general idea of 'what does what'. A fuller and more detailed explanation will be given later.

The second half of the circuit is rather more complicated at first sight (well, even at second and third sight...). However, it is not too difficult to locate a few important sections, by referring to the block diagram. The 'heart' of this part of the circuit is the PVI (IC3): as mentioned earlier, it is almost a 'slave' microcomputer in its own right. The fact that it operates in close collaboration with the CPU is readily apparent: it is the only subsection that is connected directly to virtually all the address and data lines.

The PVI is flanked (quite litterally, in the circuit) by the joystick selector (IC10), the 'loudspeaker interface' (T1) and the keyboard with its interface (IC11 and IC12). None of these merit any detailed discussion at this point. The keyboard (or keyboards, depending on how you look at it – them?) will be explained later, from the user's point of view.

Now for the remainder of the circuit. To be honest, this type of circuit



should either be described in detail or not at all... However, we will attempt to give a very rough outline, without Qs, $\overline{Q}$s, logic ones and zeroes.
Immediately below the PVI, a crystal oscillator (IC31) is used to generate the main timing signals. One of these goes down and around, passes through a divider stage consisting of IC32, IC35 and IC36, and finishes up as the 'clock' input signal for IC4, the 'USG'. This 'Universal Sync Generator' is more important than its relative size in the circuit might lead one to expect: it produces the complicated synchronising signals required for a modern (PAL) colour TV set. Furthermore, it produces reference signals that are used by both the PVI and the CPU; finally, it 'gates' the video outputs from the PVI (i.e. turns them on and off, as required) via an intriguing selection of NANDs, inverters and EXORs (IC29, IC30 and IC40).
What remains in the circuit was all lumped together in the block diagram as 'Digital video summer'. This section (consisting of IC33, IC34, IC38, IC39 and assorted NANDs and inverters) combines signals from the crystal oscillator, the gating network at the PVI outputs and the USG to produce the final video output. The correct relative levels of the various signals are determined by the resistor network in the 'summer' (R54... R62).
The complete circuit shown in figure 4 is contained on a single printed circuit board. More on this later. However, two further units are required: a power supply and (in most cases) a UHF/VHF modulator.

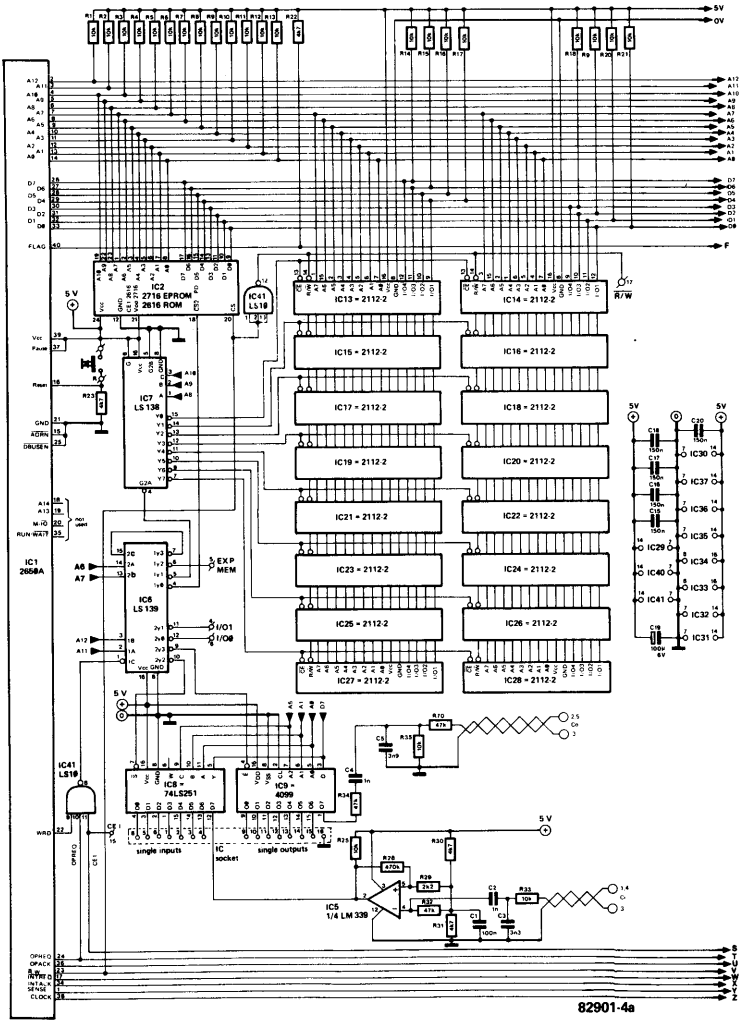
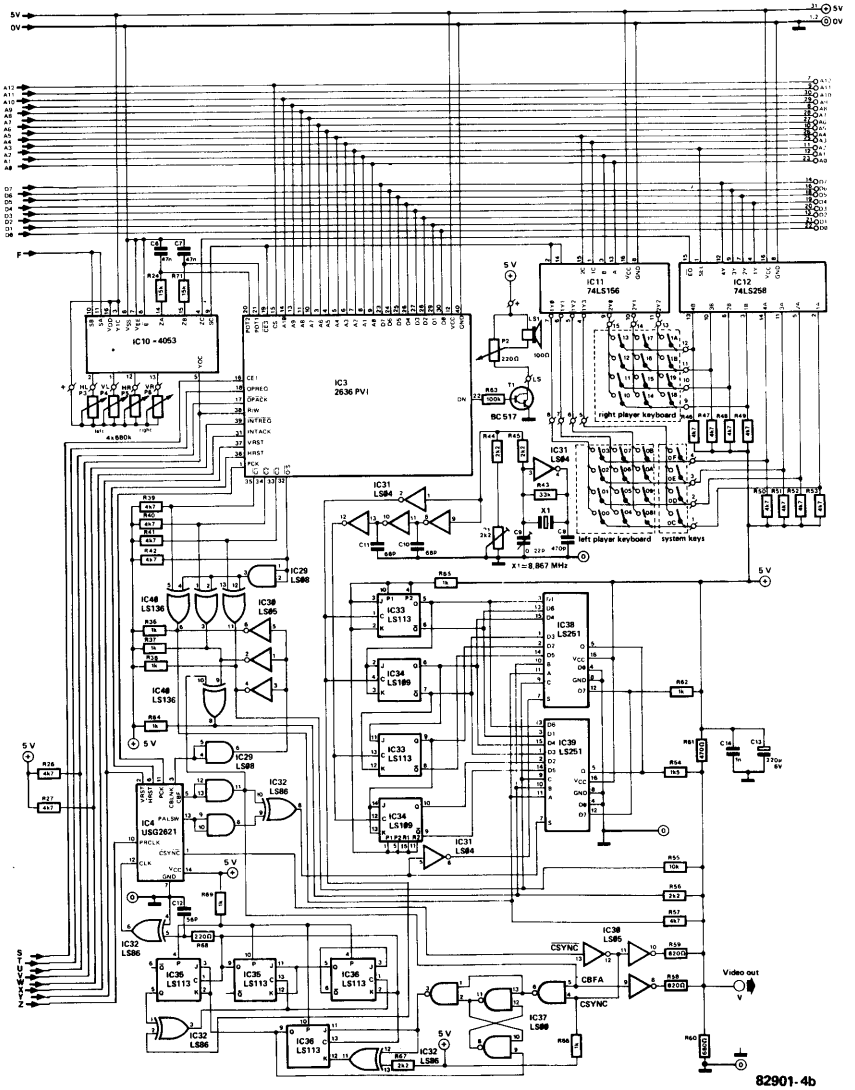


Figure 4. Complete diagram of the main circuit.

Power supply

Any stabilised supply, capable of delivering 5V at 2A, is suitable. A simple circuit is shown in figure 5. Although this configuration may seem rather peculiar, especially where T1 and T2 are concerned, the principle is quite straightforward. If the load current increases, the integrated voltage regulator (IC1) will attempt to supply this current itself. However, in so



doing it will increase the voltage drop across R2, thereby turning on T2 – which then proceeds to deliver the bulk of the current. In the event of a short circuit occurring, T1 limits the current through T2 to a safe value and internal protection circuits maintain the dissipation in the IC within its limits.

A suitable printed circuit board is shown in figure 6.

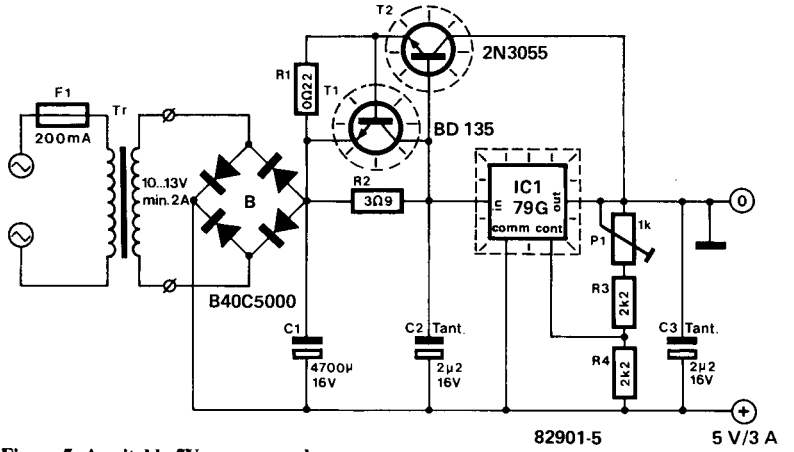


Figure 5. A suitable 5V power supply.

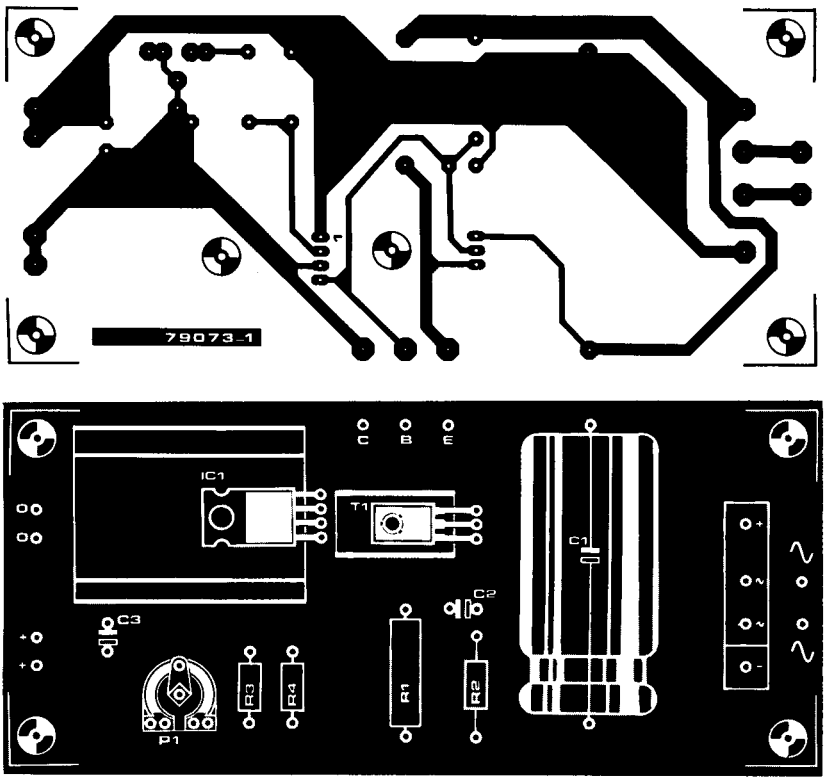


Figure 6. Printed circuit board and component layout for the power supply (EPS 79073-1).

Parts list for power supply.

Resistors:
R1 = 0.22 Ω/3 Watt
R2 = 3.9 Ω
R3,R4 = 2k2
P1 = 1k

Capacitors:
C1 = 4700 μF/16 V
C2,C3 = 2μ2/16 V (Tant.)

Semiconductors:
T1 = BD 135, BD 137, BD 139
T2 = 2N3055
IC1 = μA 79G
B1 = B40C5000

Sundries:
Tr1 = main transformer,
10 V/2A secondary

UHF/VHF modulator

The circuit is shown in figure 7, and the printed circuit board in figure 8. The fundamental carrier frequency is derived from a 27 MHz crystal in an oscillator circuit based on T1. For domestic use, crystal stability is not always required. In that case the crystal, X1, can be replaced by a 10n capacitor. The output signal of this oscillator is amplified by T2 and T3 and differentiated by the three RC networks C3/R4, C4/R6 and C5/(R9 + P1). The resulting waveform at the junction of R8 and R9 is a sequence of short spikes containing harmonic multiples of 27 MHz up to around 1 GHz.
The video signal is fed in via P2 and modulates the carrier by varying the forward bias on D1 and thus changing its impedance. This causes the level of the r.f. signal appearing across R10 to vary in sympathy with the video

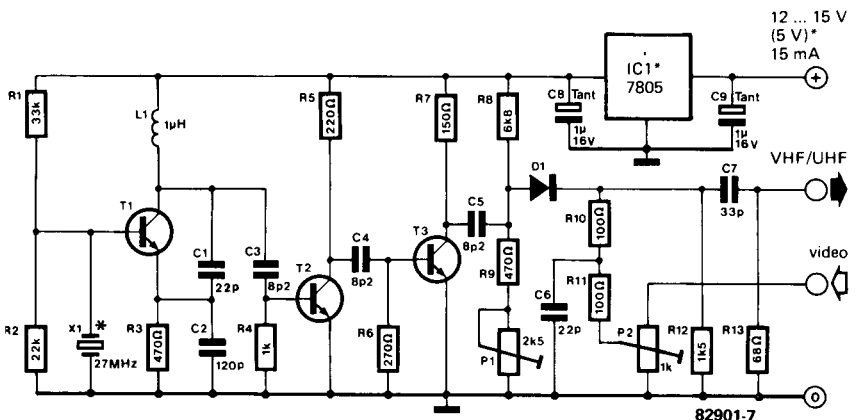
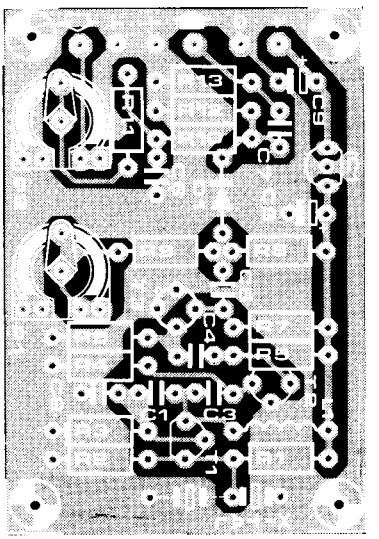


Figure 7. The UHF/VHF modulator. Note that IC1 is not required in this application.



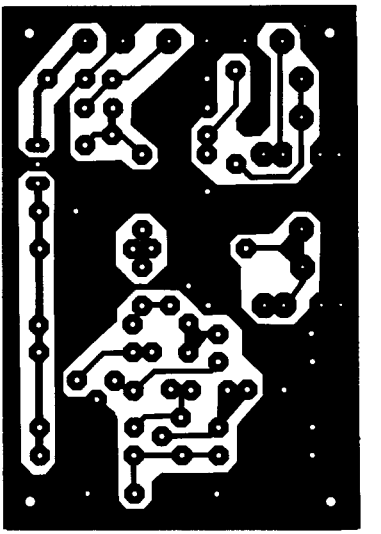


Figure 8. Printed circuit board and component layout for the UHF/VHF modulator (EPS 9967).

input signal, i.e. the carrier signal is amplitude modulated. The signal is coupled out via C7 to a coaxial output socket. R13 matches the output impedance of the modulator to that of the coaxial cable.
Potentiometer P1 can be used to set the carrier level by varying the static forward bias on D1, whilst P2 adjusts the video input level and hence the modulation depth.

Parts list for UHF/VHF modulator

Resistors:
R1 = 33 k
R2 = 22 k
R3,R9 = 470 Ω
R4 = 1 k
R5 = 220 Ω
R6 = 270 Ω
R7 = 150 Ω
R8 = 6k8
R10,R11 = 100 Ω
R12 = 1k5
R13 = 68 Ω
P1 = 2k5 (2k2) preset potentiometer
P2 = 1 k preset potentiometer

Capacitors:
C1 = 22 p
C7 = 33 p
C2 = 120 p
C3,C4,C5 = 8p2
C6 = 22 p
C8,C9 = 1 μ/16 V tantalum

Semiconductors:
T1,T2 = BF 194, BF 195, BF 254, BF 255, BF 494, BF 495
T3 = BFY 90
D1 = 1N4148
IC1 = not required (see text)

Miscellaneous:
L1 = 1 μH
X1 = crystal, 27 MHz approximately.

Construction details

The TV games computer is built up from four basic units: the main circuit, keyboard (figure 11), power supply and UHF/VHF modulator. The wiring between these units (and various external odds and ends: joysticks, loudspeaker) is shown in figure 9.

The p.c. board for the main circuit (figure 10) requires some comment. It is a double-sided p.c.b. with plated-through holes. A prime example of modern technology with, regrettably, the associated 'teething problems'. Technology has not yet reached the point where plated-through holes are 100% reliable (at a reasonable price, that is) and, as an interim solution, manufacturers often simply mount all components on the board and reject any circuits that don't work. This is rather unsatisfactory for the home constructor, and he is more inclined to 'trouble-shoot'. For complicated circuits like the TV games computer, this can be extremely time-consuming. For this reason, it is advisable to check the board *before* mounting the components. A first, visual, check is possible by holding the

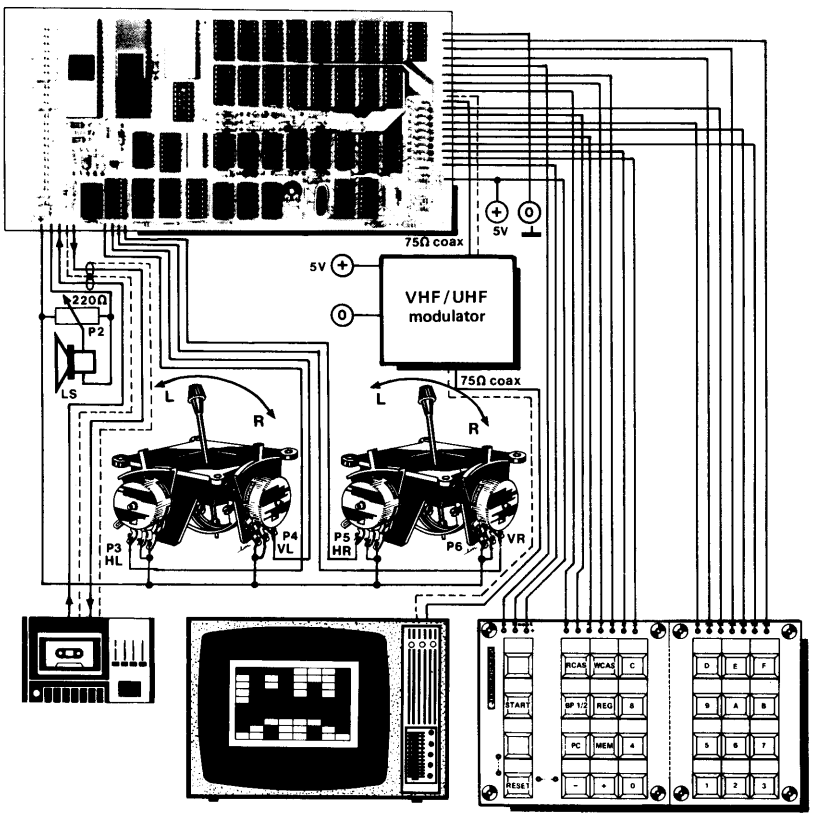


Figure 9. Interconnection between the various units.

board up to the light and looking through the holes: the plating should be clearly visible. To make assurance doubly sure, each hole can be tested individually with a multimeter: with one probe on each side of the board, the resistance should be zero.
When mounting the components, it is strongly advised to use a suitable miniature soldering iron and first-class IC sockets. In a unit like this, tracing dud contacts afterwards can be a harrowing experience.
A minor point worth noting concerns R58. In the circuit, this resistor is shown connected to the video output – and rightly so. However, observant readers have found that it is connected to positive supply on the p.c. board. When this mistake was discovered, we immediately tested several of our prototypes to see what the effect was. To our surprise and relief, it doesn't

Parts list for main board

Semiconductors:
IC1 = 2650A (Signetics)
IC2 = 2616 (Signetics)
IC3 = 2636 (Signetics)
IC4 = 2621 (Signetics)
IC5 = LM339 (National Semiconductor)
IC6 = 74LS139
IC7 = 74LS138
IC8,IC38,IC39 = 74LS251
IC9 = CD 4099
IC10 = CD 4053
IC11 = 74LS156
IC12 = 74LS258
IC13 . . . IC28 = MM2112–4 (450 ns access time)
IC29 = 74LS08
IC30 = 74LS05
IC31 = 74LS04
IC32 = 74LS86
IC33,IC35,IC36 = 74LS113
IC34 = 74LS109
IC37 = 74LS00
IC40 = 74LS136
IC41 = 74LS10
T1 = BC517

Resistors:
R1 . . . R21,R25,R33,R35,R55 = 10 k
R22,R23,R26,R27,R30,R31,R39 . . R42 R46 . . . R53,R57 = 4k7
R24,R71 = 15 k
R28 = 470 k
R29,R44,R45,R56,R67 = 2k2
R32,R34,R70 = 47 k
R36,R37,R38,R62,R64,R65,R66,R69 = 1 k
R43 = 33 k
R54 = 1k5
R58,R59 = 820 Ω
R60 = 680 Ω
R61 = 470 Ω
R63 = 100 k
R68 = 220 Ω
P1 = 2k2 preset potentiometer
P2 = 220 Ω potentiometer

Capacitors:
C1,C15,C16,C17,C18,C20 = 150 n MKH
C2,C4,C14 = 1 n
C3 = 3n3
C5 = 3n9
C6,C7 = 47 n
C8 = 470 p
C9 = 0 . . . 22 p trimmer
C10,C11 = 68 p (cer.)
C12 = 56 p (cer.)
C13 = 220 µ/6 V
C19 = 100 µ/6 V

Sundries:
Xtal: 8.867 MHz
Loudspeaker: 100 Ω/500 mW
2 Joysticks (P3 . . . P6): potentiometer values 680 k.
28 'digitast' switches (for the keyboard).

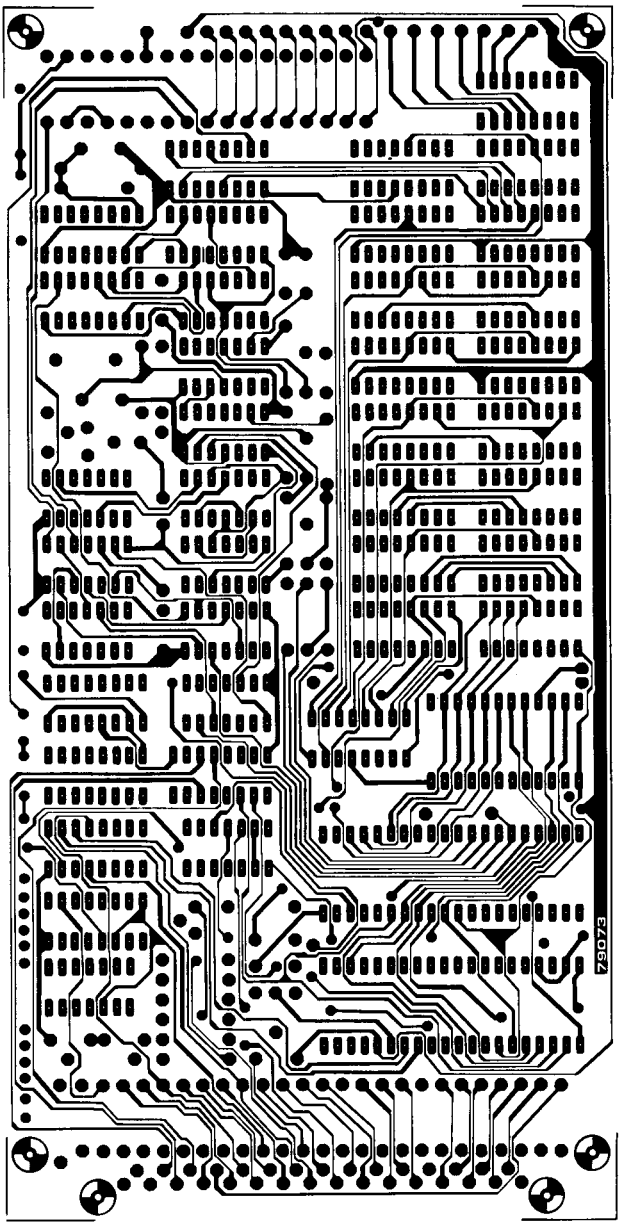


Figure 10. Printed circuit board and component layout for the main circuit (EPS 79073). The board is double-sided with plated-through holes. Note that it is reproduced here at 70% of actual size.

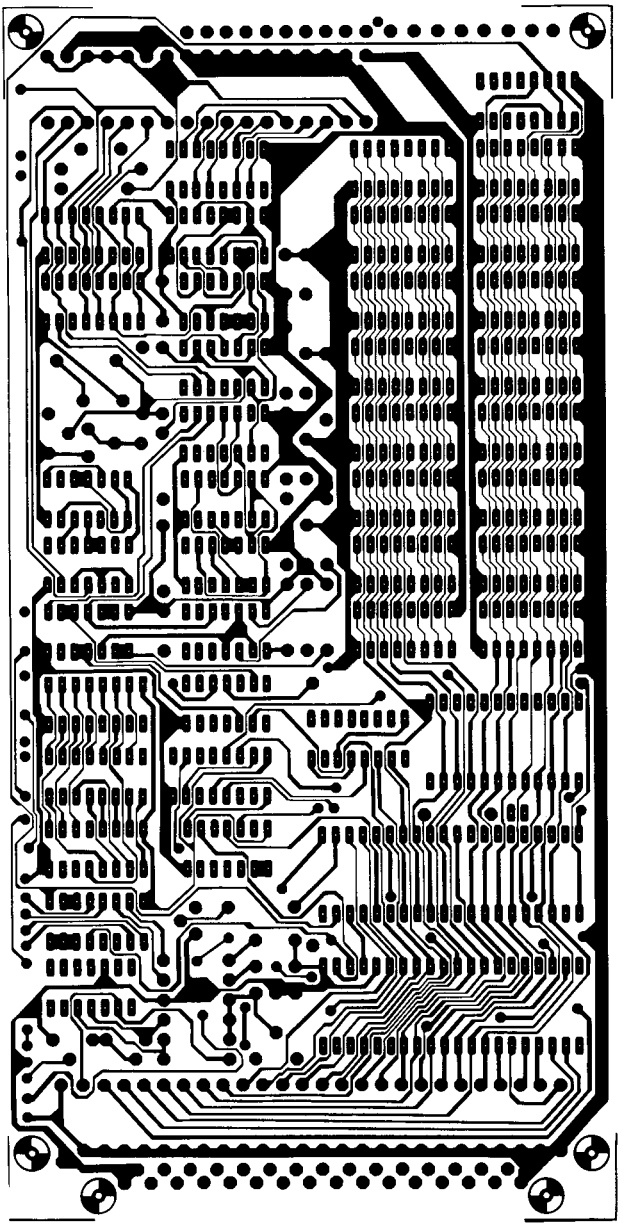


Figure 10.

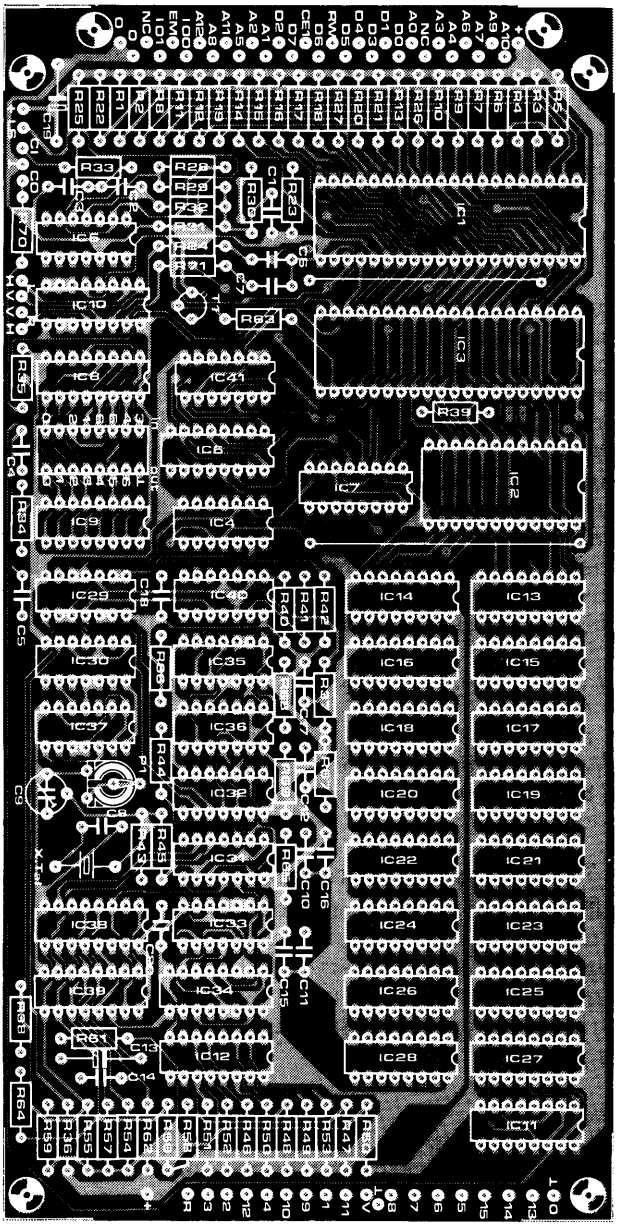


Figure 10.

make a scrap of difference! Which is why we didn't modify the board layout. However, it is quite easy to make the correct connection, by soldering that end of R58 to the adjacent lead from R56. This can be seen in the lower left-hand corner of figure 10.
The main board has a large number of in- and output connections, most of which however remain unused in the basic tv games computer. The connections between main board and keyboard (see figure 9) are clearly numbered on both boards. Note that the two wire links shown as dotted lines on the keyboard p.c.b. should not be mounted in this case.
The key numbers shown on the component layout for the keyboard correspond to the actual addresses of the keys. However, for normal use the indications shown in figure 12 are preferable, since they correspond to normal use in the monitor program. If the keyboard was only to be used in this application, it would have been more logical to run all the keys together into one 7 x 4 block; however, the keyboard will often be used as two separate small keyboards for two players. For this reason, the suggested layout is designed for easy 'cutting along the dotted line'.
The only constructional comment regarding the power supply is that *all due care should be taken!* If, due to a fault occurring at a later date, the supply voltage suddenly jumps to way over 5V, several expensive ICs may die a sudden death. For the same reason, the supply voltage must be adjusted to 5V *before* connecting it to the rest of the unit. A drop of laquer on the potentiometer will not only keep it from sliding off position, it will also serve as warning not to touch it at a later date.
The UHF/VHF modulator must be adequately screened. Of course. Modulators should always be mounted in metal boxes. This unit can be powered from the main +5V supply, so the regulator (IC1) can be omitted and the holes in the board for its two outer pins can be bridged with a wire link.

Calibration procedure

With everything neatly built and all interconnections made, and after a final visual check of the wiring, it is now time to switch on. *Note that, as stated earlier, the power supply must already be tested and adjusted to 5V.*
The calibration procedure is simplicity itself. In fact, the modulator contains exactly the same number of adjustment points as the rest of the circuit – two, to be precise!

UHF/VHF modulator
Set P1 to its mid-position and tune the TV set to one of the harmonics of the carrier. When the carrier is picked up, the snow-storm effect on the screen of the TV set will disappear. Turn P2 up to maximum. This completes the initial adjustment. Some 'touching up' will be dealt with later.

Main circuit
Operate the reset and start keys. After correct adjustment, this should cause a blue screen to appear with four yellow letters at the lower left-hand corner.

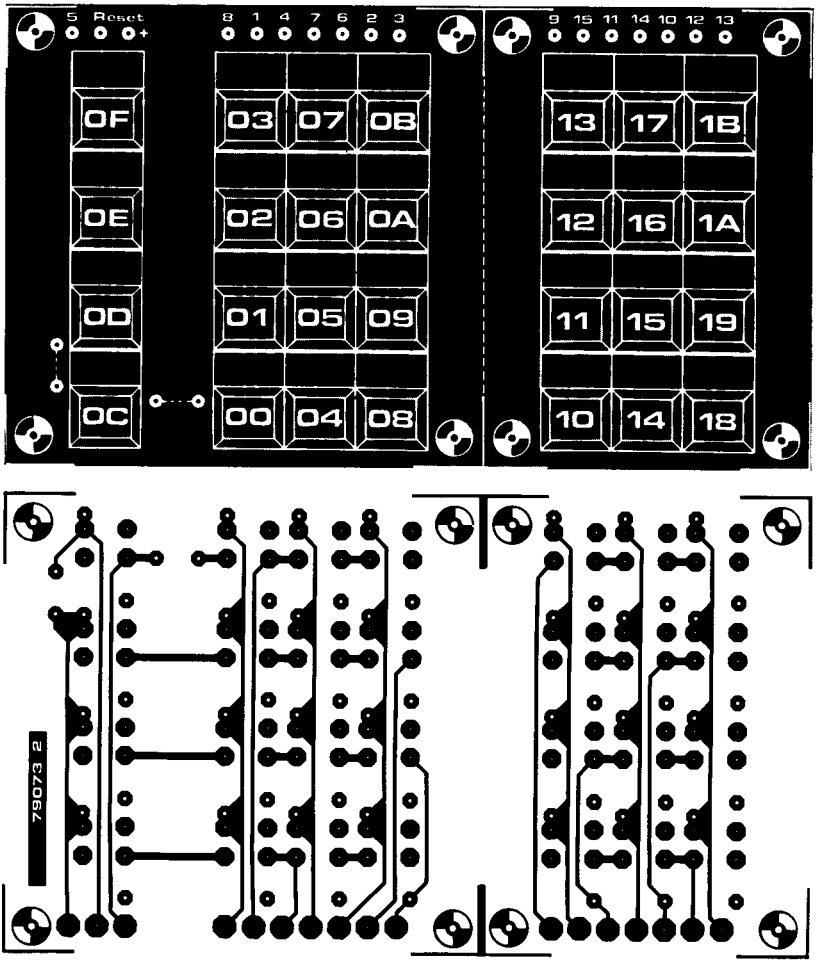


Figure 11. Printed circuit board and component layout for the keyboard (EPS 79073-2). Note that the wire links (shown dotted) should be omitted.

The only adjustment points in the main circuit are P1 and C9 in the crystal oscillator. These can be 'calibrated' by looking at the picture:

- if P1 is incorrectly adjusted, the oscillator will not run at all, in which case no picture will appear. The simplest adjustment procedure is to turn P1 a little bit further than necessary to obtain a picture.
- C9 determines the oscillator frequency, and incorrect adjustment will lead to poor colour or even no colour at all.

Final touches

Having obtained a picture, it becomes a simple matter to adjust for

maximum picture quality:
The TV set is tuned to the sideband that gives the best picture; if tuned to the wrong sideband the picture will tend to appear negative.
If the picture lacks vertical synchronisation (i.e. rolls), or if some local broadcast transmitter interferes with the picture, P1 can be readjusted slightly and the TV set retuned accordingly.
P2 on the modulator board can be used to adjust the contrast.
C9 on the main board influences the colour above all else. Both C9 and P1 should be adjusted for optimum picture quality.

Final notes

The cassette interface should prove suitable in most cases. However, for the odd recorder the output signal level may be too high or too low, in which case the value of R70 can be altered accordingly. In some cases (excuse the pun) problems may also occur if the cassette recorder is placed too close to the TV set. The solution is, of course, simple: move the recorder further away.

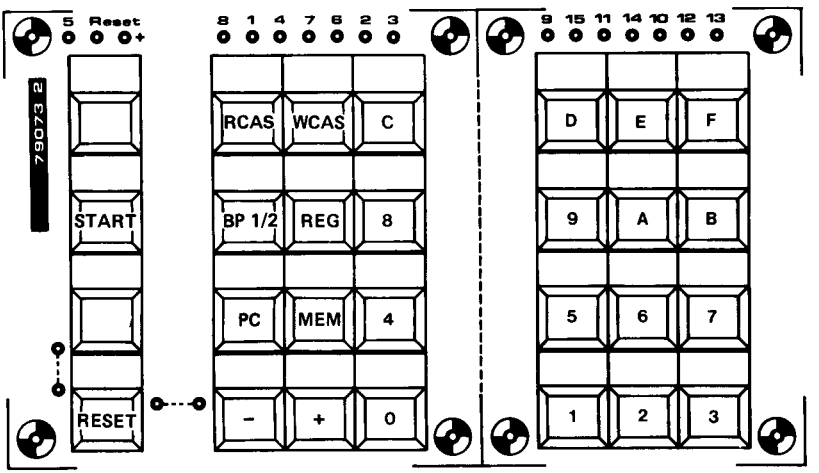


Figure 12. Suggested texts for the key tops.

3

Instructions for use

Having seen, in general terms, what the games computer can do and how it does it, the next step is to get it on the job. Assuming that the complete unit has been built, adjusted and connected up to the TV set as described, it is now time to start typing instructions on the keyboard.

For the moment, the most important 'procedure' to know is how to transfer programs from an ESS record (or tape) to the memory in the computer. After switching on, the keys 'reset' and 'start' are pressed, whereupon the letters 'IIII' should appear at the lower lefthand corner of the screen. The machine is now ready for programming.
Each program on tape or disc is preceded by a so-called 'file number'. This is a single hexadecimal digit (not Ø), and is intended as an aid when looking for one particular program out of several on the same tape. Having operated the reset and start keys, the rest of the procedure is therefore as follows:
- press the 'RCAS' key ('Read Cassette'). The computer will respond by asking for the file number of the program: 'FIL = '.
- type in the desired file number;
- operate the '+' key;
- start the tape at some point before the start of the desired program.

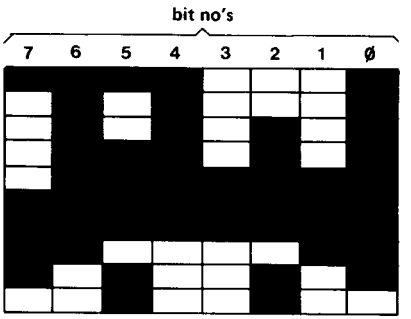
The file number will now appear at the top of the screen ('FIL + 3', for instance) and the machine will start its search for the correct program. If it finds other file numbers first, it will display these on the screen; meanwhile, two dots under the '+' sign will flash rapidly. As soon as the correct file number is located it will proceed to store the following program; this is indicated by the fact that the two dots now flash slowly. When the complete program has been stored, the text 'PC = Ø9ØØ' (for instance) will appear; the four-digit number is the so-called 'start address' for the program, as specified on the tape. As soon as this text appears,
- the '+' key is pressed. This starts the program and, with it, the game!

Should an error occur while the program is being read from the tape, the transfer is stopped and the text 'Ad = ØDØØ' (for instance) will appear. The number signifies the first 'address' in the group of instructions ('block') where the error occurred. In this case, the tape must be wound back and the whole procedure repeated.

None of the other keys will normally be required until one starts writing programs. However, to satisfy initial curiosity, we can run through them quickly:
- WCAS, which stands for 'Write Cassette'. This key is used when storing

programs on tape. The machine will first want to know the first address in the program (BEG =); then the last address (End =); then the so-called start address (Sad =); and finally the file number (FIL =).

- BPI/2: this key is used for inserting so-called break points in a program, as an aid to 'debugging' it.
- REG: this is used for checking or altering the contents of seven 8-bit 'registers' and a 16-bit 'status register' in the CPU.
- PC, for 'program counter'. This key can be used for entering the 'start address' of a program; when a program has been copied from tape, this will normally have been taken care of automatically.
- MEM: the 'memory' key. This is used when storing a 'home-brew' program in the memory by means of the keyboard. 'Storing a home-brew program'. That sounds fascinating, but how do you go about it?



	binary code	hexa-decimal code	(final) address in PVI	(temporary) address in RAM
	11110001	F1	1FØØ	ØAØØ
	Ø1Ø1ØØØ1	51	1FØ1	ØAØ1
	Ø1Ø1Ø1Ø1	55	1FØ2	ØAØ2
	Ø1110101	75	1FØ3	ØAØ3
	Ø1111111	7F	1FØ4	ØAØ4
	11111111	FF	1FØ5	ØAØ5
	11111111	FF	1FØ6	ØAØ6
	11ØØØØ11	C3	1FØ7	ØAØ7
	1Ø1ØØ1Ø1	A5	1FØ8	ØAØ8
	ØØ1ØØ1ØØ	24	1FØ9	ØAØ9
Horizontal position of object	1ØØØØØØØ	80	1FØA	ØAØA
Horizontal position of duplicate (off screen)	11111111	FF	1FØB	ØAØB
Vertical position of object	Ø1ØØ1111	4F	1FØC	ØAØC
Vertical offset of duplicate (off screen)	11111111	FF	1FØD	ØAØD
Size of object	11ØØØØØØ	C0	1FCØ	ØACØ
Colour of object (red)	ØØØ11ØØØ	18	1FC1	ØAC1

82901-13

Figure 13. Painting by numbers!

Painting by numbers

Take the steam engine in figure 1. How do you go about getting it on the screen?

The picture on the screen is determined by information stored in the Programmable Video Interface (PVI). However, for the purpose of this explanation the RAM (the section of memory in which we can store any desired information) will be used as a go-between. First, however, let us see what information the PVI requires.

Without going into too much detail... Obviously, the shape of the 'object' is the first thing to look at. As stated earlier, any object is built up in an 8 x 10

rectangle: 8 horizontal and 10 vertical divisions. At this point, computer jargon becomes useful: it can be used as shorthand. A 'bit' is a single unit of information: voltage present or absent on a particular wire; for digital systems, '1' or 'Ø' respectively. A certain number of bits (the number of bits depending on the system in question) is called a 'byte'.

For the 'games computer' a byte consists of 8 bits; in other words, all information along the data bus is passed in groups of 8 bits, each being Ø or 1.

Each row in the 8 x 10 rectangle for one object consists of 8 squares. Each of these squares corresponds to one 'bit', so a complete row is one 'byte'. Therefore, 10 bytes are required for one object. If we take a closer look at the locomotive (figure 13), the top row consists of four squares, followed by a three-square gap and one final square. The bit corresponding to each filled-in square is '1', so the first byte is 1111ØØØ1.

This 'binary number' can also be written in shorthand. Reading from right to left, the first bit counts for 1, the second for 2, the third for 4 and the fourth for 8. These four bits, added, can therefore correspond to any number between 0 and 15. So-called hexadecimal numbers also run from 0 to 15: the numbers from 0 to 9 are followed by the first letters of the alphabet – A is 10, B is 11, and so on. Reading the first byte as two groups of four bits, we can therefore use hexadecimal numbers as short-hand: 1111 = F and ØØØ1 = 1, so the first byte is F1.

Using this system, the complete picture can be described as 'F1, 51, 55, etc.', as shown in figure 13. So far so good: we can now describe any object by means of ten (hexadecimal) numbers. However, these numbers must be stored in a section of the PVI that will be 'addressed' by the CPU when it decides that an object should be displayed. The 'addresses' within the PVI (and anywhere else, for that matter) are also given in hexadecimal numbers. The ten bytes corresponding to the shape of the first object are stored in the PVI at addresses 1FØØ through 1FØ9. The following four addresses (1FØA through 1FØD) are used to store the horizontal and vertical position coordinates of the object and any desired duplicate(s); two further addresses (1FØE and 1FØF) are used as 'scratch-pad memory' (for the moment, they are best forgotten) and the shape of the second object is stored at addresses 1F1Ø through 1F19. And so on.

The position of the object has been mentioned briefly. This is determined, quite simply, by the distance from the top left-hand corner of the screen to the top left-hand corner of the 8 x 10 rectangle. The picture is divided into 227 horizontal and 252 vertical units; the horizontal and vertical displacement of the object are determined by the numbers stored in bytes 1FØA and 1FØC respectively (for the first object) – the horizontal coordinate and the vertical offset (with respect to the main object) of a duplicate are stored in bytes 1FØB and 1FØD.

The first two bits in byte 1FCØ determine the size of the first object (the third and fourth bit – from the left – are for the second object, and so on). 'ØØ' is the smallest size, 'Ø1' is twice as large, '1Ø' is x 4 and '11' is x 8.

Finally, the colour of the objects is stored in bytes 1FC1 and 1FC2. Reading from right to left, the first three bits determine the presence or absence of

Table 3.

This program can be used to produce an 'object' on the screen. It is entered via the keyboard as follows:

keystroke(s)	display on screen	explanation
reset, start	IIII	start 'monitor' program
MEM	Ad=	user: 'I want to store a program in memory'. Computer: 'What is the first address?'
Ø; 9, Ø; Ø	Ad= Ø9ØØ	The first address is Ø9ØØ.
+	Ø9ØØ xx	Roger. What data? ('xx' is the data already stored here).
7; 6	Ø9ØØ 76	The first 'data byte' is 76.
2; Ø	Ø9Ø1 2Ø	The second is 2Ø.
0; 5	Ø9Ø2 Ø5	etcetera.

'Typing errors' are easily corrected: the '–' key can be used to 'step back'. For instance, if the first two data bytes were keyed in as:

7; 6	Ø9ØØ 76	So far, so good.
1; Ø	Ø9Ø1 1Ø	Wrong!
–	Ø9ØØ 76	Back step.
+	Ø9Ø1 1Ø	
2; Ø	Ø9Ø1 2Ø	That's what I meant . . .
Ø; 5	Ø9Ø2 Ø5	etcetera

The complete program is as follows:

address	data	explanation
Ø9ØØ	**7620**	
Ø9Ø2	**Ø5CA**	Load the PVI with the data stored in the RAM under addresses ØA00 through ØACA.
Ø9Ø4	**Ø6CA**	
Ø9Ø6	**ØD4AØØ**	
Ø9Ø9	**CD7FØØ**	
Ø9ØC	**FA78**	
Ø9ØE	**ØC1E88**	Return to the 'monitor' program if the '–' key is operated.
Ø911	**4410**	
Ø913	**9979**	
Ø915	**1FØØØØ**	
Ø918	**Ø4ØØ**	Store 'ØØ' in all RAM addresses from ØA00 through ØACA (i.e. erase this section of memory).
Ø91A	**Ø5CA**	
Ø91C	**Ø6CA**	
Ø91E	**CD4AØØ**	
Ø921	**FA7B**	Disable 'score' display (by storing 'impossible' numbers – FF – in the score bytes ØAC8 and ØAC9).
Ø923	**Ø4FF**	
Ø925	**CCØAC8**	
Ø028	**CCØAC9**	Make the screen blue (by storing Ø9 in byte ØAC6).
Ø92B	**Ø4Ø9**	
Ø92D	**CCØAC6**	Determine the size of the object (Ø2 = half-size).
Ø93Ø	**Ø4Ø2**	
Ø932	**CCØACØ**	Go to first section of program (load PVI): start address Ø9ØØ.
Ø935	**IFØ9ØØ**	

Having loaded the program to this point, the PVI can be 'cleaned out' by using the 'erase' program starting at address Ø918 as follows:

keystroke(s)	display	explanation
PC	PC = ØØØØ	'I wish to run a program . . .
Ø; 9; 1; 8	PC = Ø918	. . . starting at address Ø918'
+	(blue)	The program is running.
–	PC = Ø918	Back to the 'monitor' program.

The rest of the program can now be stored, using the same routine as before: 'MEM', followed by the first address (ØA00), '+' and then the 'data' (F1, 51, etc.).

(Table 3, continued)

address	data	explanation
ØAØØ	**F1**	
ØAØ1	**51**	
ØAØ2	**55**	This data describes the shape
ØAØ3	**75**	of the first object – the
ØAØ4	**7F**	steam engine (see figure 3).
ØAØ5	**FF**	
ØAØ6	**FF**	
ØAØ7	**C3**	
ØAØ8	**A5**	
ØAØ9	**24**	
ØAØA	**8Ø**	Horizontal position of object.
ØAØB	**FF**	Horizontal position of duplicate(s) (off screen).
ØAØC	**4F**	Vertical position of object.
ØAØD	**FF**	Vertical offset of duplicate(s).
ØACØ	**CØ**	size of object
ØAC1	**18**	colour

This object can now be displayed on the screen, by using the 'load PVI' program starting at address Ø9ØØ. Keystrokes, as before: PC; Ø; 9; Ø; Ø; +.

After operating the '–' and 'MEM' keys, up to three more objects can be stored (starting at addresses ØA1Ø, ØA2Ø and ØA4Ø); a background can be added (from address ØA8Ø); the sizes and colours of the objects can be varied (ØACØ...ØAC2); background and screen colours can be chosen (address ØAC6 – note that the background will only appear if the second data 'number' is 8...F); finally, data in address ØAC7 determines the sound. Note that it is possible to step to and fro through the memory without changing data by operating the '+' and '–' keys. If complete chaos results, the 'erase' program can be run – from address Ø918. Have fun!

each of the primary colours in object 2; the next three bits determine the colour in the first object; the two left-hand bits are unused.

Now, let's cut a long story short. We know what information the PVI needs; let's put it there and see what happens. Utilising the 'monitor software', present in the ROM, the necessary information can be stored in the RAM – after which the CPU can be instructed to transfer this information to the PVI, to produce the required picture on the screen.

The information is typed in via the keyboard, as shown in table 3. Basically, the complete section of memory between addresses ØAØØ and ØACA is first erased, after which the correct information for the desired object is stored in the Random Access Memory (starting at address ØAØØ). Finally, a brief program is started that causes the CPU to transfer the requisite information from the RAM to the PVI – and the picture appears on the screen.

The explanations given in the table are intended as a challenge: by playing around with the shape, colour, position and size information all sorts of other objects can be created. Have fun!

Background

The background is programmed in much the same way as an object. It was mentioned earlier that the background consists of 160 squares: 10 rows of 16 squares each. Each square is defined by its top and left-hand edges, so that there are 320 edges in all; each of these can be selected independently, by setting a corresponding bit in the PVI to '1'. The 16 edges in one row correspond to 16 bits – two 'bytes', in other words. For one row of squares, two bytes are therefore required for the upper edges and two further bytes for the left-hand sides; the 10 rows thus require 40 bytes in all. As illustrated in figure 14, the addresses of these bytes run from 1F8Ø and 1F81 for the first row of 'upper edges' to 1FA6 and 1FA7 for the last row of 'left-hand sides'. As when 'programming' for an object, storing FF (= 1111 1111) in both the first and second bytes (1F8Ø and 1F81) causes the whole first row of 'upper edges' to appear on the screen. Similarly, 'FF' and '2A' in the third and fourth byte, respectively, will produce the pattern of left-hand edges shown.

Unlike the situation when displaying objects, the size and position of the background are, of course, fixed. The size of the individual lines can, however, be varied. Starting from the top of the picture, one bit is used to determine whether the first row of 'top edges' appears as full-width lines or whether each 'edge' is shortened to a dot. A second and third bit can similarly widen the upper and lower halves of the left-hand edges to fill the adjoining square. The next row of squares is dealt with in the same way, using three more bits. Finally, two further bits can be used to widen all the

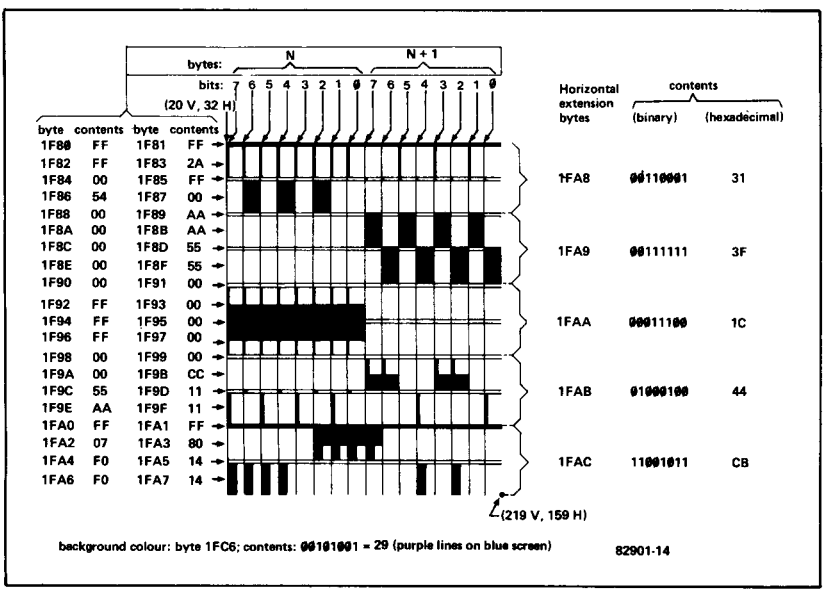


Figure 14. The background can be constructed from a wide variety of horizontal and vertical lines, squares and dots.

lines (top edges and left-hand sides) in the first two rows, to either ¼ or ½ the width of a square.
To sum it up: 8 bits – one byte – determine the width of all the lines in the first two rows. Four further bytes take care of the other four pairs of rows; the addresses of these five bytes run from 1FA8 to 1FAC.
The various possibilities are shown in figure 14. The information given in hexadecimal numbers to the left of the drawing determines which lines appear; it is stored under the 40 corresponding addresses. To the right of the picture, the addresses and contents of the five 'horizontal extension bytes' are listed.
One final point remains to be discussed, before the background can be conjured up on the screen: the colour. One byte in the PVI takes care of this (address: 1FC6). From right to left, the first three bits determine the presence or absence of the three primary colours in the general background between the lines ('screen'); the fourth bit turns the background lines on and off ('background enable') – note that, unless this bit is 1, no background at all will appear! The fifth through seventh bits determine the colour of the lines in the background; the last bit is unused.
Using the same principles outlined earlier (table 3, in particular) it is now possible to produce any desired background pattern, by entering the necessary information into the PVI (again using the RAM as go-between).

What's the score?

Displaying the score on the screen is simplicity itself. As mentioned earlier, the score consists of four digits. The first two are stored in the PVI under

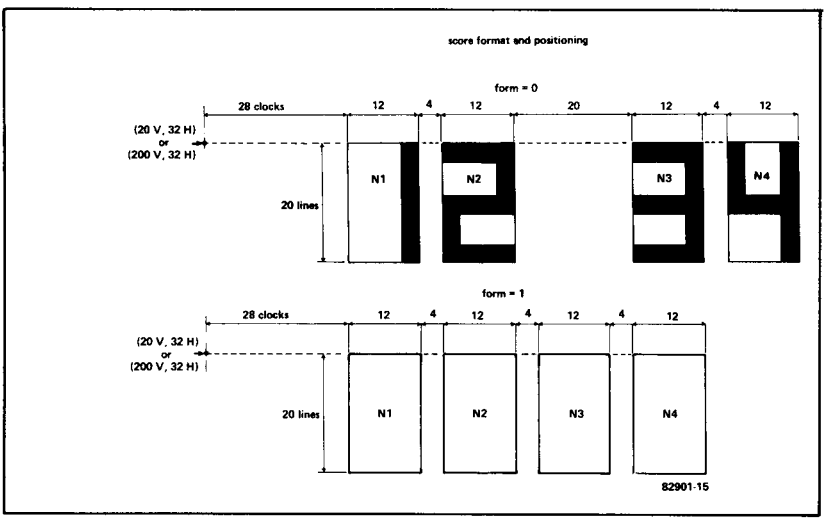


Figure 15. The score display: two groups of two digits or one group of four.

address 1FC8 (four bits for each digit); the second pair in byte 1FC9.
The only other points to determine are the position and type of display required. In byte 1FC3, the extreme right-hand bit determines the position: Ø for the top of the screen and 1 for the bottom. The second bit in the same byte determines the score format. As shown in figure 15, a Ø causes the score to be displayed as two two-digit numbers; a single four-digit number is obtained by storing a 1 at this bit.
No other possibilities for position or size are available, and the colour is the 'opposite' to that of the screen. In most cases, these limitations do not pose too much of a problem. However, if more flexibility is required, it is of course quite possible to use one or two of the 'objects' to create a score display of any size, shape or colour and in any position!

Collisions

The score and sound effects will depend to a large extent on collisions. When an object hits the background, this is detected and a bit corresponding to that object is set to 1. For the four objects, the four left-hand bits in byte 1FCA are used for this purpose. Similarly, the six right-hand bits in byte 1FCB are used to signal inter-object collisions.
The CPU can 'read' these two bytes as required, and use the information to update the score, change the direction of travel of an object, add sound effects, and so on.

Sound effects

The only sound produced by the basic unit is a square-wave. The frequency is determined by an 8-bit number, stored in byte 1FC7 in the PVI. If the number is zero (ØØ), no sound is produced; otherwise, the output frequency is determined by the value stored. To be precise, the frequency equals

$$f_O = \frac{7874}{n + 1},$$

where n is the value stored in the 'sound byte'. For instance, if n = Ø1, the output frequency will be approximately 4kHz; as higher numbers are stored the frequency decreases, until for the highest value (FF) the output becomes approximately 30Hz.
At a later date, other sound effects can be added if required. The extension board, to be described later, contains two programmable sound effect generators.

4

TV games

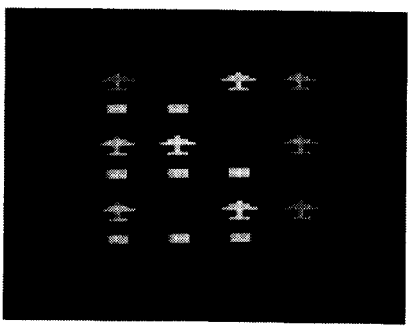
over 20 Kbytes on one tape

In general, a TV games computer that can only play three or four games soon loses its charm. Fortunately, the Elektor version has the saving grace that it makes it relatively easy to develop your own (games) programs. Even so, there is a distinct need for ready-made software. The ESS cassettes fulfil this need, by providing 15 programs each! In this article, we will give some idea of the programs available on the oldest of these (ESS 007).

An advantage of using a tape is that it provides room for a large number of programs. An obvious choice, given the file numbering system of the TV games computer, is fifteen programs – corresponding to files 1...F. The procedure for loading a program into the computer was described extensively at the beginning of the previous chapter. We will not give the rules for each game here, since these are included with the tape.

File 1: Mastermind

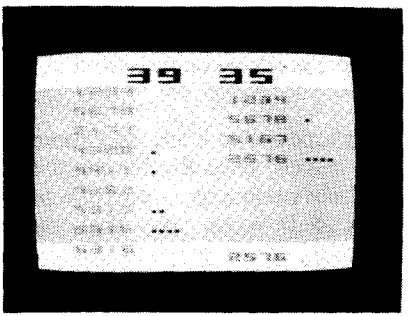
The object of this game is well-known: guess an unknown (colour) code, on the basis of very limited 'feedback'. In this version, purple squares indicate how many correct colours are correctly positioned, and white squares indicate the remaining correct colours in the wrong position. All eight colours are used (black included!), and the same colour may appear more than once in a code.
The computer plays the part of the 'passive' player: it sets up a code, and displays the result of each of your tries.



File 2: Code breaker

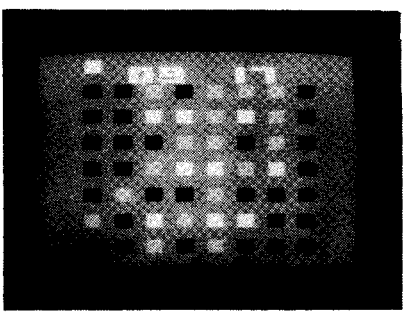
This is the same principle as Mastermind, but the code consists of digits or shapes instead of colours. Furthermore, it is also possible for two players to play simultaneously (each with a different code!), as shown in the photo. Each player's score, and the total number of games played, are displayed at the top of the screen. In all, 24 different variations of the game can be played with the same program.

An interesting point for 'home programmers' to note: when this program is running, the shapes' data (for digits or random shapes) is stored in the monitor RAM area from 0800 on! This is quite permissible, provided no monitor routines are used in the program. The only RAM data that should not be altered is that contained from address 08B9 to 08BF – among other things, this includes the interrupt vector that points to address 0903!



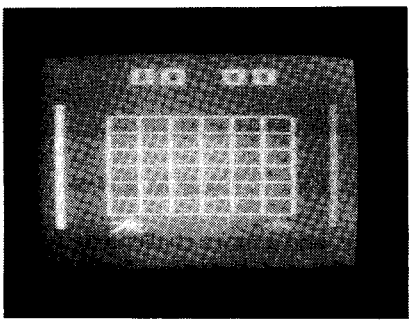
File 3: Reversi

Basically, the object of this game is to capture as many of the opponent's pieces as possible. To do this, you place a piece on the board in such a way that one or more of the opponent's pieces are 'trapped' between two of yours. For example, in the situation shown in the photo a grey square could be placed in the top left-hand corner (second row, second column). This would trap the two squares to its right, so that these can be captured. When captured, squares change to your colour – they are not removed from the board.



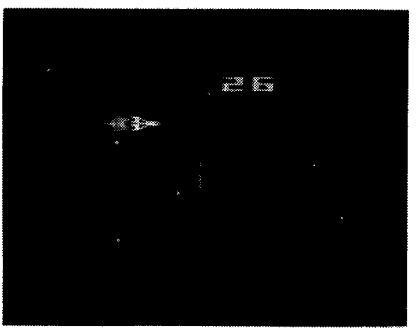
File 4: Amazone

Each 'Amazone' has the capabilities of both Queen and Knight in chess: it attacks squares along horizontal, vertical and diagonal lines, and also any square that can be reached by two horizontal moves followed by one vertical, and vice versa. The players alternately place their Amazone on a square which is not under attack, and which has not been used earlier on in the game. A player loses when there is no legal square left to move, or when he runs out of time.
In this version, the time display consists of the two vertical bars at the left and right of the screen; the score is indicated at the top of the screen. For practice, the game can be played against the computer.



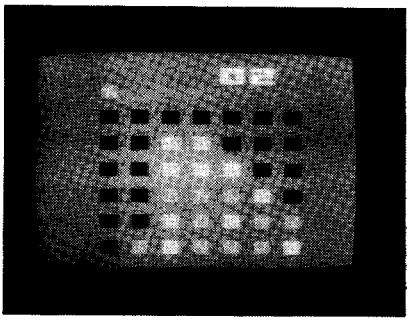
File 5: Space shoot-out

This is an updated version of a game that will be used as an example in a later chapter. The object is, quite simply, to shoot down the rocket before your time runs out.
A joystick calibration routine has been included, so that the program should run without problems on any TV games computer. Furthermore, it is now an easy matter to modify the time limit and the 'proficiency level': the accuracy required to shoot down the rocket can be varied in three steps from 'beginner's luck' to 'pinpoint'.



File 6: Four in a row

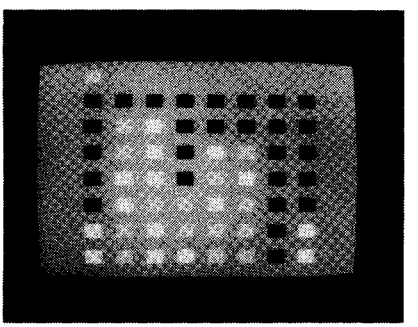
This is a fairly well-known game. The object is to get four of your squares in a row (horizontal, vertical or diagonal).
As those who have already tried this program will know, the computer is an infuriatingly skilled opponent.



File 7: Four in a row

Skill comes with practice, and a more challenging game becomes desirable. This version uses an 8 x 7 board, instead of 7 x 6 as in File 6. The extra row and column make quite a difference! It is distinctly noticeable that the computer needs much more time to work out its moves – no trick here, it really is a question of the time it needs for its calculations.

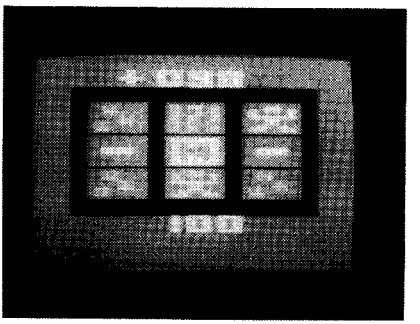
Several readers have asked how the red and green squares are made in this program. Basically, the trick is quite simple. The largest object size is used, and two objects are superimposed for the left-hand half of the screen, the other two being used for the right. Of the two left-hand objects, one is green and the other red; one shape bit corresponds to one square in the picture. Let us assume now, that the left-hand bit (bit 7) is set in the first row of both objects: the result is a black square in the top left-hand corner of the screen. Resetting this bit in the green object leaves a red square and vice versa. Using the odd rows and odd bits only (1, 3, 5, etc.) produces the display shown.



File 8: Jackpot

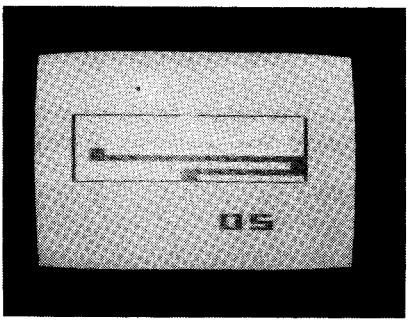
This is the well-known 'one-armed bandit'. The rotating drums, 'hold' and 'brake' options and prize display are all included. There are also two novel features. A car on the centre line always means 'no prize' – cars cost money! Furthermore, the 'score' at the top of the screen shows the total gain or loss (nearly always the latter...).

The program is quite interesting, but also highly complicated. For those who feel like exercising their disassembling skills, data is included from 09DE to 09FF, 0B10 to 0B1F, 0C00 to 0D12 and 0FF6 to 0FFF. The odd unused byte is set to 00.



File 9: Surround

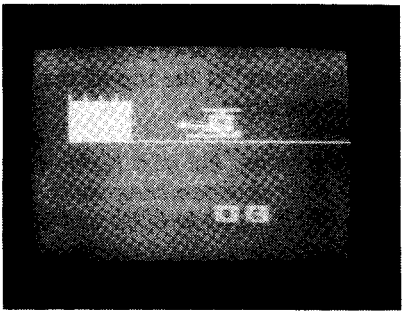
The object of this game is to 'box in' the opponent. A point is lost when a player collides with the background, his opponent or even himself. Since the objects move at quite a speed, fast reactions are required to win!



File A: Shapes

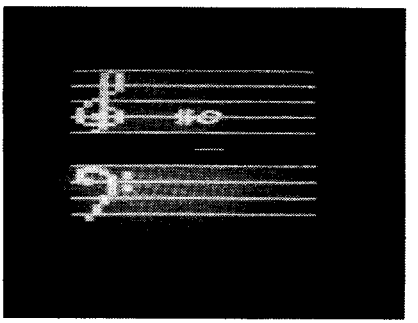
This has proved quite popular with the younger generation... Twenty-five different shapes appear in the 'garage', and then move out to the right. The object is to guess what they represent.

We won't spoil your fun by telling you! The data for the shapes is stored in ten-byte groups from ØAØØ on.



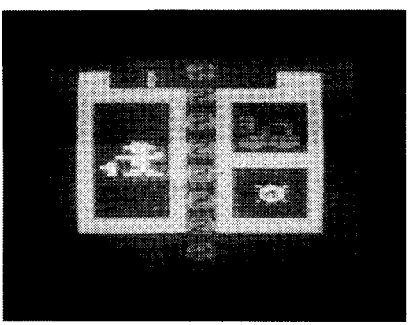
File B: Piano

Any desired melody can be keyed in, provided it fits in a two-octave range. When played back, the corresponding notes are displayed in step with the tune.



File C: PVI programming

This is a rather complicated program, intended as an aid when developing your own games. It provides the possibility of programming objects and



background shapes on the screen, so that the results can be judged more easily. A 'relative address' calculation routine is also included.
This program has proved extremely useful in practice – once you get used to operating the keys!

File D: Disassembler

The object here is quite simple: to make the decoding of an existing program much less laborious! The program to be examined is split up into one-, two- or three-byte instructions, and displayed accordingly.
It should be noted that this program is stored from Ø8CØ to Ø8F6 and from 1F8Ø to 1FAD.

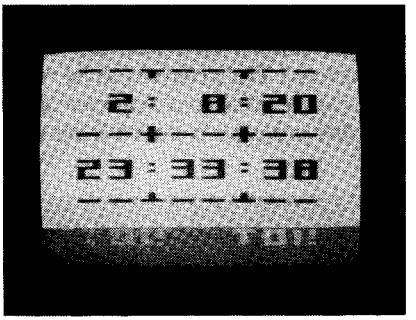
File E: Test patterns

This is exactly what its name implies: a whole series of test patterns for a (colour) TV set. As such, it has proved extremely useful on numerous occasions...

File F: Lotto

This is intended primarily for our readers in Germany. There, a game is played on the national TV networks that is similar to Bingo. You fill in six random numbers from 1 to 49 on a card, and send it in. At the end of the

week, six numbers are 'drawn' in deathly silence – with umpteen million viewers holding their breath... If your numbers come up, you win.
The only mental work involved is in trying to think up six numbers every week. The obvious way to avoid this is to get your home computer to do the job for you...



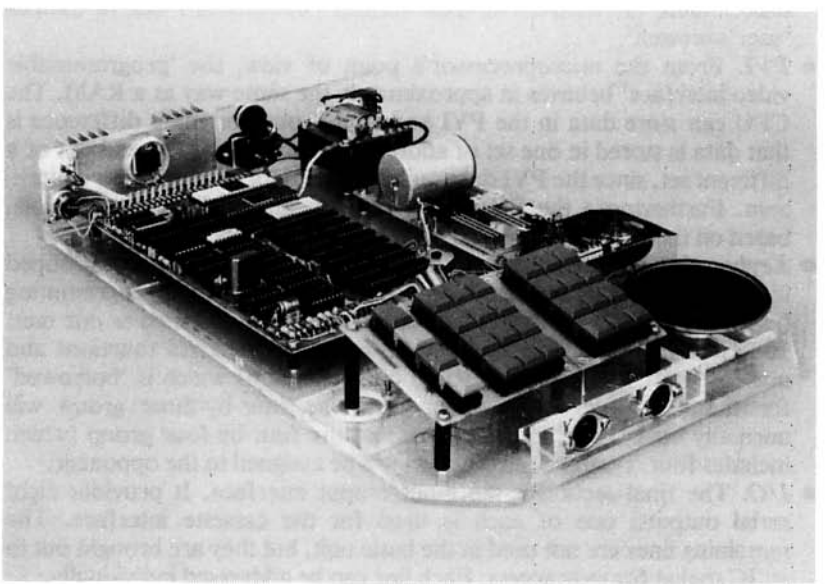
5

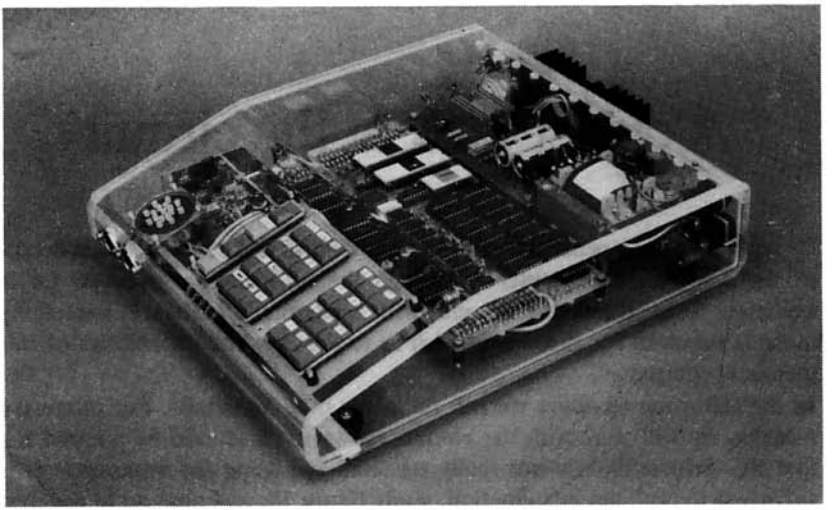
A closer look at the circuit

By now, we have a general idea of the capabilities and constructional details of the TV games computer. Although the information given so far is sufficient to build and use the unit with ready-made software, something more is required for those who wish to exploit the full capabilities of this 'personal computer'.
In the following chapters we will fill in the missing details. For interested readers, we will start with the circuit. However, it should be pointed out that this information is not really essential for using the computer. It is sufficient to read the introduction, study figure 18, and pass on to the next chapter!

The block diagram (figure 16) was already given in the first chapter. However, to save a lot of leafing to and fro, it is repeated here. A closer look at the various sections is now in order.

- *CPU.* The central processing unit is the Signetics type 2650A. This is an 8-bit microprocessor with a 15-bit address bus; however, in this design only 13 address bits are used – corresponding to 8K instead of the full





32K. This has the advantage that data- and address bus do not need buffering.

- *ROM.* This section contains the control programs (monitor software), 2K bytes in all. A single 2K ROM or EPROM is used, type number 2616 or 2716 respectively.
- *RAM.* The basic unit contains a 2K random access memory – sufficient for reasonably long programs. Note, however, that the first ¼K is used by the monitor program, so that it is only available to the user when the monitor program is not in use at all. To be more precise, Ø8ØØ... Ø8BF are used as 'monitor scratch'; Ø8CØ...Ø8FF can be used as 'user's scratch'.
- *PVI.* From the microprocessor's point of view, the 'programmable video interface' behaves in approximately the same way as a RAM. The CPU can store data in the PVI and read it out; the main difference is that data is stored in one set of addresses and other data is read out of a different set, since the PVI does quite a bit of data manipulation on its own. Furthermore the PVI provides video and audio output signals, based on the data stored; it also has two analog inputs.
- *Keyboard (interface).* A 28-key hexadecimal keyboard is used, grouped in one four-by-four and one four-by-three matrix. During programming (using the monitor software) the complete keyboard is used as one unit: 16 keys for hexadecimal numbers, 8 for special control functions and one for 'start'. Three keys remain unused, one of which is 'borrowed' for the reset function. During play, the four-by-three group will normally be assigned to one player and the four-by-four group (which includes four 'system control' keys) will be assigned to the opponent.
- *I/O.* The final section is the input/output interface. It provides eight serial outputs; one of each is used for the cassette interface. The remaining lines are not used in the basic unit, but they are brought out to an IC socket for easy access. Each line can be addressed individually.

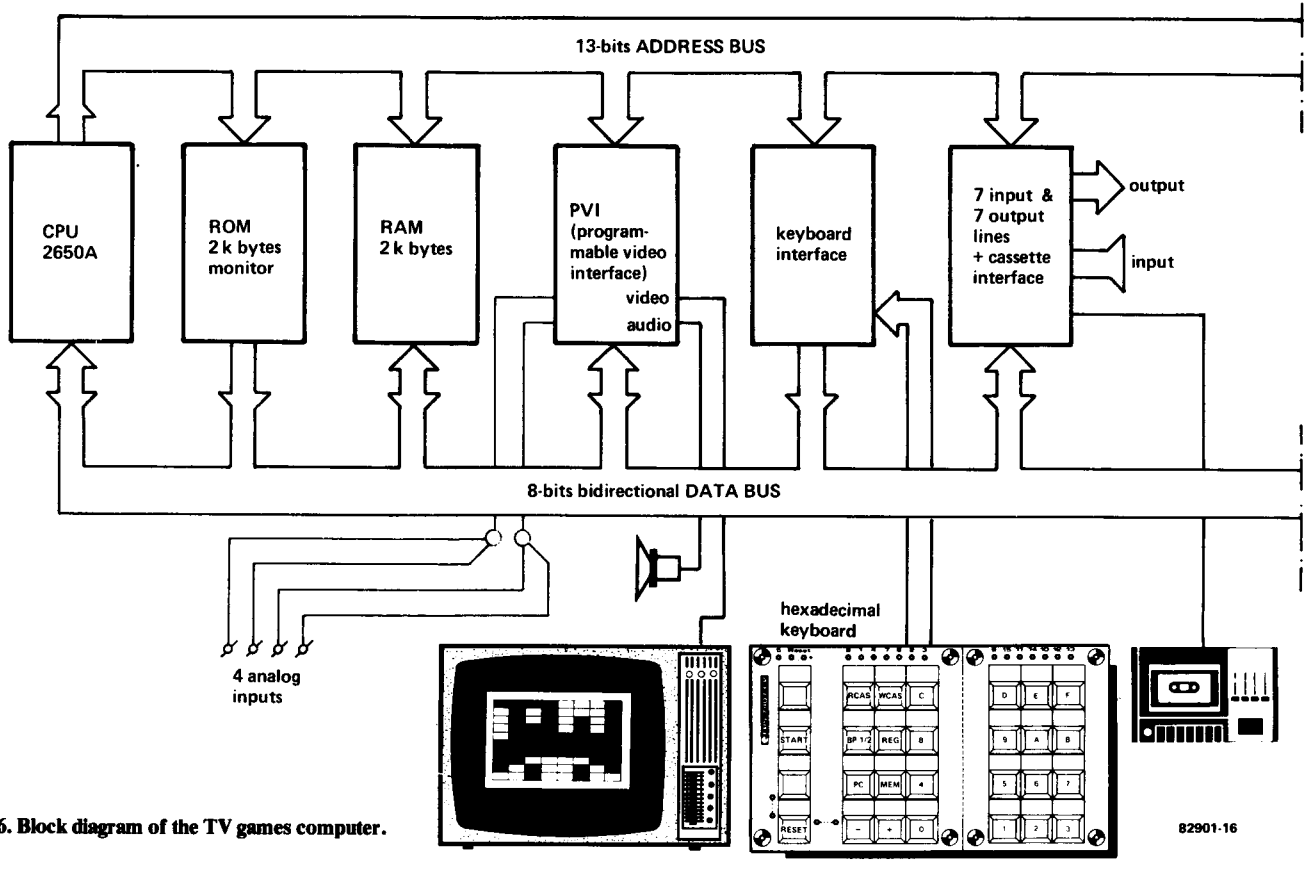


Figure 16. Block diagram of the TV games computer.

Hardware

The circuit diagram of the 2650 computer is repeated here, as figure 17. Starting at the left-hand end, the overstretched rectangle represents IC1: the CPU, type 2650A. The 8-bit data bus and 13-bit address bus run along the top. It should be noted that the 2650A has a 15-bit address bus; however, in this design the two MSBs of the address remain unused (A13 and A14, pins 19 and 18 respectively). The data and address bus can remain unbuffered.

The 13 address bits correspond to 8K addresses subdivided into four groups of 2K each. Which brings us to the address decoder. The two highest address bits used (A12 and A11) are fed to (one half of) IC6, causing one of its four corresponding outputs (1yØ... 1y3) to become active. The first of these outputs is used as a 'select' signal for the ROM (IC2). This Read Only Memory occupies the full 2K area. It contains the 'monitor' program, which will be discussed in greater detail further on.

The second output from IC6 activates a 2K RAM, consisting of 16 Random Access Memory ICs type 2112. To be more precise, the output from IC6 'enables' a further address decoder, IC7. This IC is connected to the following three address bits (A10 ... A8); when enabled, it subdivides the 2K RAM area into eight subsections of ¼K. Each of these subsections corresponds to two 2112 ICs, both taking care of half the data bits.

The third output from IC6 remains unused for the present. If required at a later date, it can be used for further extension of the RAM area: the circuit described above (IC7 and IC13... IC24) can be duplicated to provide a further 1½K of RAM. (Note that the last ½K should remain unused).

Finally, the fourth 2K address area as decoded by IC6 is used for the video interface (PVI), keyboard and other input/output units. In practice, IC6 has little to do with this: the PVI contains its own address decoder, and it is fully capable of looking after itself – and the keyboard, for that matter. However, the second half of IC6 *is* used for a subdivision of part of this area into four further 'sub-subdivisions'. One of these is used for the input gate (IC8) and one for the output gate (IC9); the remaining two are available for extension of the input/output capabilities as and when required. The cassette interface is connected to one input to IC8 and one output from IC9; only a minimum of hardware is required, thanks to the fact that the (monitor) software involved is highly sophisticated.

So much for the general subdivision of the address area. A more detailed breakdown is given in figure 18.

The sections described so far can be considered as the 'microcomputer proper', suitable for any number of applications. In this particular application, the ultimate goal is to create an interesting display on a TV screen. Which brings us to the second half of the circuit, with its centrepiece: the Programmable Video Interface.

The PVI

The PVI provides an audio output signal and most of the information required for the final video signal. The audio signal requires little further

hardware: a single (Darlington) transistor and a miniature loudspeaker.
The video signal is a more complicated matter. Several timing signals are required, and these are provided by the Universal Sync Generator, IC4. This IC (type 2621) was designed for use in conjunction with the PVI – although it can prove quite useful in other applications. It receives a clock signal, via a divider chain consisting of IC32, IC35 and IC36, from a crystal generator using one inverter in IC31. The generator frequency (8.867 MHz) is exactly twice the frequency of the colour subcarrier. The USG provides several outputs, derived from this clock signal. Two clock outputs, PRCLK (processor clock) and PCLK (positive clock), are fed to the processor and PVI respectively; a 'vertical reset' signal, VRST, goes to the 'sense' input of the CPU and to the PVI; a horizontal reset signal (HRST) is only used by the PVI. The 'composite blanking' output, CBLNK, controls a gating circuit consisting of several inverters and EXORs contained in IC29, IC30 and IC40. The three 'colour' outputs from the PVI are passed through this gating circuit, inverted or suppressed there as the timing signals dictate, and applied to the resistive summing network (R54... R61) to provide the 'intensity' signal. The remaining outputs from the universal sync generator provide timing signals for the sync pulses, PAL switch and colour burst. Without going into great detail about the composite video signal required for a PAL colour TV set, it is virtually impossible to give a clear explanation of the remainder of the circuit. Suffice it to say that IC33, IC34, IC38 and IC39 receive both direct and delayed signals from the clock generator, together with the three colour signals and a signal derived from the colour burst and PAL switch output from the USG, and that from all these inputs they derive a single output: the modulated colour subcarrier; and, finally, that the collection of gates and inverters at the bottom right-hand corner of the circuit diagram take care of the sync signals and colour burst. It is best to draw a veil over the details – the important thing is that, believe it or not, it works...
Back, now, to the PVI. So far, only its outputs have been mentioned briefly. Obviously, the colour and audio outputs must ultimately be determined by the CPU. Basically, the CPU is in charge of the total interplay of all the components in the system, while the PVI is concerned with the details of putting a picture on the screen. If complete chaos is to be avoided, good communication between these two units must be assured. It is hardly surprising, therefore, that the PVI is connected to virtually all lines on the address and data busses. The address decoder in the PVI is used not only for its 'internal organs' but also to control some further external inputs. The analog multiplexer, IC10, selects one or other pair of potentiometers in the two joystick controls (as determined by the Flag output from the CPU) and feeds the corresponding analog signals to the PVI; analog-to-digital conversion is done by the PVI itself. The addressing of the keyboard interface (IC11 and IC12) is also under (partial) control of an address decoder in the PVI.
Once again, we have side-stepped the issue: the PVI. The reason for this is simple: although all the hardware described above is centered around the PVI, a description of that unit itself would plunge us into the middle of the

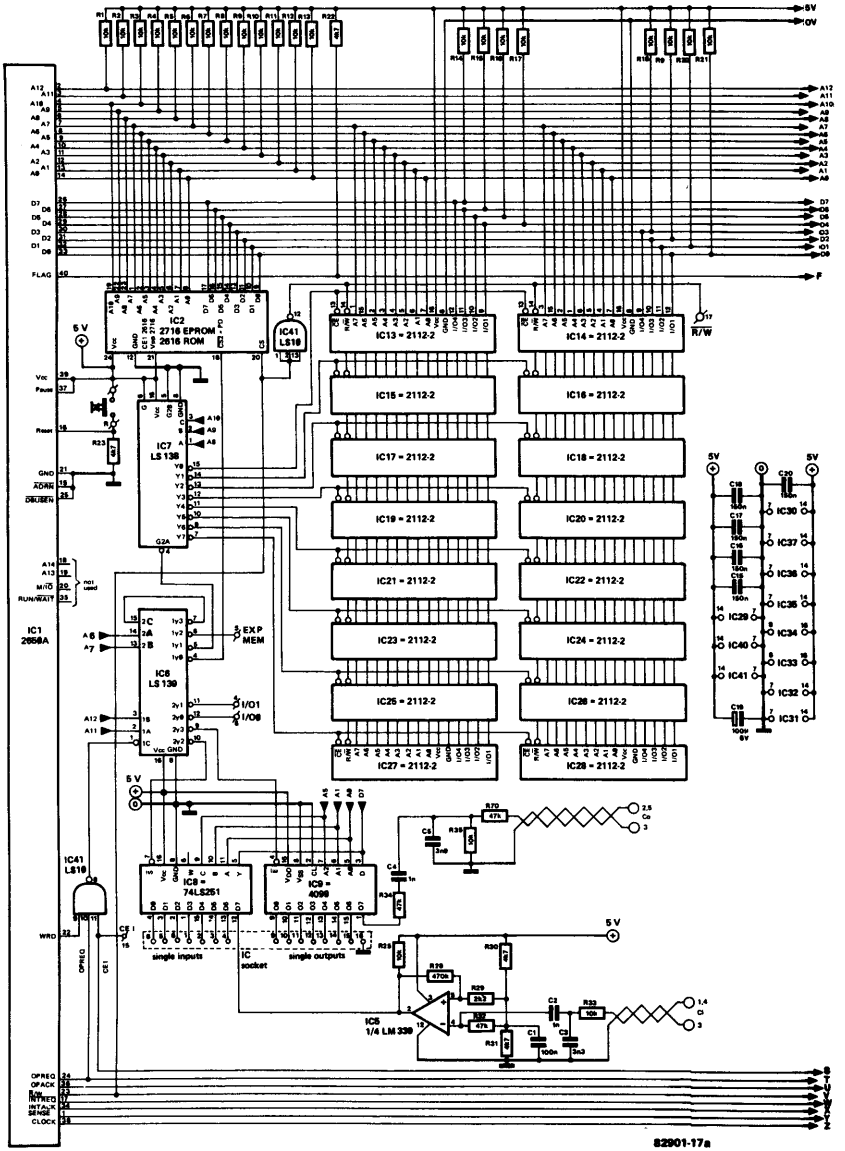
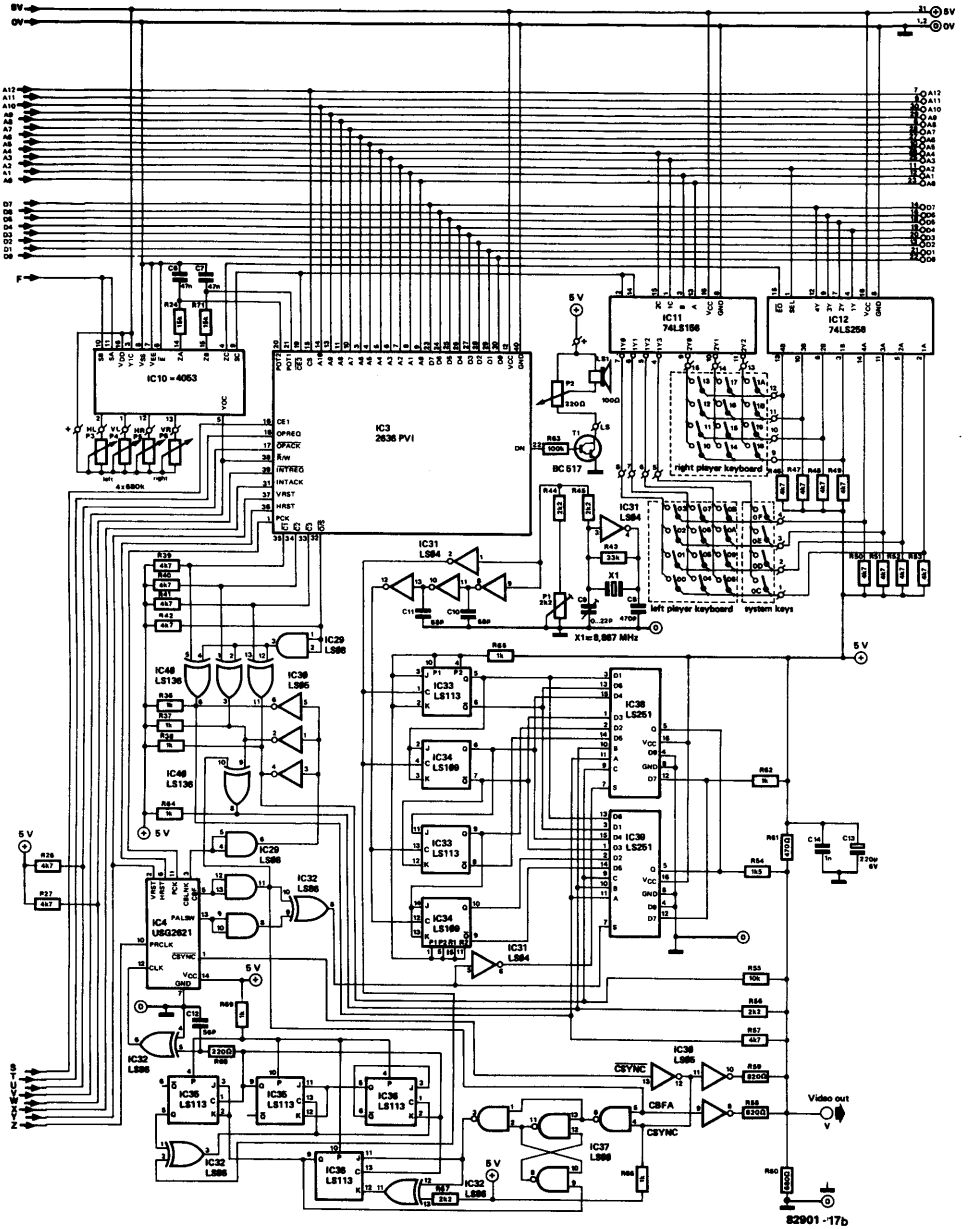


Figure 17. The main circuit of the games computer.


5V
0V
IC10 = 4053
IC3
2636 PVI
IC11
74LS166
IC12
74LS258
Right player keyboard
left player keyboard
system keys
BC 517
X1=8,867 MHz
IC4
U6G2621
CSYNC
CBFA
Video out
82901 -17b

software. And that is to come later. First, one final point remains to be discussed regarding the hardware: the external connections. Most of these have been mentioned already, or else are fairly obvious. The address and data busses, together with some of the more important control signals, are brought out on a 31-pin connector. The corresponding pin numbers are all clearly shown in figure 17. All the remaining connections should be clear: a video output, an audio output, supply and keyboard connections, the cassette interface and, finally, the seven inputs and seven outputs from the serial I/O selectors that are brought out via a 14-pin IC socket as shown.

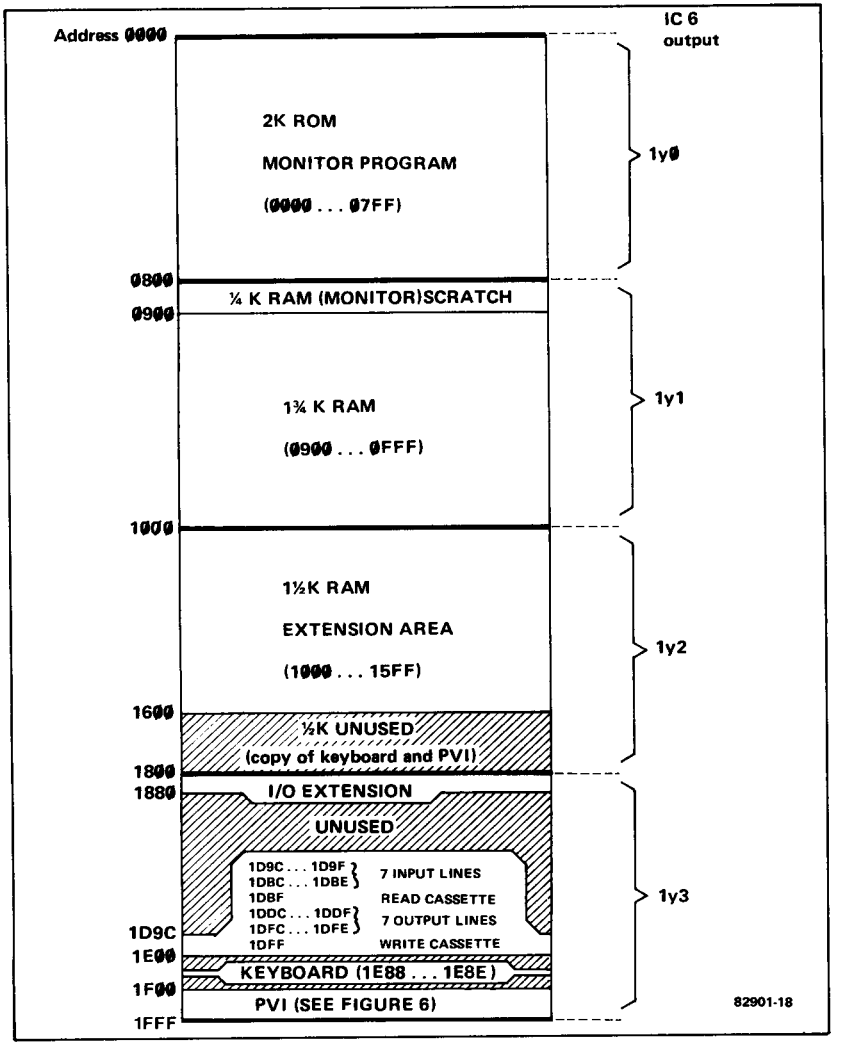


Figure 18. Each section in the unit has its own address, determined by the 'hardware'. The 'memory map' given here shows which addresses correspond to the various sections.

6

A closer look at the software

No matter how sophisticated the hardware, the final results in any microcomputer system depend on the software. Programs, in other words. When developing programs, good 'monitor' software supplied by the manufacturer can be a great help... Let us take a closer look at the monitor program in this unit.

After switching on, operating the reset and start keys starts the monitor program. The letters 'IIII' appear at the lower left of the screen. The keys now have the functions shown in figure 19. The following brief explanations should suffice to illustrate the possibilities of the monitor software.

MEM:

The MEMory key is used for loading programs into the memory by means of the keyboard. When this is operated the text 'Ad =' will appear at the lower left-hand corner of the screen. A four-digit address should now be entered, using the hexadecimal keys (0...F). If any keying errors are made at this point, the address can simply be typed in again. Once the correct four digits are on the screen, the '+' key is operated. The address will shift to the left-hand side of the screen; simultaneously, the (data) contents of that address will appear to the right as a two-digit number. There are now two possibilities:

1. The data at this memory location must be changed (obviously, this is only possible if the address is within the RAM area). To do this, the new data is entered from the keyboard (two digits). The next address will now automatically be addressed, so that the next data byte can be keyed in, and so on. This means that when storing a program, only the first address must be keyed in; from then on, only the data bytes are entered. This saves a lot of hard work. Typing errors are corrected by first operating the '−' key, then the '+' key, and finally keying in the correct data.
2. The data are to remain unchanged. In this case, the contents of the next address can be called up by operating the '+' key, or the data in the preceding address by using the '−' key.

PC:

Having loaded a program, the 'start address' (in practice this is often the first address) must be loaded into the Program Counter. After operating the 'PC' key, the text 'PC = 0000' will appear on the screen. The start address is now entered from the keyboard (four digits); finally, operating the '+' key loads this address into the program counter and starts the program.

BP 1/2:

Hopefully, a new program will work first time. In practice it rarely does. When trying to locate errors in the program, so-called Break Points can prove useful: When a break point is reached, the processor will jump back to the monitor program. Operating the 'BP 1/2' key causes the address of the first break point to appear on the screen; for instance 'bP1 = 0000'. If the key is depressed again, the position of the second break point will appear. Immediately after the first or second break point address has appeared, a new address can be entered from the keyboard, followed by the '+' key. After placing one or two break points, the program can be started in the usual way ('PC' key and start address); if the processor reaches the first break point it will jump back to the monitor program and display the text 'br1 = xxxx', where 'xxxx' is the address of the break point. It should be noted that the break point address should always correspond to the *first* byte of an instruction, since this instruction will be replaced by a jump instruction (ZBRR-indirect, to be precise). After jumping back to the monitor program, the original instruction is automatically restored at that address, so that the original program is not affected.

REG:

Another useful routine when developing and trouble-shooting programs. When the REGister key is operated, the text 'r0 = xx' appears on the screen, where 'xx' is the data contained in register 0. The 2650 has seven general-purpose 8-bit registers (R0... R6). The contents of each can now be called up by repeated operation of the '+' key: new data can be entered, as required – it is stored in the register by operating the '+' or '–' key. The contents of 'R7' and 'R8' can also be displayed; these represent the lower and upper byte of the 16-bit status register in the CPU. The significance of the various bits in the status register is given in figure 20. A full explanation will be given later; the effect of the various instructions on the status register is shown in the instruction set (figure 22). For the moment, however, we are more interested in the PVI than in the CPU.

WCAS:

Having written a program, the Write CASsette routine can be used to store it on tape. When this key is operated, the processor will first ask for the first address in the memory area to be copied: 'BEG ='. This address (four digits) is entered, followed by '+', whereupon the processor will request the final address: 'End ='. Having entered this, and operated the '+' key, a start address may also be stored ('Sad ='). Finally, the program must be identified by a 'file number' ('FIL ='), consisting of a single hexadecimal digit (not 0). As soon as the '+' key is now operated, the program data will be transmitted to the cassette recorder. During transmission, the file number is displayed at the top of the screen; when the complete program has been transmitted, the first, last and start addresses will be displayed on the screen with the file number.

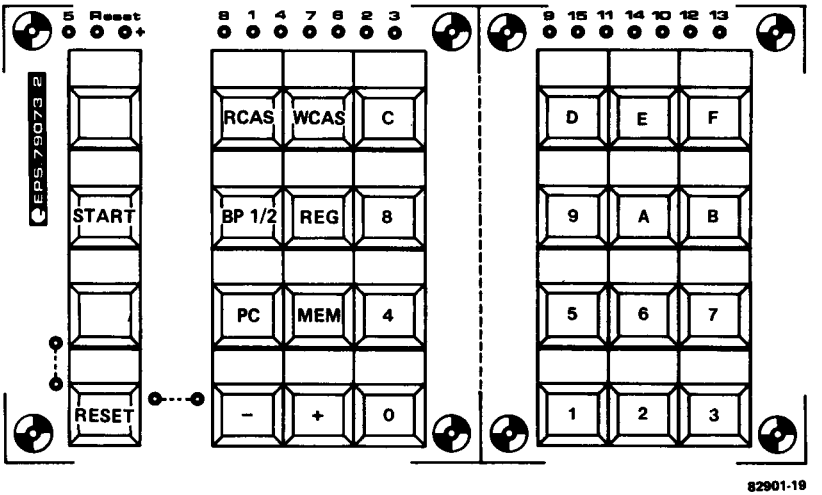


Figure 19. When the processor is in the 'monitor' mode, the keyboard functions are as shown here.

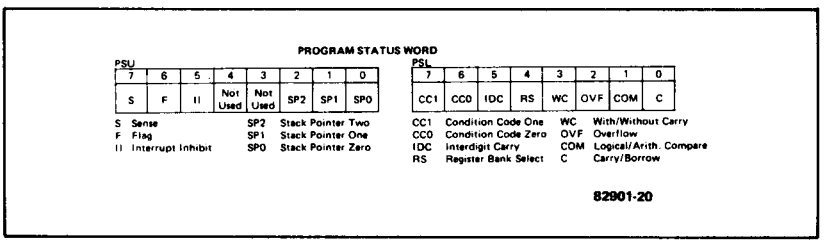


Figure 20. The 16 bits in the status register each have their own significance, as shown here. A full explanation will be given later.

RCAS:

Having stored a program on tape, it can be recalled by means of the Read CASsette routine. This was dealt with extensively in chapter 3: 'Instructions for use'. However, one additional facility was not mentioned: the 'verify' mode. After the 'RCAS' key and file number entry, the '–' key is operated instead of the '+' key. The monitor will now read the program back from the tape, but instead of storing it in the memory it will compare it with the data already present there. Any errors will be indicated; since the original program is still stored in RAM, the WCAS routine can then be repeated.
Since these cassette routines (WCAS and RCAS) are of prime importance in the TV games computer, it is worth trying them out in practice.
This can be done without even loading a program: there is always some kind of data in the memory! The test sequence is as follows:
– operate the 'reset' key;

– operate the 'start' key ('IIII' should appear);
– press the 'WCAS' key ('bEG =');
– enter Ø9ØØ, followed by '+' ('bEG = Ø9ØØ, End =');
– enter ØFFF, followed by '+' ('End = ØFFF, SAd =');
– enter Ø9ØØ, followed by '+' ('SAd = Ø9ØØ, FIL =');
– enter 1, *but not* '+' ('FIL = 1');
– start the tape in the Record mode, and set the level to about half way;
– operate the '+' key.

Hopefully, the recording level meter should indicate approximately nominal full modulation during the first second or so after the '+' key is operated; it will then drop back slightly (to a few dB below full modulation). If this is not the case, the level setting can be corrected, after which the whole sequence described above will have to be repeated. Having found the correct level, it is wise to make a note of it, for future reference.

Having made a complete recording at correct level, the various addresses entered above will reappear on the screen. The test can now be concluded:
– operate the 'RCAS' key ('FIL =');
– enter the file number, '1' ('FIL = 1');
– press the '–' key *(not '+'!)*.

The text 'FIL – 1' will jump to the top of the screen. The tape can now be played back, and the data recorded on it will be compared with the original data in the memory. During this time (approximately 35 seconds) two dots will flash below the '–' sign on the screen. At the end of this time, all the original data will reappear on the screen with the added line 'PC = Ø9ØØ'. If this happens, all is well and the cassette interface is working.

In the unhoped-for event that the check routine breaks off before the end of the recording, with the message 'Ad = Ø9AØ', for instance, then something is wrong... In our experience, moving the recorder further away from the TV set invariably cures the problem.

The only exception is when it consistently breaks off at the start of a main block (Ø9ØØ, ØAØØ and so on). This indicates a hardware fault at those particular memory ICs – a short or open circuit, for instance.

Software for the PVI

The basic principles of the PVI were explained in earlier chapters, and there is no point in repeating them here. However, a general outline of 'what the PVI looks like to the CPU' can prove useful (figure 21).

The address field of the PVI runs from 1FØØ to 1FFF – 256 bytes in all – and can be subdivided into three sections. The first of these (from 1FØØ to 1F6E) contains the data for the basic shape and position of the objects; the second (from 1F8Ø to 1FAE) is for the background and the third is for input/output and further 'control' data.

Objects

How to program the basic shape of an object has already been explained, but the 'position' was only mentioned briefly. The screen is divided into a

grid pattern, consisting of 227 horizontal 'clocks' and 252 vertical 'lines'. The position of the first object is determined by storing the horizontal and vertical coordinates of its top left-hand corner in bytes 1FØA and 1FØC respectively. The position of one or more duplicates is determined by the data in bytes 1FØB and 1FØD. The horizontal position is determined in the same way as that of the main object: the number of 'clocks' to the left-hand edge of the object is entered in 1FØB. The vertical position, however, is given as an 'offset' with respect to the object (or duplicate) immediately above. For instance, if an object is 40 lines high and positioned at the top of the screen (1FØC = ØØ), an offset of 60 lines stored in 1FØD will cause two duplicates to appear below the main object: one at midscreen and one near the bottom.
If an object and/or its duplicates is not to appear on the screen, 'off-screen' coordinates can be entered: for instance, 'FE' in bytes 1FØA through 1FØD. Why 'FE', and not 'FF'? This has to do with the vertical offset for the duplicates, in 1FØD. This value must be one less than the desired number of lines between successive duplicates. In other words, 'FF' indicates 'zero lines' – object and duplicates touch; 'FE' would mean 'minus one' (or '255'), and this disables the duplicates entirely.

Background
Once again, the basic principles have already been discussed – it is now merely a question of dotting the i's and crossing the t's. The addresses of all the relevant bytes and their significance is given in figure 21.
The data stored in the 'vertical bar definition bytes' (1F8Ø... 1FA7) determines which upper edges (2 lines high) and which left-hand sides (18 lines) will appear on the screen. The width of the upper edges and sides is determined by the 'horizontal extension definition bytes' (1FA8... 1FAC): a '1' in bit Ø will widen the corresponding upper edges to 8 clocks; similarly bit 1, for instance, is used to select a width of either 1 or 8 clocks for the bars on the following nine lines (the upper half of the left-hand edges). If the bars are not already widened to 8 clocks by bits Ø... 5, bits 6 and 7 can be used to widen them to 2 or 4 clocks as shown. Note that these last two bits affect all 40 lines corresponding to that byte.

I/O and control
This section was not mentioned as such, although most of its functions were explained. It consists of 16 bytes (only 12 of which are used), running from 1FCØ to 1FCF. Due to incomplete address decoding inside the PVI, the same area corresponds to four sets of addresses, as can be seen in figure 21. However, the first block will normally be used. The functions of the bytes are as follows:

1FCØ:

This determines the sizes of the four objects (and their duplicates, if any). The minimum size is 8 clocks by 10 lines, and this corresponds to ØØ in the two 'size bits' for that object. Twice the size corresponds to Ø1, four times the size to 1Ø and eight times the size is achieved with 11 in the 'size bits'.

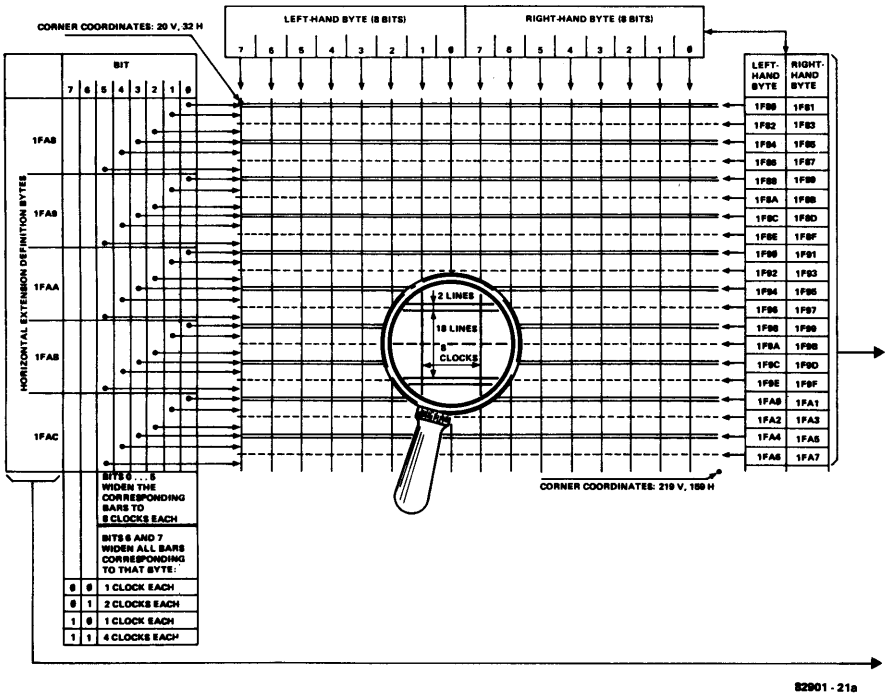


Figure 21. A complete overview of the PVI bytes. The main divisions are given in the central block; more detailed information on the 'object descriptor' areas, the 'background' section and the 'I/O and control' area is contained in the surrounding blocks.

1FC1 and 1FC2:

These bytes determine the colour of the four objects. Each bit determines the *absence* of one of the primary colours (Red, Green or Blue) in one object, as shown. Red, for instance, is ...Ø11...; white is ...ØØØ...

1FC3:

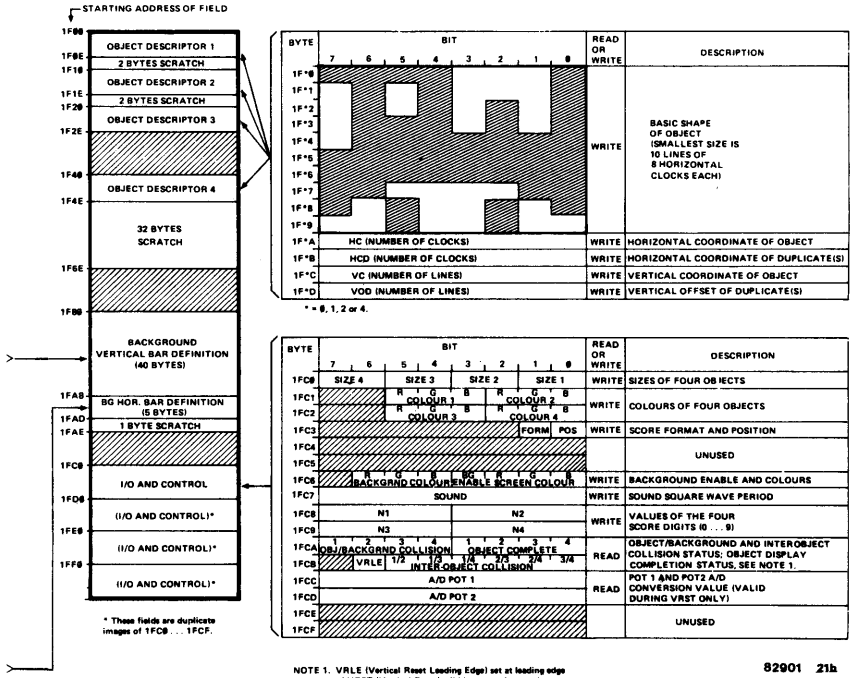
Bit Ø determines the position of the score display: Ø = top of screen, 1 = bottom. Bit 1 determines the format: Ø = two groups of two digits, 1 = one 4-digit number.

1FC4 and 1FC5:

Unused.

1FC6:

Bits Ø, 1 and 2 determine the *presence* of the three primary colours in the 'screen' (i.e. the background 'between the lines'); bits 4, 5 and 6 determine the *absence* of the colours for the background lines. Bit 3 is the 'background enable' bit: it must be '1' for the background to appear.



1FC7:

The 'sound' byte. If ØØ is stored here, no sound will be produced; otherwise, the output frequency is determined as:

$$f_o = \frac{7874}{n + 1},$$

where n is the decimal equivalent of the data stored in this byte.

1FC8 and 1FC9:

The four groups of four bits correspond to the decimal digits in the score display; if the bits correspond to one of the hexadecimal 'digits' A... F, the display is blanked.

The bytes described so far are all 'write bytes' – that is to say that data can be stored here by the processor, but these bytes can not be 'read' by the CPU. For the last four bytes, the reverse is true: data is stored in them by the PVI itself, and the processor can only read this data – not alter it.

1FCA and 1FCB:

At the start of each picture on the TV screen all bits are reset. As the

picture is 'written', collisions between objects and the background are detected and corresponding bits (4... 7 in byte 1FCA) are set to 1. Similarly inter-object collisions set bits Ø... 5 in byte 1FCB.
Furthermore, as soon as each object is completed on the screen this is indicated by one of bits Ø... 3 in 1FCA. This information can prove useful if the same 'object descriptor area' is to be used for more than one shape in the same picture – for instance, to provide complicated score displays at both top and bottom of the screen. As soon as bit 3, say, in 1FCA becomes '1', new data could be stored in bytes 1FØØ to 1FØD.
Finally, bit 6 in 1FCB indicates that the picture on the screen has been completed, so that all possible collisions must by now have been detected. Before starting on the display of the next picture, all bits in these two bytes are reset. They are also reset when read.

1FCC and 1FCD

These bytes contain the 8-bit digital equivalents of the analog input signals from the two potentiometers. Note that the values are only valid during the VRST – 'between pictures', in other words.

As mentioned earlier, data should be stored in the PVI by the processor. For this reason, a program was given in chapter 3 (table 3) to transfer data from the RAM to the PVI. This program can prove a useful aid when exploring the possibilities of the PVI as outlined above: data can be entered from the keyboard, reading the addresses in figure 21 as 'ØA...' instead of '1F...'. For instance, a 'score' display can be produced by storing the necessary data in addresses ØAC3, ØAC8 and ØAC9. The 'Load PVI' program (start address Ø9ØØ) will then cause the CPU to transfer this data to the PVI.
Normally, however, the processor will take a more active part in the proceedings. It is time to take a closer look at this unit.

The processor

If the capabilities of the 2650 are to be exploited to the full, a detailed explanation of the instruction set (figure 22) is a must. We will come to this later.
To illustrate the possibilities, however, the 'development' of a short program can be discussed. Going back to the 'steam engine', let us see how we can set it in motion. From figure 21, we know that the horizontal position is stored in byte 1FØA in the PVI; the same 'horizontal coordinate' is also available in address ØAØA in the RAM, since that is where it was entered from the keyboard.
To move the object horizontally, the contents of byte 1FØA in the PVI must be increased or decreased ('incremented' or 'decremented') at regular intervals. A suitable instruction would therefore appear to be 'Branch on Incrementing Register Relative' (BIRR), since this increments the contents of a register in the 2650. So far, so good; but the register must first contain the position of the object.

The first instruction, therefore, will have to be a 'Load' instruction – to copy the contents of RAM address ØAØA into a register in the processor; register 1, for instance. Using a Load Absolute instruction, we find that the 'op code' (operation code) is ØØØØ 11 followed by the number of the register (Ø1); in hexadecimal notation, this is ØD. Then the 'absolute address' must be specified (ØAØA), so the complete instruction becomes ØDØAØA.
Having copied the 'position data' into R1, the next step could be to increment it. However, since we will actually be using a 'jump' instruction in this case ('branch'), we may as well use it to jump back to the beginning of this program; in other words, this instruction will be included at the end of the program. Each time the value changes, the new horizontal coordinate will have to be loaded into the PVI. Let's take care of this first.
A useful instruction would appear to be 'Store Absolute'. Since the data is in R1, the op code is 11ØØ 11Ø1, or CD; this is followed by the address of the PVI register, so the complete instruction is CD1FØA.
Now for the increment instruction (BIRR)? No, not yet. It *could* be added at this point, and the position would be changed 'at regular intervals' – but the intervals would be so short that the object would appear as little more than a blur. To bring the 'speed' down, some sort of delay will have to be added in the program. No 'delay' instruction exists, as such, so some other means must be found.

Once again, an increment or decrement instruction is the solution. If a register is loaded with 'FF' it will take 256 cycles of a decrement instruction for the contents to become zero. Each decrement cycle takes three machine cycles (3.38 μs in this unit), so it will take 256 x 3 x 3,38 μs (or approximately 2.5 milliseconds) for the complete count-down. This is still rather fast, so the same system can be used to count off 256 count-downs – more than 500 milliseconds in all. Still not enough? Add a third cycle, with a count-down of 2, for example.

The complete 'delay' routine now uses three registers, with corresponding decrement instructions. Since R1 is used for the position data, RØ, R2 and R3 are used here. The 'delay data' (Ø2, FF and FF, for instance) must first be loaded, using a Load Immediate instruction; then, 'Branch on Decrementing Register Relative' (BDRR) instructions are used to decrement each register and jump back as required. The complete delay routine is as follows:

```
Ø7Ø2   LODI,R3   load register R3
Ø6FF   LODI,R2   load register R2
Ø4FF   LODI,RØ   load register RØ
F87E   BDRR,RØ   decrement register
                 RØ
FA7A   BDRR,R2   decrement register
                 R2
FB76   BDRR,R3   decrement register
                 R3
```

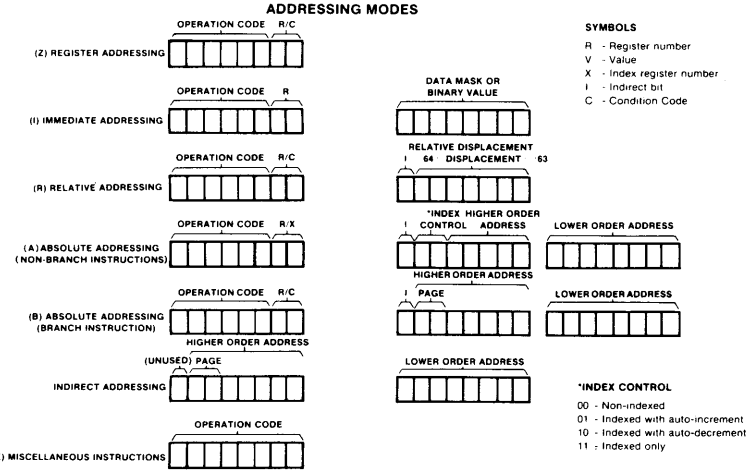
	DESCRIPTION OF OPERATION	FORMAT	MNEMONIC	OP CODE R or CC 0	1	2	3	BYTES	CYCLES	BIT FORMAT	PSW BITS AFFECTED CC	IDC	C	OVF	SP	II	F	NOTE
LOAD/STORE	Load register zero	2	LOD Z	00	01	02	03	1	2	Z	•							1
	Load immediate	5	LOD I	04	05	06	07	2	2	I	•							1
	Load relative	7	LOD R	08	09	0A	0B	2	3	R	•							1,6
	Load absolute	10	LOD A	0C	0D	0E	0F	3	4	A	•							6
	Store register zero	2	STR Z	—	C1	C2	C3	1	2	Z	•							1
	Store relative	7	STR R	C8	C9	CA	CB	2	3	R								6
	Store absolute	10	STR A	CC	CD	CE	CF	3	4	A								6
ARITHMETIC	Add to register zero w/wo carry	2	ADD Z	80	81	82	83	1	2	Z	•	•	•	•				1
	Add immediate w/wo carry	5	ADD I	84	85	86	87	2	2	I	•	•	•	•				1
	Add relative w/wo carry	7	ADD R	88	89	8A	8B	2	3	R	•	•	•	•				1,6
	Add absolute w/wo carry	10	ADD A	8C	8D	8E	8F	3	4	A	•	•	•	•				1,6
	Subtract from register zero w/wo borrow	2	SUB Z	A0	A1	A2	A3	1	2	Z	•	•	•	•				1
	Subtract immediate w/wo borrow	5	SUB I	A4	A5	A6	A7	2	2	I	•	•	•	•				1
	Subtract relative w/wo borrow	7	SUB R	A8	A9	AA	AB	2	3	R	•	•	•	•				1,6
	Subtract absolute w/wo borrow	10	SUB A	AC	AD	AE	AF	3	4	A	•	•	•	•				1,6
	Decimal adjust register	3	DAR	94	95	96	97	1	3	Z	•							1,10
LOGICAL	AND to register zero	2	AND Z	—	41	42	43	1	2	Z	•							1
	AND immediate	5	AND I	44	45	46	47	2	2	I	•							1
	AND relative	7	AND R	48	49	4A	4B	2	3	R	•							1,6
	AND absolute	10	AND A	4C	4D	4E	4F	3	4	A	•							1,6
	Inclusive-OR to register zero	2	IOR Z	60	61	62	63	1	2	Z	•							1
	Inclusive-OR immediate	5	IOR I	64	65	66	67	2	2	I	•							1
	Inclusive-OR relative	7	IOR R	68	69	6A	6B	2	3	R	•							1,6
	Inclusive-OR absolute	10	IOR A	6C	6D	6E	6F	3	4	A	•							1,6
	Exclusive-OR to register zero	2	EOR Z	20	21	22	23	1	2	Z	•							1
	Exclusive-OR immediate	5	EOR I	24	25	26	27	2	2	I	•							1
	Exclusive-OR relative	7	EOR R	28	29	2A	2B	2	3	R	•							1,6
	Exclusive-OR absolute	10	EOR A	2C	2D	2E	2F	3	4	A	•							1,6
ROTATE/COMPARE	Compare to register zero arithmetic/logical	2	COM Z	E0	E1	E2	E3	1	2	Z	•							2
	Compare immediate arithmetic/logical	5	COM I	E4	E5	E6	E7	2	2	I	•							3
	Compare relative arithmetic/logical	7	COM R	E8	E9	EA	EB	2	3	R	•							3,6
	Compare absolute aritmetic/logical	10	COM A	EC	ED	EE	EF	3	4	A	•							3,6
	Rotate register w/wo carry	3	RRR	50	51	52	53	1	2	Z	•	•	•	•				1
	Rotate register left w/wo carry	3	RRL	D0	D1	D2	D3	1	2	Z	•	•	•	•				1
BRANCH	Branch on condition true relative	7	BCT R	18	19	1A	1B	2	3	R								7,8
	Branch on condition true absolute	8	BCT A	1C	1D	1E	1F	3	3	B								7,8
	Branch on condition false relative	7	BCF R	98	99	9A	—	2	3	R								7
	Branch on condition false absolute	8	BCF A	9C	9D	9E	—	3	3	B								7
	Branch on register non-zero relative	7	BRN R	58	59	5A	5B	2	3	R								7,8
	Branch on register non-zero absolute	8	BRN A	5C	5D	5E	5F	3	3	B								7,8
BRANCH	Branch on incrementing register relative	7	BIR R	D8	D9	DA	DB	2	3	R								7,8
	Branch on incrementing register absolute	8	BIR A	DC	DD	DE	DF	3	3	B								7,8
	Branch on decrementing register relative	7	BDR R	F8	F9	FA	FB	2	3	R								7,8
	Branch on decrementing register absolute	8	BDR A	FC	FD	FE	FF	3	3	B								7,8
	Zero branch relative, unconditional	6	ZBRR	—	—	—	9B	2	3	ER								6
	Branch indexed absolute, unconditional	9	BXA	—	—	—	9F	3	3	EB								5,6
SUBROUTINE BRANCH/RETURN	Branch to subroutine on condition true, relative	7	BST R	38	39	3A	3B	2	3	R					•			7,8
	Branch to subroutine on condition true, absolute	8	BST A	3C	3D	3E	3F	3	3	B					•			7,8
	Branch to subroutine on condition false, relative	7	BSF R	B8	B9	BA	—	2	3	R					•			7
	Branch to subroutine on condition false, absolute	8	BSF A	BC	BD	BE	—	3	3	B					•			7
	Branch to subroutine on non-zero register, relative	7	BSN R	78	79	7A	7B	2	3	R					•			7,8
	Branch to subroutine on non-zero register, absolute	8	BSN A	7C	7D	7E	7F	3	3	B					•			7,8
	Zero branch to subroutine relative, unconditional	6	ZBSR	—	—	—	BB	2	3	ER								6
	Branch to subroutine, indexed, absolute unconditional	9	BSXA	—	—	—	BF	3	3	EB								5,6
	Return from subroutine, conditional	3	RET C	14	15	16	17	1	3	Z					•			8
	Return from subroutine and enable interrupt, conditional	3	RET E	34	35	36	37	1	3	Z					•	•		8

	DESCRIPTION OF OPERATION	FORMAT	MNEMONIC	OP CODE R or CC 0	1	2	3	BYTES	CYCLES	BIT FORMAT	PSW BITS AFFECTED CC	IDC	C	OVF	SP	II	F	NOTE
INPUT/OUTPUT	Write data	3	WRTD	F0	F1	F2	F3	1	2	Z								
	Read data	3	REDD	70	71	72	73	1	2	Z	•							1
	Write control	3	WRTC	B0	B1	B2	B3	1	2	Z								
	Read control	3	REDC	30	31	32	33	1	2	Z	•							1
	Write extended	5	WRTE	D4	D5	D6	D7	2	3	I								
	Read extended	5	REDE	54	55	56	57	2	3	I	•							1
MISC.	Halt, enter wait state	1	HALT	40	—	—	—	1	1	E								
	No operation	1	NOP	C0	—	—	—	1	1	E								
	Test under mask immediate	5	TMI	F4	F5	F6	F7	2	3	I	•							4
PROGRAM STATUS	Load program status, upper	1	LPS U	—	—	92	—	1	2	E					•	•	•	
	Load program status, lower	1	LPS L	—	—	93	—	1	2	E	•	•	•	•				9
	Store program status, upper	1	SPS U	—	—	12	—	1	2	E	•							1
	Store program status, lower	1	SPS L	—	—	13	—	1	2	E	•							1
	Clear program status, upper, masked	4	CPS U	—	—	74	—	2	3	EI					•	•	•	
	Clear program status, lower, masked	4	CPS L	—	—	75	—	2	3	EI	•	•	•	•				9
	Preset program status, upper, masked	4	PPS U	—	—	76	—	2	3	EI					•	•	•	
	Preset program status, lower, masked	4	PPS L	—	—	77	—	2	3	EI	•	•	•	•				9
	Test program status, upper, masked	4	TPS U	—	—	B4	—	2	3	EI	•							4
	Test program status, lower, masked	4	TPS L	—	—	B5	—	2	3	EI	•							4

NOTES

1. Condition code CC1, CC0: 01 if positive, 00 if zero, 10 if negative
2. Condition code CC1, CC0: 01 if R0 > r, 00 if R0 = r, 10 if R0 < r
3. Condition code CC1, CC0: 01 if r > V, 00 if r = V, 10 if r < V
4. Condition code CC1, CC0: 00 if all selected bits are 1s, 10 if not all the selected bits are 1s
5. Index register must be register 3 or 3'
6. Requires two additional cycles if indirection is specified
7. Requires two additional cycles if indirection is specified and branch is taken
8. Specify CC = 11 for unconditional branch
9. RS, WC and COM bits in PSW are also affected
10. CC assumes number in register is a binary number

ADDRESSING MODES



PROGRAM STATUS WORD

82901-22

PSU

7	6	5	4	3	2	1	0
S	F	II	Not Used	Not Used	SP2	SP1	SP0

S Sense
F Flag
II Interrupt Inhibit
SP2 Stack Pointer Two
SP1 Stack Pointer One
SP0 Stack Pointer Zero

PSL

7	6	5	4	3	2	1	0
CC1	CC0	IDC	RS	WC	OVF	COM	C

CC1 Condition Code One
CC0 Condition Code Zero
IDC Interdigit Carry
RS Register Bank Select
WC With/Without Carry
OVF Overflow
COM Logical/Arith. Compare
C Carry/Borrow

Figure 22. A brief summary of the instruction set for the 2650, as provided by the manufacturer. A full explanation is given later on.

Table 4.

address	data	mnemonic	explanation
ØBØØ	ØDØAØA	LODA,R1	Load position in R1.
ØBØ3	CD1FØA	STRA,R1	Load data from R1 into PVI.
ØBØ6	Ø7Ø1	LODI,R3	
ØBØ8	Ø64Ø	LODI,R2	
ØBØA	Ø48Ø	LODI,RØ	Delay routine.
ØBØC	F87E	BDRR,RØ	
ØBØE	FA7A	BDRR,R2	
ØB1Ø	FB76	BDRR,R3	
ØB12	ØC1E88	LODA,RØ	Check the 'PC' key:
ØB15	442Ø	ANDI,RØ	if it is not depressed,
ØB17	9903	BCFR	jump to ØB1C;
ØB19	1FØØØØ	BCTA,UN	if it is, jump to monitor.
ØB1C	D965	BIRR,R1	Increment position and jump back.
ØB1E	1FØBØ3	BCTA,UN	Jump back to ØBØ3.

Table 4. This program can be used as an extension of the demonstration program contained in chapter 3 (table 3): it causes the 'object' to move across the screen.

Now, the increment instruction for R1 could be given – the program, as such, is then complete. However, it still contains some imperfections. In the first place, there is no 'easy way out' from this program, once it is running! For this reason, a 'keyscan' is added after the delay routine: the 'PC' key is scanned and a jump back to the monitor program occurs if this key is depressed. A further small problem is associated with the increment instruction (BIRR) itself. This is a so-called conditional jump – in this case, the jump back to the beginning of the program is only carried out if the contents of the register (R1) don't become zero. After a certain number of 'increment' steps, however, the contents *will* become zero; no jump occurs, and the next instruction is carried out. Since we want the jump back to occur after each cycle, the following instruction must be a further (absolute) jump: BCTA.

The complete program is now as shown in table 4: the Load instruction into R1; the Store instruction into the PVI; the delay routine (ØB06... ØB1Ø); the 'keyscan' (ØB12...ØB19); and, finally, the increment and jump instructions. Finally, we need some way to start the program. The original 'load PVI' program (table 3 in chapter 3) can be modified slightly: at address Ø915, the data becomes 1FØBØØ instead of 1FØØØØ. Now, if the '–' key is operated while the 'load PVI' program is running, a jump occurs to address ØBØØ in the RAM (instead of back to the monitor). If the program described here (table 4) is loaded in RAM from address ØBØØ on, it will then set the object (steam engine) in motion. To get back to the monitor program, the 'PC' key must be depressed. Note that it will have to be held down until the delay routine has been completed – up to 1 second, in other words.

7

Bits and bytes, clocks and lines

The information given so far is sufficient for 'passive' use of the TV games computer: running programs off ESS tapes, with at least some idea of 'what is going on'. When it comes to taking a more active part in the proceedings – writing your own programs – something more is needed. The 'instruction set', 'programming procedures' and 'special routines' will all be dealt with in the following chapters.
First, however, some groundwork is required: a good understanding of a few basic principles. Not many, fortunately – the TV games computer is a fairly simple machine – but none the less: essential!

The first major hurdle is to learn to think in hexadecimal ('hex') numbers. Just understanding the system isn't enough: it must become second nature. When you find yourself counting resistors, or sheep: '...8, 9, A, B – oh no, I mean 10, 11,...' – that means you're on the right road!

The basic idea was explained earlier. Digital systems use 'bits', that can be either Ø or 1. Working from right to left in a group of bits, the first is given the value 1, the second represents 2, the third 4, the fourth is 8, and so on. Working from right to left, in other words, each bit has twice the value of its predecessor. Using four bits in this way, you can specify 16 numbers. The first ten of these (Ø...9) are written in the normal way; the remaining six (10... 15) are written as the first six letters of the alphabet. The result is as shown below:

bits	value	hex digit
0000	0	Ø
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

In each case, the value is found by simply adding the values of all bits that are given as '1'. Take 9, for example: the extreme right-hand bit is 1, so that counts for 1; the second and third bits (corresponding to values 2 and 4) are Ø, so they don't count; the last (left-hand) bit is 1, so it adds 8 to the total value. All in all, 1 + 8 = 9.

All this is quite straightforward. The only problem is to get used to associating any given group of four bits with the corresponding hex digit, and vice versa, without having to look it up in the table each time. This requires quite a bit of practice. As a first step, several examples are given below, each written as follows:

1100 0011 = C3

Eight bits (one 'byte') are shown with the corresponding two-digit 'hex' number. Each example of this type can be used as an exercise in either direction: you can look at the bits, work out the hex number, and check to see whether you got it right; or else look at the hex number and work out the bits. Ready? Here we go:

0001 1000 = 18
0000 1001 = Ø9
0101 0000 = 5Ø
0011 0100 = 34
0110 0010 = 62
1000 0111 = 87
0111 1000 = 78
1100 0000 = CØ
1100 1100 = CC
0000 1111 = ØF
1110 1010 = EA
0011 1011 = 3B
0001 0111 = 17
0010 1101 = 2D
1111 1110 = FE
0100 0000 = 4Ø

Enough for now. There will be many more examples in the following chapters. However, the importance of all this should not be underestimated. The TV Games computer has a hex keyboard, to avoid having to key in thousands of ones and zeroes, but very often (as when programming an object shape) it is the corresponding ones and zeroes that matter!

The TV picture

When writing programs, it is a great help to be able to visualise how the PVI puts a picture on the screen. The basic principle is illustrated in figure 23. Starting at the top, the black bar represents the Vertical Reset pulse,

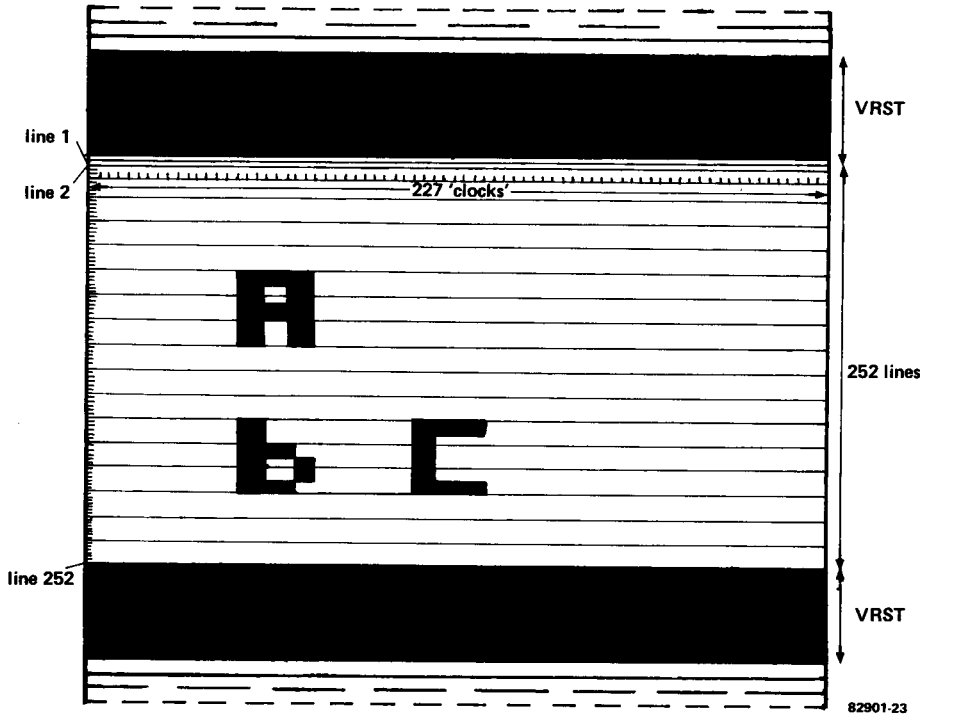


Figure 23. The basic principle of the TV picture. This is written from left to right and from top to bottom.

VRST. After this, the first actual picture line is written from left to right; its total length is approximately 230 'clocks' (more on this later). The second line is then written below the first, again from left to right, and so on down the screen – for a total of 252 lines. Finally, another VRST pulse, followed by the next picture 'frame'.

The important point to note is that all this takes time: 20 milliseconds for one complete frame. Object 'B' in figure 23 will therefore actually appear on the screen several milliseconds after 'A'. 'B' and 'C' are written almost simultaneously: each 'line' in C follows the corresponding part of B with a lag of only a few microseconds.

Milliseconds and microseconds are quite fast enough to make the human eye think that the complete, moving picture is present all the time. However, the microprocessor can get confused. On some occasions, it can be too fast – changing the picture data several times in one frame, which doesn't improve the final result... At other times, surprisingly enough, it can be too slow. Microprocessors *do* need time to think, and if all sorts of calculations must be carried out after one object is completed on the screen and before the next starts, a sufficient gap must be left between those objects.

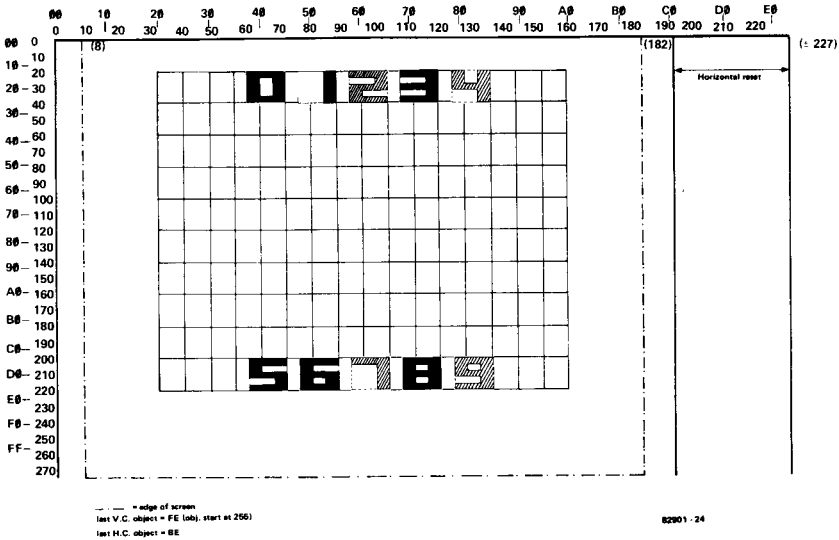


Figure 24. The position of background, score and total display areas with respect to the horizontal 'clocks' and vertical 'lines'.

To put this into perspective: each line in the picture (including the horizontal reset pulse) corresponds to slightly less than 230 'clocks' or clock pulses, as mentioned above. Twelve clock pulses correspond to one processor 'cycle', which means that just over nineteen 'cycles' fit on each line. What's a 'cycle'? In practical terms, it is best to refer back to figure 22. For each instruction, this figure gives the basic number of cycles required; as can be seen, this varies from 2 to 4 (or even more, in a few cases like 'indirect addressing' – again: more on this later). With about twenty cycles to a line, this means that only five to ten instructions can be executed during each picture line. In other words: you can't run lengthy program sections 'between the lines'!

A further point worth noting is that the PVI 'sees' a slightly larger picture than that actually appearing on the TV screen. This is illustrated in figure 24. The 'clocks' are given along the top and the 'lines' run down the left-hand side. Depending on the adjustment of the TV set, the section shown within the dot-dash line will normally appear on the screen. Two small areas, one at the left and one at the right, are off-screen – but 'visible' to the PVI! If an object is moved from left to right, it will disappear off the screen at the right and only reappear at the left after a noticeable delay. While off-screen, it can still collide with another object: a point worth noting when designing games programs!

Also shown in the same figure are the positions of the basic background and the score display. This can be quite useful as an initial reference for object positions. Like many other useful diagrams, it is repeated in the appendix.

8

The more important instructions

Telling the microprocessor what to do.

The essence of a TV game program is that the processor puts an interesting display on the screen, which poses some kind of interesting challenge: shoot the aliens, find your way through a maze, fly a helicopter, or whatever. While you play the game, the processor is checking to see how you're doing and modifying the display accordingly.
This means that the processor is playing an active role behind the scenes. However, it's a rather stupid thing, so you have to tell it exactly what to do before you start. 'Telling it what to do' means that you must provide it with a list of instructions; together, these instructions are the 'program'. So far, the programs have been fed into the unit from a tape. Now, we're coming to the point where you can start to write your own programs! In this chapter, we will deal with the more important instructions – sufficient to get a moving display on the screen.

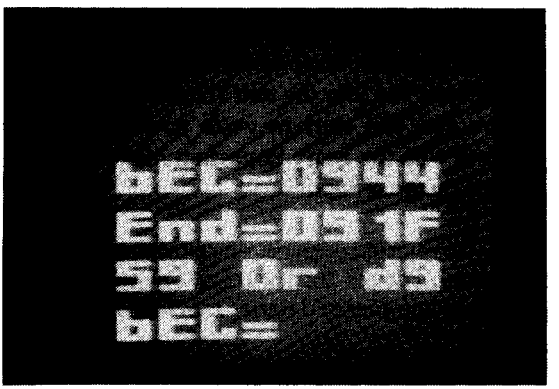
To make full use of a microprocessor, one should normally have access to the instruction manual. For the 2650, this is a 174-page book... Fortunately, the main points can be summarised rather more briefly.

Addressing modes

When fetching or storing data, or jumping to and fro in a program, it is essential to specify the 'address' concerned. Obviously. In the TV Games computer, there are several different ways of doing this.

Absolute or relative
An 'absolute' address is simply the address itself. For instance, in machine language the instruction for 'Load Absolute into register zero' starts with ØC (more on this later!); if the data is to be fetched from address ØFØØ, the full instruction will therefore be ØCØFØØ.
A 'relative' address, on the other hand, specifies a small jump in the program. Basically, the processor will calculate an 'absolute' address by adding the specified number (between −64 and +63) to the address that follows that particular instruction. As an example, if the instruction 'Load Relative into register zero, 2F' (in machine language: Ø82F) is located at the two address bytes Ø93E and Ø93F, the following address is Ø94Ø. The 'absolute' address corresponding to this instruction is therefore Ø94Ø + 2F = Ø96F, and the data will be fetched from there.
The negative number required for a 'backwards' jump is entered as a '7-bit

two's complement number'. In simple language, this means that you count down from 80_{HEX}. For instance, if in the previous example the data was to be loaded from address Ø93D, the relative address would be 7D: the 'following address', Ø94Ø corresponds to 8Ø, so Ø93F corresponds to 7F, Ø93E to 7E and Ø93D to 7D. The full instruction is therefore: Ø87D. Note that this way of specifying negative numbers means that ØØ... 3F are positive; 4Ø... 7F are negative; greater than 7F don't exist.
All this may or may not seem simple in theory; in practice it has proved a source of endless programming errors... It is easier to miscalculate a relative address than to get it right! For simple programs, one may as well use 'absolute addressing' – the additional memory space required (the corresponding instructions are longer) is rarely a problem.
However, practice makes perfect, and as programs get more complex it becomes worthwhile to start using relative addresses wherever possible. As an aid to the beginner, file C on the ESS 007 cassette contains a calculation routine for relative addresses – a useful check!



Direct or indirect

The two types of addressing explained above are both referred to as 'direct' address modes: data is transferred from or to the specified address. An alternative possibility is a two-step operation: specify an address where the desired address can be found. This is referred to as 'indirect' addressing.
Although both absolute and relative indirect addresses are possible, only the latter are useful in the basic TV games computer. A relative address is converted to an indirect relative address by adding 8Ø. In the example given above, the 'load relative' instruction Ø82F was located at addresses Ø93E and Ø93F; the data was then fetched from address Ø96F. However, if the instruction is modified to Ø8AF (2F + 8Ø = AF) the contents of addresses Ø96F and Ø97Ø will be used as the absolute address for this instruction: if the data stored at these addresses is ØA and ØØ, say, the 'load indirect relative, 2F' instruction will be carried out as if it read 'load absolute from ØAØØ'.
Once again, for simple programs it is easier, quicker and more reliable to use the corresponding 'absolute' instruction, and forget about the 'relative indirect' mode. As an aid to courageous novices, the calculation routine

Table A.

```
08F0    C0 60 50 CE
08F4    3D CE 50 60                          DATA
08F8    C0 00 28 FF
08FC    63 FE 00 00

0900    7620        PPSU, I1
0902    05C3        LODI, R1
0904    0400        LODI, R0                 clear PVI
0906 ┌► CD5F00      STRA, I-R1
0909 └─ 597B        BRNR, R1

090B    050E        LODI, R1
090D ┌► 0D48F0      LODA, I-R1               store
0910 │  CD7F00      STRA, I/R1               object
0913 └─ 5978        BRNR, R1                 shape

0915    0401        LODI, R0                 size
0917    CC1FC0      STRA, R0

091A    0400        LODI, R0                 colour*
091C    CC1FC1      STRA, R0

091F ┌► 0C1E88      LODA, R0                 wait for
0922 │  F420        TMI, R0                  'PC' key
0924 └─ 9879        BCFR

0926    1F0000      BCTA, UN                 return to monitor
```

* As things stand, these two instructions are unnecessary: the data in address 1FC1 is already 00 after the 'clear PVI' routine. However, other colours can now be selected by modifying the data in the LODI instruction.

Table A. An illustration of what can be achieved with the instructions described in this chapter! The program is started at address 0900. If it works, proceed to Table B!

mentioned above actually gives two results: if the relative jump in the previous examples is calculated, the answer will appear as '2F Or AF' – for direct and indirect, respectively!

Indexed

In contrast to the 'relative' and 'indirect' addressing modes, 'indexed' addressing can prove extremely useful in even the simplest of programs. The basic idea is that the data stored in one of the registers is added to a specified 'absolute' address; the result of this addition is used as the absolute address for the instruction. The register containing the additional data for the address is referred to as the 'index register', and this register must be specified in the instruction. The data are always transferred to or from register zero when indexed addressing is used.

To specify the basic indexing mode, *6000* is added to the absolute address. Thus 0D6900 is *not* interpreted as 'load register one from absolute address 6900'; if we assume that the data already in register one is 0A, the instruction will be read as 'load register *zero* from absolute address 090A' – i.e. from 0900 plus the data in register one.

Two further extensions of this instruction make it invaluable: 'indexed with

auto-increment' and 'indexed with auto-decrement', specified by adding 2000 or 4000 to the absolute address. In both cases, the final address is calculated in the same way – by adding the data in the 'index register' to the specified absolute address. However, *before* calculating the final address, the data in the index register are increased by one ('auto-increment') or one is subtracted from the data ('auto-decrement').
The value of this instruction is best illustrated in an example. Let us assume that we want to clear all 'background data' in the PVI. This means storing 00 in all addresses from 1F80 ... 1FAC: 45 in all! Instead of using 45 individual 'store absolute' instructions, a single 'store absolute, indexed with auto-decrement' instruction can be used, with a bit of padding:

```
     052D      LODI, R1
     0400      LODI, R0
  ┌► CD5F80    STRA, I-R1
  └─ 597B      BRNR, R1
```

The 'shorthand abbreviations' given after the actual machine-code instructions are referred to as 'mnemonics'. They are simply a quick way to jot down what the instruction does.
This brief section of program is executed as follows. First, the 'index register', R1, is loaded ('LODI, R1' = Load Immediate, Register 1 – more on this later) and '00' is loaded into Register 0. This is followed by the 'Store Absolute, Indexed to Register 1 with auto-decrement' instruction – incidentally, the value of mnemonics is clearly illustrated here: it is a lot quicker to write 'STRA, I-R1' than the mouthful given above. At this point, the value in R1 (2D) is reduced by one and the result (2C) is added to the basic absolute address 1F80 (5F80 = 1F80 + 4000 for 'auto-decrement'). The value in R0 (00) is then stored in the resultant absolute address: 1F80 + 2C = 1FAC. One down, 44 to go! The next instruction, which will be explained in greater detail later, is 'Branch if Register 1 is Non-zero, Relative'. Since R1 is most definitely non-zero (it is still 2C at this point), the 'relative branch' is executed: the program 'jumps back' to the beginning of the previous instruction, as indicated by the arrow. This whole performance is repeated, storing 00 in progressively lower PVI addresses, until the data in R1 becomes zero. At this point, the BRNR, R1, instruction does *not* result in a jump back, since R1 *is* zero, and the rest of the program is carried out.
For those who feel like trying out this program, it is more interesting to turn the background *on* instead of off. In that case, the background and screen colour must also be specified: '69' in address 1FC6 gives yellow on blue. Furthermore, the objects will have to be cleared, since they are also used by the monitor program. A complete program is given in Table 5; the reason for starting at address 0903 (instead of 0900) will be given later.
While on the subject of indexed addressing, one final point should be noted. In general, this mode is available as a variation of all absolute addresses, *with the exception of branch instructions*. The only two indexed branch instructions, BXA and BSXA, will be discussed further on.

Table 5.

0903	054E	LODI, R1	clear objects
0905	0400	LODI, R0	
0907	CD5F00	STRA, I-R1	
090A	597B	BRNR, R1	
090C	0469	LODI, R0	colour
090E	CC1FC6	STRA, R0	
0911	052D	LODI, R1	background
0913	04FF	LODI, R0	
0915	CD5F80	STRA, I-R1	
0918	597B	BRNR, R1	
091A	40	HALT*	

* Not the best way to end a program, as we shall see, but good enough for now!

To or from register (zero)

Nearly all instructions involving transfer or manipulation of data require the use of a register. Obviously, the register to be used must be specified in the instruction.

In the examples already given, and Table 5 in particular, this principle is clear. The first byte of each instruction specifies the basic instruction *and* the register involved. For instance, the basic instruction for 'Load Immediate' is 04xx (where 'xx' is the data to be loaded); adding the number of the register to this gives the complete instruction: 04xx for Register 0, 05xx for R1, 06xx for R2 and 07xx for R3. In practice, this means that four variations exist for most instructions: one for each register. (It also means that the second digit in an instruction specifies the register involved: 0, 4, 8 and C for register 0 (0803, for instance); 1, 5, 9 and D for register 1; and so on).

Finally, some instructions refer to data transfer or manipulation involving two registers, *one of which is always register zero.* The instruction 'Load Register 0 from Register 1', for instance, is 01. Similarly, 'LODZ, R2' (to use the mnemonic) is 02. It should be noted that in some cases, but not all(!), *both* registers can be specified as Register 0. This can sometimes be useful, as will be explained under 'tricks and gimmicks', in chapter 10.

Registers

We have already mentioned 'registers' several times. It is now time to take a closer look at them. To put it in a nutshell, a register can be visualised as a memory location *inside the microprocessor itself.* In the 2650, 8-bit registers are used; this means that they can store any data value from 00 to FF. In all, seven 'general-purpose' registers are available: register 0 and two 'banks' of three registers (R1, R2, R3 and R1′, R2′ and R3′). Of these

Table B.

The program given in table A should produce a white object on a blue screen. To include a 'background', the program can be modified from address 091F on, as follows:

(091C	CC1FC1	STRA, R0)	
091F	0480	LODI, R0	
0921	CC1F91	STRA, R0	
0924	CC1F93	STRA, R0	
0927	CC1F9F	STRA, R0	
092A	CC1FA1	STRA, R0	
092D	040C	LODI, R0	load background
092F	CC1F98	STRA, R0	
0932	0430	LODI, R0	
0934	CC1F99	STRA, R0	
0937	0401	LODI, R0	
0939	CC1FAB	STRA, R0	
093C	0449	LODI, R0	colour
093E	CC1FC6	STRA, R0	
0941	0C1E88	LODA, R0	wait for 'PC' key
0944	F420	TMI, R0	
0946	9879	BCFR	
0948	1F0000	BCTA, UN	return to monitor

The complete program is again started from 0900. For the next step, see Table C.

seven, register 0 is always immediately available; at any given moment, however, only one of the register banks (R1 ... R3 or R1′ ... R3′) is accessible. The other bank, and the data contained in those three registers, is 'in cold storage'. (The way in which one or other of these banks can be selected will be discussed below: see 'Program Status Word'). Any instruction referring to R1, R2 or R3 is performed only on that register in the selected bank – it has no effect on the corresponding register in the other bank.

Program Status Word

The 'Program Status Word' refers to two special-purpose 8-bit registers: the 'Program Status Upper' (PSU) and 'Program Status Lower' (PSL). Each bit in these registers has a special meaning, as illustrated in figure 25. Briefly, the most important points as they relate to the complete TV games computer are as follows:

– *Sense*: this bit is '1' for the duration of the vertical reset pulse, at the end of each 'frame'. It can be used, for example, to synchronise the program to the actual display on the screen.

– *Flag*: can be set, reset and tested at will, as an indication of some condition relating to the program – for instance, to distinguish between the first and following runs through a particular section in the program.

– *Interrupt Inhibit*. The PVI generates 'interrupts' at the end of each

Figure 25.

Program Status Word

PSU

bit:	7	6	5	4	3	2	1	ø
function:	S	F	II	Not Used	Not Used	SP2	SP1	SPØ
hex code:	8Ø	4Ø	2Ø	1Ø	ØB	Ø4	Ø2	Ø1

S	Sense	SP2	Stack Pointer Two
F	Flag	SP1	Stack Pointer One
II	Interrupt Inhibit	SPØ	Stack Pointer Zero

PSL

7	6	5	4	3	2	1	ø
CC1	CCØ	IDC	RS	WC	OVF	COM	C
8Ø	4Ø	2Ø	1Ø	ØB	Ø4	Ø2	Ø1

CC1	Condition Code One	WC	With/Without Carry
CCØ	Condition Code Zero	OVF	Overflow
IDC	Interdigit Carry	COM	Logical/Arith. Compare
RS	Register Bank Select	C	Carry/Borrow

82901-25

frame and each time an object is completed. If this bit is set, these interrupt requests are ignored; otherwise, program execution 'jumps' from wherever it happens to be to address ØØØ3 and runs the program section that it finds there as a subroutine. Note that this can cause chaos, if one isn't aware of the mechanism; for this reason, it is advisable to start every program with the instruction '762Ø' (i.e. set Interrupt Inhibit). The Interrupt Inhibit bit is set automatically by the processor when the interrupt routine is executed; it is only reset by an explicit command in the program.

– *Stack pointers.* These three bits are set and reset by the processor, to keep track of the 'subroutine levels'. The stack is eight levels deep, which means that the main program may branch to a subroutine, that may branch to a further subroutine, and so on up to eight times before starting to 'climb back up' by means of Return instructions. It is possible to modify the stack pointers deliberately, as part of a program, but this is unwise for beginners...

– *Condition Code.* These two bits are set by (the result of) several different instructions, as shown in the Instruction Set given in figure 22. For instance, if the data loaded into a register is ØØ, the condition code will also be set to ØØ. Most of the branch and return instructions can be made 'conditional', by specifying a particular condition code setting: in that case, a 'Branch on Condition True' instruction, for instance, will only be executed if the actual condition code at that point corresponds to the one specified. If the two don't correspond, the instruction is ignored.

Table 6. **Load and Store**

description		example	comments
Load register zero	(LODZ)	Ø2	**from** R2 **to** RØ
Load immediate	(LODI)	Ø4xx	'xx' = data
Load relative	(LODR)	Ø8yy	'yy' = displacement
Load absolute	(LODA)	ØCzzzz	'zzzz' = address
Store register zero	(STRZ)	C1	**to** R1 **from** RØ
Store relative	(STRR)	C8yy	'yy' = displacement
Store absolute	(STRA)	CCzzzz	'zzzz' = address

– *IDC, WC, OVF, COM, C.* These five bits will be dealt with later; see: Arithmetic and Compare, chapter 10.
– *Register bank Select.* This bit is used to select one or other of the two 'register banks' described above.

Various manipulations are possible on the two Program Status registers, as will be described. The Clear, Preset and Test instructions will prove the most useful; these can be used to set any bit (or combination of bits) to Ø or 1, as required, and to test the setting of any bit(s). An example was given above: '762Ø' is the code for 'Preset Program Status, Upper, Masked 2Ø'; as can be seen in figure 25, this sets the Interrupt Inhibit bit.

Instruction Set

Several instructions have already been mentioned briefly; having laid the groundwork, it is now possible to examine the instruction set in greater detail.

Load and store
The principle of these instructions is obvious: data is transferred into (Load) or from (Store) a specified register.
Load Register zero and Store Register zero transfer data between RØ and one of the other three registers. 'C1', for example, transfers data *from* RØ *to* R1. Note that the instructions 'ØØ' and 'CØ', for 'LODZ, RØ' and 'STRZ, RØ', don't exist.
Load immediate transfers the data given in the instruction to the specified register. 'Ø7CA' (= LODI, R3) loads the data 'CA' into register 3.
Load relative and Store relative refer to the relative addressing mode described earlier. Relative Indirect addressing can also be used, as described earlier.
Load absolute and Store absolute are used when absolute or absolute indexed addressing is required.

In all cases, except store relative and store absolute, the two Condition Code bits are set according to the sign of the data transferred: they become

Table C.

The program given so far in tables A and B will produce a stationary object and background.
The next step is to set the object in motion, by modifying the program from address 0941 on:

```
(093E    CC1FC6   STRA, R0)
0941     0728     LODI, R3      horizontal preset
0943     0663     LODI, R2      vertical preset
0945     0C1E8C   LODA, R0  }
0948     F420     TMI, R0   }   '5' key?
094A     9802     BCFR      }
094C     DB18     BIRR, R3      increment R3!
094E     0C1E8D   LODA, R0  }
0951     F410     TMI, R0   }   '2' key?
0953     9802     BCFR      }
0955     FA0F     BDRR, R2      decrement R2!
0957     F440     TMI, R0   }   'A' key?
0959     9802     BCFR      }
095B     DA09     BIRR, R2      increment R2!
095D     0C1E8E   LODA, R0  }
0960     F420     TMI, R0   }   '7' key?
0962     9C0991   BCFA      }
0965     FB00     BDRR, R3      decrement R3!
0967     CF1F0A   STRA, R3  }   update position
096A     CE1F0C   STRA, R2  }
096D     03       LODZ, R3
096E     3B83     BSTR, UN, Ind. (0984)  }
0970     C3       STRZ, R3               }
0971     02       LODZ, R2               } check for 'end of range'
0972     3F0984   BSTA, UN               }
0975     C2       STRZ, R2               }
0976     0502     LODI, R1  }
0978     0C1FCB   LODA, R0  }
097B     F440     TMI, R0   }   delay
097D     9879     BCFR      }
097F     F977     BDRR, R1  }
0981     1F0945   BCTA, UN      repeat key checks
0984     E404     COMI, R0  }
0986     9802     BCFR      }   SUBROUTINE:
0988     D806     BIRR, R0  }   check for end of (horizontal
098A     E4D0     COMI, R0  }   or vertical) range, and
098C     9802     BCFR      }   correct if necessary
098E     F800     BDRR, R0  }
0990     17       RETC, UN  }
0991     0C1E88   LODA, R0  }
0994     F420     TMI, R0   }   'PC' key?
0996     9C0945   BCFA      }   repeat if not
0999     1F0000   BCTA, UN      return to monitor
```

After loading this program, it should be possible to move the object to and fro horizontally by operating the '5' and '7' keys; vertical control is provided by the '2' and 'A' keys.

Ø1 if the data is a positive number, ØØ if it is zero and 1Ø if it is negative (i.e. 8Ø ... FF, corresponding to −128 ... −1).
The Load and Store instructions can be summarised as shown in Table 6.

(Subroutine)Branch
Normally speaking, a program is executed step by step: in other words, the instructions are carried out in the order in which they are stored in the memory. If a jump to a different section of the program is required, a so-called Branch instruction must be used.
There are two basic types of Branch instruction: those for a (main program) *Branch* and those for a *Branch to Subroutine*. In the former case, the main program itself jumps to a different point in the memory; a Branch to Subroutine, on the other hand, can be considered as an interruption in the main program: the main program is stopped at the branch-to-subroutine instruction, the subroutine (elsewhere in memory) is carried out, after which the main program continues at the point where it was interrupted. Several variations of both types of Branch instruction are available:
Branch (to Subroutine) on Condition True, Relative or Absolute. For each of these four basic instructions, a particular setting of the Condition Code bits can be specified; the branch will only be executed if the actual condition code corresponds to the one specified. For example, the basic instruction for Branch on Condition True, Absolute (BCTA) is '1Czzzz', where zzzz is the absolute address to which we want to jump. As it stands, this branch instruction will *only* be carried out if the condition code is ØØ. Similarly '1Dzzzz' and '1Ezzzz' specify the condition codes Ø1 and 1Ø, respectively. Finally, '1Fzzzz' would seem to refer to a condition code 11, but this code doesn't exist. In fact the corresponding instruction is used for an *un*conditional branch: a branch that is always carried out, no matter what the condition code.
Branch (to Subroutine) on Condition False, Relative or Absolute. These four instructions are similar to those described above; the only difference is that the branch is executed if the actual condition code does *not* correspond to the one specified. The 'BSFA' instruction BCzzzz, for example, will cause a branch to subroutine if the condition code is either Ø1 or 1Ø, but *not* if it is ØØ. Note that no 'unconditional' variations of these instructions exist: the corresponding codes 9Byy, 9Fzzzz, BByy and BFzzzz are used for other instructions.
Branch (to Subroutine) on Register Non-Zero, Relative or Absolute. As part of these instructions, one of the registers (RØ ... R3) is specified. If the contents of this register is *not* zero, the branch instruction is carried out; otherwise it is ignored. 'BRNA, RØ' (5Czzzz), for instance, will cause a jump to address zzzz provided the data stored in Register Ø is not zero.
Branch on Incrementing (Decrementing) Register, Relative or Absolute. These instructions are an extension of the previous set. Once again, a register is specified. In this case, however, Ø1 is first added to (increment) or subtracted from (decrement) the contents of the register, after which the branch instruction is only carried out if the new contents are non-zero.

Table 7. **Branch (to subroutine)**

description		example	comments
Branch:			
On Condition True, Relative	(BCTR)	18yy	1Byy = unconditional
On Condition True, Absolute	(BCTA)	1Czzzz	1Fzzzz = unconditional
On Condition False, Relative	(BCFR)	98yy	9Byy: see below
On Condition False, Absolute	(BCFA)	9Czzzz	9Fzzzz: see below
On Register Non-zero, Rel.	(BRNR)	58yy	
On Register Non-zero, Abs.	(BRNA)	5Czzzz	
On Incrementing Register, Rel.	(BIRR)	D8yy	
On Incrementing Register, Abs.	(BIRA)	DCzzzz	
On Decrementing Register, Rel.	(BDRR)	F8yy	
On Decrementing Register, Abs.	(BDRA)	FCzzzz	
Zero Relative, Unconditional	(ZBRR)	9Byy	
Indexed Absolute, Unconditional	(BXA)	9Fzzzz	R3 only!
Branch to Subroutine:			
On Condition True, Relative	(BSTR)	38yy	3Byy = unconditional
On Condition True, Absolute	(BSTA)	3Czzzz	3Fzzzz = unconditional
On Condition False, Relative	(BSFR)	B8yy	BByy: see below
On Condition False, Absolute	(BSFA)	BCzzzz	BFzzzz: see below
On Register Non-zero, Rel.	(BSNR)	78yy	
On Register Non-zero, Abs.	(BSNA)	7Czzzz	
Zero Relative, Unconditional	(ZBSR)	BByy	
Indexed Absolute Unconditional	(BSXA)	BFzzzz	R3 only!
Return from subroutine:			
Conditional	(RETC)	14	
And Enable Interrupt, Conditional	(RETE)	34	

Note that no 'Branch-to-subroutine' version of these instructions exists.

Zero Branch (to Subroutine) Relative, Unconditional. These two instructions are relatively useless in the TV games computer, since they specify a branch relative to address *0000*: the start of the monitor program!

Branch (to Subroutine) Indexed, Absolute, Unconditional. These two instructions are *the only two indexed branch instructions* that exist. The value in the index register (which *must* be R3) is added to the basic absolute address given, and the branch is executed to the resultant address.

Return from subroutine, conditional. As before, a condition code is specified as part of this instruction; if the actual condition code matches, the subroutine is terminated. An unconditional end of the subroutine is indicated by the condition code 11, so the instruction RETC, UN is 17. A variation on this instruction exists (RETE) that not only ends the subroutine, but also resets the Interrupt Inhibit bit. Not a good idea, until one has gained enough experience to start using the interrupt facility...

The complete set of branch instructions is summarised in Table 7.

Program Status

The function of the various 'bits' in the Program Status Registers was

Table 8. **Program Status, Test, Compare, etc.**

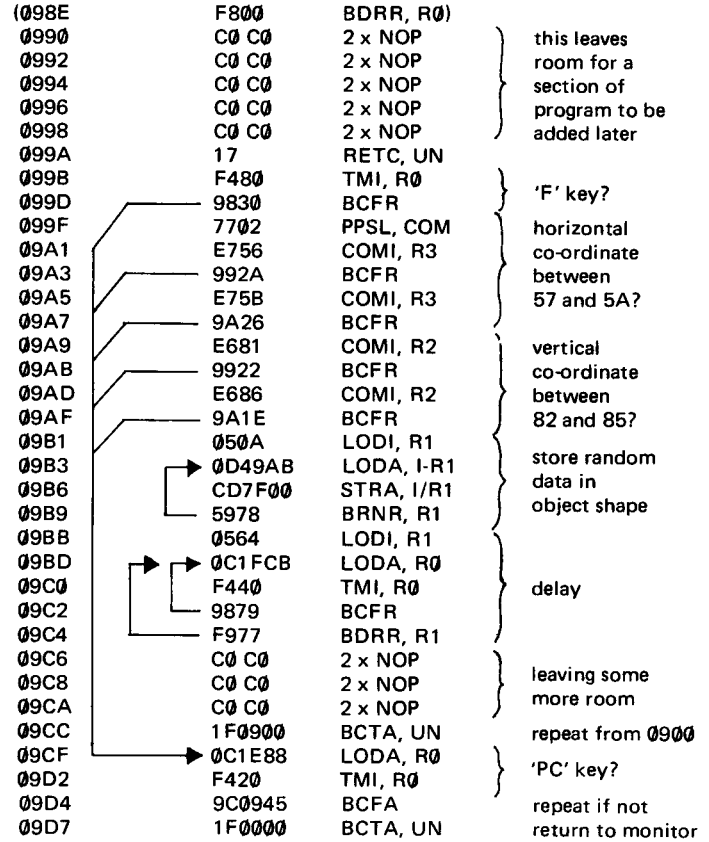
description		example	comments
Load Program Status, Upper	(LPSU)	92	from RØ
Load Program Status, Lower	(LPSL)	93	from RØ
Store Program Status, Upper	(SPSU)	12	to RØ
Store Program Status, Lower	(SPSL)	13	to RØ
Clear Program Status, Upper, Masked	(CPSU)	74 mm	mm = mask
Clear Program Status, Lower, Masked	(CPSL)	75 mm	mm = mask
Preset Program Status, Upper, Masked	(PPSU)	76 mm	mm = mask
Preset Program Status, Lower, Masked	(PPSL)	77 mm	mm = mask
Test Program Status, Upper, Masked	(TPSU)	B4mm	mm = mask
Test Program Status, Lower, Masked	(TPSL)	B5mm	mm = mask
Test Under Mask Immediate	(TMI)	F4mm . . . F7mm	RØ . . . R3
Compare to Register Zero	(COMZ)	EØ . . . E3	
Compare Immediate	(COMI)	E4xx . . . E7xx	xx = value
Compare Relative	(COMR)	E8yy . . . EByy	
Compare Absolute	(COMA)	ECzzzz . . . EFzzzz	
No Operation	(NOP)	CØ	
Halt	(HALT)	4Ø	

explained above. At this point, we are only interested in the available instructions (as summarised in Table 8).
The *Load* and *Store* instructions refer to data transfer between one of the Program Status Registers and Register Ø only. 'Load Program Status Upper' (LPSU: 92) for instance, loads the contents of RØ into the PSU.
In practice, these instructions will not be used often, since in most cases *Clear, Masked* or *Preset, Masked* instructions are more suitable. 'Clear Program Status Upper, Masked 4Ø' (744Ø) will clear the 'flag' bit, without having any effect on the other bits in the PSU. Similarly, 'PPSL, RS' (771Ø) will select the second register bank.
Finally, any bit (or combination of bits) in each of the program status registers can be *tested:* 'Test Program Status Upper, Masked 4Ø' (B44Ø) will cause the Condition Code to be set to ØØ if the 'flag' is set; otherwise the Condition Code will become 1Ø.

Test under Mask; Compare
With all the conditional branching facilities available, it is obviously useful to have instructions that set the Condition Code. Basically, all types of data transfer to or data manipulation in a register do this; furthermore, the *Test Under Mask Immediate* (TMI) and *Compare* (COM) instructions set the condition code bits without altering the data in any way.
The TMI instruction is the easiest to use: a register is specified in the first part of the instruction ('F4' for register zero, 'F5' for R1, and so on) and a 'mask' in the second part. The mask simply specifies the bits to be tested: '81', for instance, is 1ØØØ ØØØ1 in binary and so the first and last bits will be tested. If, in the data contained in the specified register, these two bits are

Table D.

With the complete program given so far (in tables A . . . C), it is possible to get the object into your sights. Now, what about shooting it down?!
First, modify the instruction in address Ø962: instead of '9CØ991', enter '9CØ99B'.
The existing program, from address Ø990 is then extended as follows:



Once the object is accurately (!) centered, it can now be 'shot to pieces' by operating the 'F' key.

'1s', the condition code will be set to ØØ; if not, CC will become 1Ø. An example. If the data in R1 is Ø5, the instruction F5Ø1 (TMI, R1, Ø1) will set the condition code to ØØ – the data, Ø5, is ØØØØ Ø1Ø1. By contrast, F581 will set the CC to 1Ø: ØØØØ Ø1Ø1.
The compare instruction is basically similar, but it is both more precise and more versatile – and also more complicated to use. In this case, a data value is specified instead of a mask, and the condition code can be set in three ways: Ø1 for 'greater than', ØØ for 'equals' and 1Ø for 'less than'.

There are two main points to watch, when using this instruction: what is meant by 'greater than' (data in register greater than data specified, or vice versa; see the footnotes in the Instruction Set) and *what type of comparison* is required. With the 'COM bit' in the PSL set to Ø, an 'arithmetical' comparison will be performed: all values from 8Ø to FF are treated as *negative* numbers (two's complement!). If the COM bit is set to 1 (by means of the instruction 77Ø2 = PPSL, COM) a 'logical' comparison will result: the data is treated as a positive 8-bit binary number.

No Operation
A surprisingly useful instruction, this! When the processor finds the code 'CØ' it simply carries on to the next instruction. There are two cases where this can be particularly useful: to 'delete' instructions that prove unnecessary, without having to re-enter the rest of the program, and to 'leave a gap' into which further instructions are to be added at a later date.

Halt
This stops the processor, quite drastically. The only way to start it up again is either to operate the 'reset' key or provide an interrupt – provided the interrupt inhibit bit is not set. In general, this is not a good idea; in the TV games computer, a 'return to Monitor' instruction (1FØØØØ = BCTA, UN, for instance) will usually be more suitable.

At this point, it is worth noting that all the instructions used in tables A...E have been discussed. If you have already tried this program, it will be clear why this chapter is headed 'The more important instructions'! If you haven't keyed it in yet, now's the time to do it...

Table E.

Finally, what about adding a time limit? As follows:

— first, fill in the space in the program starting at address 0990:

```
(098E      F800     BDRR, R0)
0990       7710     PPSL, RS      ┐
0992       E700     COMI, R3      │  set the 'clock' (R3 in the upper
0994       9802     BCFR          │  register bank) going as soon as the
0996       0709     LODI, R3      │  object is first moved
0998       7510     CPSL, RS      ┘
```

— similarly, fill in the space starting at 09C6:

```
(09C4      F977     BDRR, R1)
09C6       7710     PPSL, RS      ┐
09C8       0700     LODI, R3      │  set R3' to zero when
09CA       7510     CPSL, RS      ┘  the object is hit
```

— modify the data at address 0981: instead of '1F0945', the instruction becomes '1F09DA'.

— at address 09D4, the instruction is modified to '9C0976' (instead of 9C0945).

— the program is extended, from address 09DA on, as follows:

```
(09D7      1F0000   BCTA, UN)
09DA       7710     PPSL, RS      ┐
09DC       E700     COMI, R3      │
09DE       9804     BCFR          │  if 'clock' stopped (R3' = 00),
09E0       0619     LODI, R2      │  preset R2' for one-second count
09E2       1B0B     BCTR, UN      ┘
09E4       CF1FC9   STRA, R3      ┐  update score and
09E7       FA06     BDRR, R2      ┘  decrement R2'
09E9       0619     LODI, R2      ┐  reset R2' and
09EB       FB02     BDRR, R3      ┘  decrement R3'
09ED       1B05     BCTR, UN         branch if R3' = 00
09EF       7510     CPSL, RS      ┐  repeat key check routine
09F1       1F0945   BSTA, UN      ┘
09F4       CF1FC9   STRA, R3      ┐
09F7       04FF     LODI, R0      │  store 00 in score, make screen
09F9       CC1FC6   STRA, R0      │  white ('you lose'!) and repeat
09FC       7510     CPSL, RS      │  via delay routine
09FE       1F09BB   BSTA, UN      ┘
```

9

Programming tips

Getting started.

The instructions explained in the previous chapter are sufficient for simple programs. The remaining facilities will be dealt with in chapter 10. First, however, it is more interesting to experiment with the computer, and a few practical programming tips should prove useful.

First and foremost: remember to block the Interrupt facility if this is not required in the program! For the time being, it is advisable to start every program with the instruction '762Ø' (PPSU, II).
There are several ways to end a program. Usually, one of the keys ('PC', for instance) is used to initiate a jump back to Monitor. The jump itself can be done in several ways. The most obvious is a Branch Absolute instruction, '1FØØØØ', and this is what we have been using so far. A shorter version is the ZBRR instruction '9BØØ'. In either case it should be noted that returning to monitor via address ØØØØ always causes the data in RØ to become Ø9. If this is undesirable, an alternative return to monitor routine can be used:

```
3FØ5CD    BSTA, UN
1FØØ14    BCTA, UN
```

Don't ask us to explain this one – that would require an extensive discussion of the monitor software!
When it comes to the program itself, the first thing is to work out what you want to do. Obviously. For simple programs, this can usually be put into words quite easily. The program given in Table 5 in the previous chapter, for example, was originally specified as 'clear objects; define background colour; load FF in all background bytes'. For more complicated programs, some more extensive advance plotting may be required – using a flow chart, say – but usually a complicated program can be 'broken up' into several simple routines. These can each be tried and tested individually, before 'tacking them together' to obtain the complete final program.
As each semi-complete routine is entered, it is highly advisable to store it on tape *before* the first test run. This lesson was learned the hard way, when one misplaced relative address caused 'garbage' to be stored at all sorts of awkward places throughout the program. The only solution was to laboriously re-enter the whole program...

Planning a program

The few details given so far (how to start a program, how to end it, and how to break it up into small sections) are of fundamental importance – even to the happy few who can work out a complete program as they go along, keying it straight in without putting anything on paper first. For most ordinary mortals, however, some initial paper planning is just as essential. There are several ways to go about this, depending on personal preference and skill. A good way to start is to jot down, in a few lines, what you would like to do. For example:

– objects 1 and 2 are vertical lines;
– clear other objects;
– set vertical position of objects 1 and 2 to mid-screen;
– horizontal movement only, via – and + keys for object 1, and PC and MEM keys for object 2;
– when objects collide, change screen colour to red; otherwise leave it blue.

At this point, you can either start writing out the necessary instructions (as mnemonics or in 'machine language': the hex codes for the various instructions) or else draw up a 'flow chart'. The latter quite literally charts the 'flow' of the program, using a few basic symbols: a circle for begin and end, a rectangle for an operation (with two lines at each end if it's a subroutine) and a diamond where some kind of choice is made between two possible continuations. All these symbols are used in figure 26: the first attempt at the program outlined above.

Actually, the system used in figure 26 is just about the best way we know of for writing down a program. It combines everything: the addresses (in a kind of shorthand!), the actual instructions, the mnemonics (as you gain experience, you will probably start to omit these...) and the flow chart. Let's run through the program briefly:

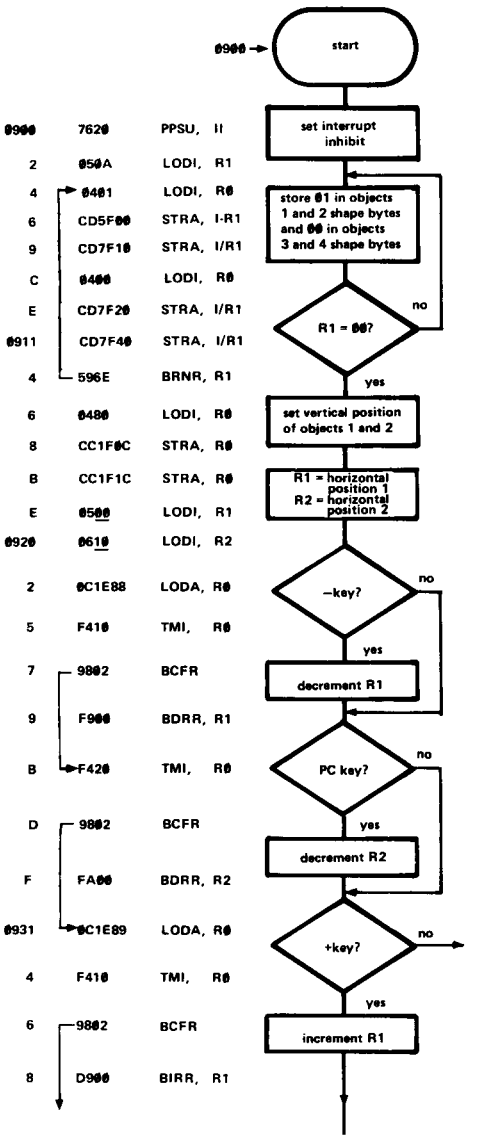
First the interrupt inhibit bit is set. Then, using indexed addressing, 'Ø1' is stored in objects 1 and 2 (from 1FØ9 down to 1FØØ and from 1F19 down to 1F1Ø) and 'ØØ' in objects 3 and 4, to clear them. The BRNR instruction, at address Ø914, keeps the processor running round this loop until all ten bytes have been stored in each object. After this, the vertical position of objects 1 and 2 is set; this completes the first three steps in the thumbnail program description given above.

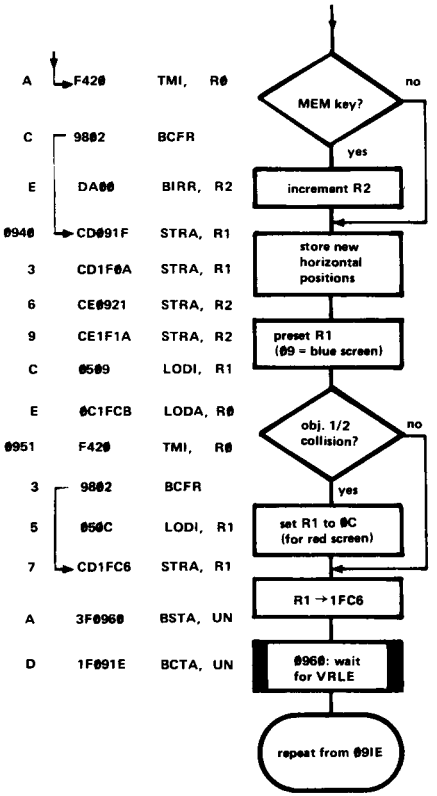
The next step is to provide movement. The initial positions are loaded into R1 and R2 – the data bytes Ø91F and Ø921 will be modified later on in the program, which is why they are underlined. Then we run through an extensive keyscan procedure, incrementing or decrementing R1 or R2 as required. This brings us down to address Ø94Ø, where the new data is stored in the 'scratch' bytes (Ø91F and Ø921) and in the horizontal position addresses (1FØA and 1F1A).

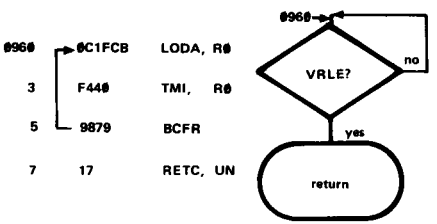
Now we come to the step 'when objects collide, change screen colour'. This is done by storing either 'Ø9' or 'ØC' in 1FC6, depending on whether the 'object 1/2 collision' bit in 1FCB is set or not. Finally, to keep the object speed down to a reasonable level, we branch to a subroutine 'wait for

VRLE'; as soon as the VRLE bit is set (at the end of the frame), the program returns from the subroutine and then branches back to Ø91E for the next keyscan.
This all looks all right, so we can key it in. Done? Tried?... If so, you will

Figure 26.



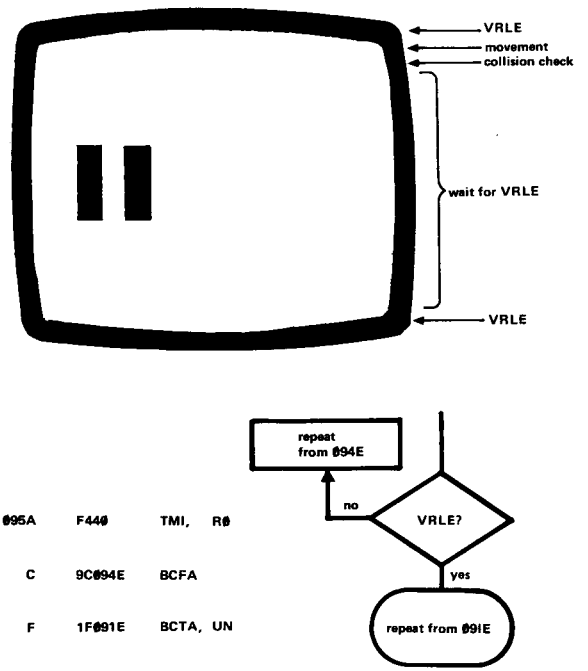




Start address: Ø9ØØ
Return to monitor via Reset key

82901-26

Figure 27.



find that it only works in part! The objects move, but the screen colour doesn't change when they collide! Why is that? Let's think – and look at figure 27 while we're at it. What we are actually doing, immediately after VRLE (at the top of the screen) is to run the movement routine – immediately followed by the collision check. At this point, the objects haven't collided! Then we simply 'wait for VRLE'. We should be doing two things: check for object 1/2 collision and wait for VRLE! The program can easily be modified to do this, by changing the instructions from Ø95A on as shown in figure 27. Note that the subroutine is no longer used – it was only included in the original program to illustrate the flow chart symbol...

De-bugging a program

When it comes to 'de-bugging' a program – and it always does, no program works perfectly first time! – there are several tricks that can prove useful.

The first thing to do, obviously, is *think!* The program described above is an example: most of it worked, giving moving objects, but either the collision detection or the colour change routine didn't work. After a little thought, the error was obvious. Unfortunately, however, program errors

are rarely that obvious until you find them (**then** they are suddenly glaringly obvious...!). So some inspired trouble-shooting is required.
We can use the same program to illustrate the possibilities. As a first example, modify the instruction at address Ø914 to '597E' – to simulate a keying error. Try running the program now, and you will find that the text 'PC = 0900' jumps to the top of the screen and stays there; nothing else happens, and the keys have no effect. This sort of thing usually indicates that the program is 'stuck in a loop' – the question is: where? In this case, it's fairly obvious: the object shapes are unchanged, so the first loop isn't doing its job. However, there is a more general method for tracking down this type of problem: use the 'Breakpoint' routine. There are two important points to watch:
– the Breakpoint address given *must be the first address* of an instruction. For instance, in the following section of program:

```
Ø900    7620    PPSU, II
Ø902    0400    LODI, RØ
Ø904    0605    LODI, R2
```

breakpoints can be specified at addresses Ø900, Ø902 and Ø904, but *not* at Ø901, Ø903 or Ø905!
– breakpoints modify the program at that point. If the breakpoint is found in the normal way, the original data will be restored automatically. However, if things really go wrong so that the reset key must be used to return to Monitor, it may be necessary to restore the data at the breakpoint by hand!
Putting this theory into practice, in our 'bugged' program: insert a breakpoint at Ø916 (this is done by operating the BP key, entering the desired address – Ø916 – followed by the + key) and start the program. No change! This means that the breakpoint wasn't reached. Now try a breakpoint at Ø904 (and remember to check the data at the previous breakpoint, Ø916!). This breakpoint will be found, indicating that the program enters this loop. The error must therefore be between address Ø904 and Ø916. Checking the instructions in this section will pinpoint the keying error in no time.
Another de-bugging trick is to insert a temporary subroutine, adding a further check. For example, it might be useful to keep track of the horizontal position of one of the objects. This can be achieved by modifying the instruction at address Ø943 to '3FØAØØ' and adding the following routine:

```
ØAØØ  CD1FØA   STRA,  R1
   3  Ø1       LODZ,  R1
   4  4488     ANDI,  RØ
   6  CC1FC8   STRA,  RØ
   9  4577     ANDI,  R1
   B  CD1FC9   STRA,  R1
   E  17       RETC,  UN
```

The actual instructions used here will be explained in a following chapter, but it is the basic idea that we are interested in now. First, the original

instruction from address Ø943 is restored at address ØAØØ. Then the horizontal position of object 1 is loaded into the score display. Since this can only cater for digits from Ø to 9, the position data is split into an upper part (the highest bit for each digit, corresponding to Ø or 8) which is stored in the left-hand score display, and a lower section (corresponding to values Ø to 7) for the right-hand display. Adding the two scores gives the total hex digit.

One further useful trick is to temporarily delete a section of program. As an example, we can insert another error: modify the instruction at Ø957 to read CØ1FC6. Be warned: this may create absolute chaos, so store the program on tape before running it! Then try it out... Something wrong, isn't there? Now, after re-loading the (incorrect) program from tape, 'cut off the tail' by modifying Ø94C to read 1FØ91E. This deletes the 'change background colour' and the 'wait for VRLE' routines – with the result that the objects will fly across the screen at terrific speed. However, the real program error is now clearly located: it must be in the section that we deleted.

Enough on programming and trouble-shooting for now. We'll come back on this subject later. First, there are a few other matters that need some further explanation.

The PVI and keyboard

The main points regarding the PVI were all discussed in earlier chapters. One point, however, did not receive all the attention it deserves:

The 'VRLE' bit, at address 1FCB, goes high at the end of each frame; it is reset at the end of the VRST pulse *or when read.* This means that it can only be read as '1' *once* for each frame. As an example, a simple 'delay' routine is given in Table 9. Basically, what happens is that the processor waits until it finds VRLE = 1; it then decrements the value in R2 and repeats the VRLE scan if R2 is not yet zero. The result is a delay, approximately equal to the value in R2 times the frame period (20 ms). By way of demonstration, this routine can be included inside the 'load background' loop in Table 5 (chapter 8). The result is given in Table 10.

Finally, the keyboard. Each column corresponds to one address: 1E88 for the column above the '–' key, 1E89 and 1E8A for the next two columns, 1E8B for the column that includes the 'reset' key (note that this key itself is *not* scanned in the keyboard layout suggested!) and 1E8C ... 1E8E for the last three columns. When reading the keyboard in this way, the four left-hand bits retrieved as data correspond to the four keys in the column – and the other four bits are all ones! '1F' at address 1E88, for instance, means that the '–' key is operated; '4F' at 1E8A corresponds to key '8'. Note that contact bounce can sometimes be a problem with this type of rapid key-scan. A more sophisticated routine, using part of the Monitor software, will be described further on.

Table 9.

```
      060A    LODI, R2
   ┌─►┌─► 0F1FCB  LODA, R3 }  wait   }  total delay:
   │  │    F740    TMI, R3 }  for    }  approximately
   │  └──  9879    BCFR, ≠ }  VRLE   }  0.2 seconds
   └─────  FA77    BDRR, R2
```

Table 10.

```
0900        7620    PPSU, I1
0902        05AD    LODI, R1
0904        0400    LODI, R0   }  clear objects
0906     ┌─►CD5F00  STRA, I-R1 }  and background
0909     └──597B    BRNR, R1
090B        0469    LODI, R0   }  colour
090D        CC1FC6  STRA, R0   }
0910        052D    LODI, R1   }
0912        04FF    LODI, R0   }
0914  ┌─────►CD5F80  STRA, I-R1 }
0917  │     060A    LODI, R2   }
0919  │  ┌─►0F1FCB  LODA, R3   }  delay   }  load
091C  │  │  F740    TMI, R3    }          }  background
091E  │  └──9879    BCFR, ≠    }          }
0920  │     FA77    BDRR, R2   }
0922  └─────5970    BRNR, R1   }
0924        40      HALT
```

In conclusion

With the information given in this chapter, it is possible to write simple programs. Now is the time to start practicing – before we discuss the rest of the instruction set, and give some rather more complicated routines...

Table 11

```
0900    7620      PPSU, II
0902    3F0161    BSTA, UN      (clear/initiate PVI)
0905    0630      LODI, R2
0907    0508      LODI, R1
0909    0E492D    LODA, I-R2    (data)
090C    CD4890    STRA, I-R1
090F    5978      BRNR, R1
0911    7710      PPSL, RS
0913    3F020E    BSTA, UN      (load MLINE)
0916    7510      CPSL, RS
0918    5A0A      BRNR, R2
091A    0C1E89    LODA, R0      wait for
091D    F410      TMI, R0       '+' key
091F    1879      BCTR          release
0921    1F0038    BCTA, UN      return to monitor
0924    7710      PPSL, RS
0926    3F02CF    BSTA, UN      (scroll)
0929    7510      CPSL, RS
093B    1B5A      BCTR, UN
```

```
092D    17 A2 A2 A2 A2 A2 A2 17    sixth line
0935    17 17 10 00 00 0D 17 17    fifth line
093D    0A 17 11 00 BC 17 00 0F    fourth line     } DATA
0945    17 17 0D 00 0E 05 17 17    third line      }
094D    14 15 0A 0C BC 12 0C 0E    second line     }
0955    0A 17 11 12 BC BC 11 0E    first line
```

Just to whet the appetite: the gimmicks used in this program will be discussed in the next chapter! The start address is 0900.

10

The rest of the instructions

So far, we have examined the basic principles of the TV games computer, and discussed the more important instructions. In this chapter we will take a closer look at the rest of the instruction set and explain some useful programming tricks.

The Load, Store, Branch, Compare, 'Miscellaneous' and 'Program Status' instructions were all dealt with in chapter 8. As illustrated in Tables A ... E, these instructions are sufficient for quite interesting little programs. However, as the extended version of the same program on the ESS 007 cassette illustrates, programs can be made rather more sophisticated by the use of the remaining instructions: Arithmetic, Logical and Rotate. (The Input/Output instructions cannot be used in the basic version of the TV games computer).

Arithmetic

Even though the computer will not normally be required to do sums, the so-called arithmetical instructions are quite useful. As shown in Table 12, a complete set of add and subtract instructions are available; the only other instruction under this heading is 'decimal adjust register'.
Both addition and subtraction are straightforward:
Ø3 + Ø5 = 8; 19 − Ø2 = 17; 28 + 13 = 3B; and so on. The calculations are performed in 8-bit true binary and negative numbers are two's complement, so that the hexadecimal calculations are valid. As a result of these calculations, three bits in the Program Status Lower will be set or reset:

- The Carry/Borrow bit (C): to be precise, this is set to 1 by an addition that generates a carry, and to Ø by a subtraction that generates a borrow. However, in most cases it is sufficient to know that this bit will be interpreted correctly in any following add or subtract operation, provided the 'with carry' bit (bit 3 in the PSL) is set. If the WC bit is not set, Carry or Borrow information is ignored – in practice, this has proved even more useful!
- The Inter-Digit Carry bit (IDC): this gives the Carry or Borrow information that applies between the lower four and the upper four bits in the register affected. This information can be ignored when binary arithmetic is performed, but it may be essential when doing decimal calculations.
- The Overflow bit (OVF): since large numbers (greater than 7F) can be interpreted as negative numbers, things can go wrong in an addition. For

Table 12

Arithmetic

description		example	comments
Add to register Zero	(ADDZ)	81	RØ: = R1 + RØ
Add Immediate	(ADDI)	84xx	xx = data
Add Relative	(ADDR)	88yy	yy = displacement
Add Absolute	(ADDA)	8Czzzz	zzzz = address
Subtract from register Zero	(SUBZ)	A1	RØ: = RØ − R1
Subtract Immediate	(SUBI)	A4xx	xx = data
Subtract Relative	(SUBR)	A8yy	yy = displacement
Subtract Absolute	(SUBA)	ACzzzz	zzzz = address
Decimal Adjust Register	(DAR)	94	

instance, 7Ø + 28 will give the result 98 − but this is equivalent to a *negative* number (−68). This type of ambiguous result is indicated by the setting of the overflow bit: if two positive numbers are added or subtracted and the result is 'negative' the OVF bit is set. Similarly, if a positive result is obtained from a calculation on two negative numbers.

So much for addition and subtraction. In practice, it is often sufficient to know that clearing the 'WC' bit results in a straightforward calculation, without any unexpected 'carries' or 'borrows'.

Decimal Adjust Register

This instruction allows BCD sign magnitude arithmetic to be performed on packed digits. Exactly how this instruction works is not so important: the main thing is to know how to use it! Depending on the setting of the Carry and Interdigit-carry bits, a two-digit hexadecimal number in a register is converted into the corresponding two-digit decimal version. Since the carry bits must be set, this instruction must be used in conjunction with an addition (or subtraction); in fact the most common case is addition, so we'll use that for a first example:

```
0900  7620    PPSU, II
   2  0403    LODI, RØ
   4  CC1FC3  STRA, RØ
   7  0400    LODI, RØ
   9  ØD1E89  LODA, R1
   C  F510    TMI, R1
   E  9805    BCFR
0910  8401    ADDI, RØ
   2  8466    ADDI, RØ
   4  94      DAR, RØ
   5  CC1FC9  STRA, RØ
   8  C86E    STRR, RØ
   A  0510    LODI, R1
```

```
    C ┌ØC1FCB    LODA, RØ
    F │ F44Ø     TMI, RØ
 Ø921 ├ 9879     BCFR
    3 └ F977     BDRR, R1
    5   1FØ9Ø7   BCTA, UN
```

All the padding is just to obtain an interesting result: a score display that increments as long as the '+' key is operated. However, the section that we are interested in starts at address Ø91Ø: add one to the score, then add '66' and perform the decimal adjust operation. In this case, it would also be possible to replace the two add instructions (84Ø1 and 8466) by a single one: 8467.
Adding the value 66 before the decimal adjust operation may seem strange – but that's how it's done! In effect, what happens is this: if either digit is in the range Ø...9, adding 6 to it will not cause the corresponding carry (or interdigit carry) bit to be set, since the result can never be greater than 'F' (fifteen). In this case, the following decimal adjust operation will subtract 6 from the digit – so that the original value is restored. However, when a digit is greater than 9 (A...F, in other words) adding 6 will give a result that is 1Ø or more (A + 6 = 1Ø, B + 6 = 11, and so on). Now the carry bit will be set, and the following decimal adjust operation leaves the new result as it is!

By way of further illustration, we can make use of the carry facility for a four-digit score display. In the program given above, modify the instructions from Ø91Ø on to read '3FØ928 – CØ – CØ' and then add:

```
 Ø928   77Ø8     PPSL, WC
    A   7521     CPSL, IDC + C
    C   Ø5ØØ     LODI, R1
    E   8467     ADDI, RØ
 Ø93Ø   94       DAR, RØ
    1   8566     ADDI, R1
    3   95       DAR, R1
    4   75Ø8     CPSL, WC
    6   C975     STRR, R1
    8   CD1FC8   STRA, R1
    B   17       RETC, UN
```

The trick here is that the operations on R1 – the addition, in particular – follow those on RØ. If RØ becomes greater than 99 the carry bit will be set, and this causes the value in R1 to be increased by one.
Subtraction is even easier, since in this case the 'add 66' step is not required. In the first program given above, the changes are:

```
 Ø91Ø   A4Ø1     SUBI, RØ
    2   CØ CØ    2x NOP
```

In the extended version, the carry bits must be set initially – this corresponds to 'no borrow'! The changes are:

```
0928   7729      PPSL, IDC/WC/C
   A   C0 C0     2x NOP
   C   0500      LODI, R1
   E   A401      SUBI, R0
0930   94        DAR, R0
   1   A500      SUBI, R1
```

Table 13

		Logic example	comments
AND to register Zero	(ANDZ)	41	R ≠ R0
AND Immediate	(ANDI)	44xx	xx = data
AND Relative	(ANDR)	48yy	yy = displacement
AND Absolute	(ANDA)	4Czzzz	zzzz = address
Inclusive Or to register Zero	(IORZ)	61	
Inclusive Or Immediate	(IORI)	64xx	xx = data
Inclusive Or Relative	(IORR)	68yy	yy = displacement
Inclusive Or Absolute	(IORA)	6Czzzz	zzzz = address
Exclusive Or to register Zero	(EORZ)	21	
Exclusive Or Immediate	(EORI)	24xx	xx = data
Exclusive Or Relative	(EORR)	28yy	yy = displacement
Exclusive Or Absolute	(EORA)	2Czzzz	zzzz = address

Logic

The instruction set includes AND, Inclusive Or (IOR) and Exclusive Or (EOR) instructions, as summarised in Table 13. The corresponding logic operations are given in Table 14; for most practical applications, it is easier to describe the effects in words:

AND

An AND instruction causes two groups of 8 bits to be compared; in the result, only those bits will be logic 1 that were 1 in both of the original groups. This instruction can therefore be used as a 'data mask'. As an example, assume that some type of delay routine or 'clock' is counting in R3, and that the three least significant bits are used to determine the screen colour. This can be achieved as follows:

```
03       LODZ, R3
4407     ANDI, R0
8408     ADDI, R0
CC1FC6   STRA, R0
```

After 'screening out' the five higher bits by means of the AND instruction, the 'Background enable' bit is added, and the result stored in the PVI.

Table 14

Logic operations

The logic operations treat each corresponding pair of bits in the two specified (8-bit) data bytes according to the following truth tables:

	Bit A (Ø . . . 7)	Bit B (Ø . . . 7)	Result
AND	Ø	Ø	Ø
	Ø	1	Ø
	1	Ø	Ø
	1	1	1
IOR	Ø	Ø	Ø
	Ø	1	1
	1	Ø	1
	1	1	1
EOR	Ø	Ø	Ø
	Ø	1	1
	1	Ø	1
	1	1	Ø

Examples

In the following examples, the original data in Register Zero is assumed to be ØF.

'ANDI, RØ, 33' (4433): data A = ØF = ØØØØ 1111
data B = 33 = ØØ11 ØØ11
result = Ø3 = ØØØØ ØØ11

'IORI, RØ, 33' (6433): data A = ØF = ØØØØ 1111
data B = 33 = ØØ11 ØØ11
result = 3F = ØØ11 1111

'EORI, RØ, 33' (2433): data A = ØF = ØØØØ 1111
data B = 33 = ØØ11 ØØ11
result = 3C = ØØ11 11ØØ

Note that all three logic operations can also be considered as 'bit-mask' operations. After an AND operation, only those bits in the original data (data A) remain logic 1 that were specified by the ones in the bit mask (data B). Conversely, after an Inclusive Or instruction only those bits in data A will remain logic Ø that were specified as 'of interest' by the zeroes in data B. Finally, an Exclusive Or operation causes those bits in data A to be inverted that correspond to the ones in data B.

Inclusive Or

Once again, two groups of eight bits are compared; in this case, however, all bits that are logic 1 in either of the two groups will be 1 in the result. Another way of looking at this is to say that only those bits will be logic Ø in the result that were Ø in both of the original groups. A complementary data mask, in other words!

Both AND and IOR instructions can also be used to set or reset one or more bits in a group of eight, without affecting the others. In the example given above, for instance, if the contents of R3 are to be used for both screen and background colours:

```
Ø3          LODZ, R3
6408        IORI, RØ
CC1FC6      STRA, RØ
```

The Inclusive Or instruction is added to ensure that the Background enable bit is always set.

Exclusive Or

Quite apart from its 'logical' function, this instruction can be used as a 'selective inverter'. If we take one group of 8 bits as the original data and exor it to a second group, the result will be that some of the bits in the first group will be inverted, as specified in the second group. Complicated? Not really. Each bit in one group specifies what happens to its partner in the other: if it's logic 1, the partner is inverted; if it's logic Ø, the corresponding bit in the other group is not affected. A few examples. Let us assume that the 'data' (i.e. one of the two groups of 8 bits) is FF in all cases: 1111 1111. 'EOR, FF' will invert all bits, giving ØØ as result. Similarly, 'EOR, CØ' will invert the first two bits (CØ = 11ØØ ØØØØ), so the result will be ØØ11 1111 = 3F.

Finally, a more practical example. As mentioned in chapter 9, scanning one column of the keyboard will always give a logic 1 for the four least significant bits. The 'C' key, for instance, (column address 1E8A) is decoded as 8F. This unwanted data can be removed as follows:

```
ØC1E8A      LODA, RØ
240F        EORI, RØ
```

Note that it is just as easy (and perhaps more 'logical') in this case to use an AND instruction as data mask: 'ANDI, FØ' will produce the same result.

Rotate

The 'Rotate Register Right' and 'Rotate Register Left' instructions do exactly that: the data in the specified register is shifted one place to the left or right, respectively. If the 'With Carry' bit in the PSL is reset, the data will shift around the loop – out one end and in the other. When the WC bit is set, however, things get rather more complicated: the 'Carry' and 'Interdigit Carry' bits also come into play. Fortunately, there is no need for a long-winded explanation: Figure 28 illustrates all the possibilities!

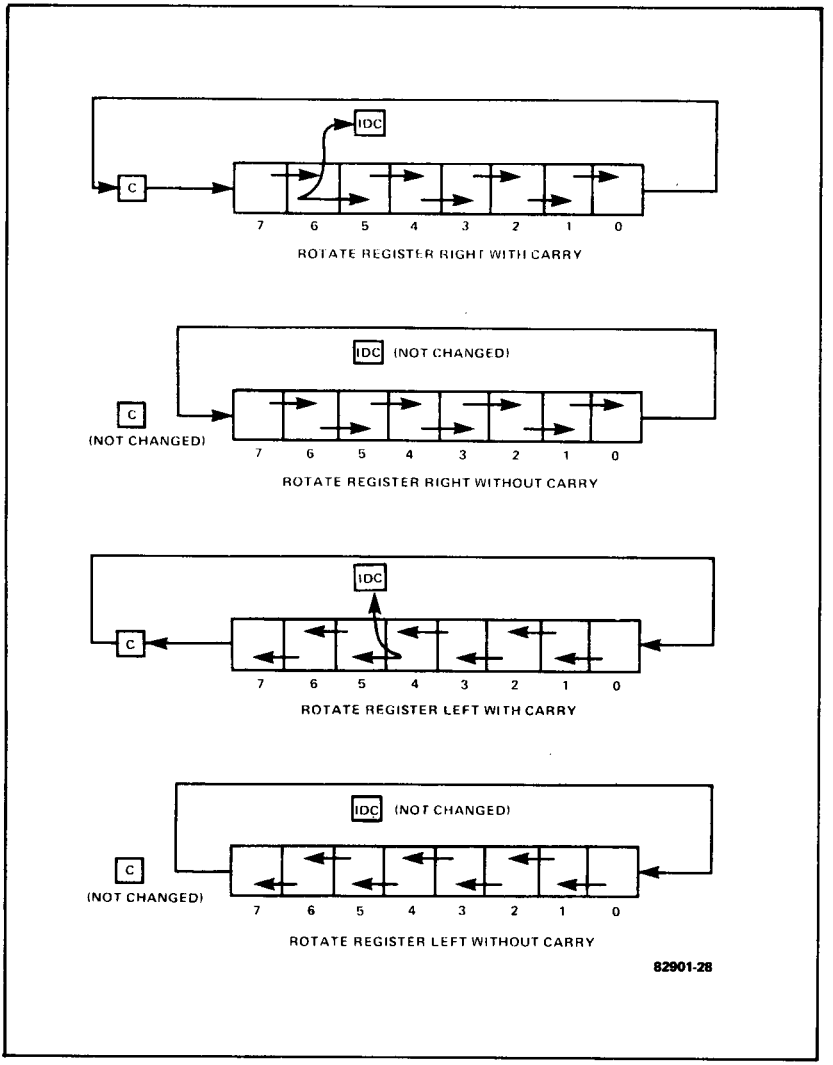


Figure 28. The effects of Rotate instructions.

Tricks and gimmicks

EORZ, RØ

In machine language: '2Ø'. The data in register zero is exored to the data in register zero; this means that if a bit is logic 1 it will be inverted, but any logic Ø's will be left alone. The result? ØØ in register Ø! The advantage is that this instruction is one byte shorter than the equivalent 'Ø4ØØ', for LODI, RØ.

IORZ, RØ
This instruction ('60' in machine language) has no effect on the data in register zero. However, an operation *is* performed – even if it has no effect – and so the Condition Code bits are set according to the data in RØ: Ø1 for 'positive', ØØ for 'zero' and 1Ø for 'negative'.

Multiplication and division
Rotating the data in a register one step left is equivalent to multiplying by two (provided no overflow occurs, but that can be checked). Similarly, shifting one step right is a division. What about multiplying by three? No problem:

```
C1      STRZ, R1
D1      RRL, R1
81      ADDZ, R1
```

Job done.
The original data, in register zero, is copied into register one; after multiplication by two, it is added to the original data in register zero.

LODI as scratch
In the course of a program, it is often necessary to update certain data at regular intervals. An obvious example is using the 'score' bytes for a time display. Once new data is loaded in the PVI, it can be left there indefinitely and the 'score' will remain unchanged. However, the awkward thing is that this data can not be read back from the PVI when a new update is required. The only solution is to keep track of the PVI data by also storing it at some point in the 'normal' memory. When a time update is required, the present data are retrieved from this 'scratch-pad memory', updated, and the new data stored both in the PVI and in the memory scratch.
All this is nothing new. However, in practice one little trick has proved useful. Since the program itself is stored in random access memory, there is nothing to stop you modifying instructions in the course of the program. If we assume, for instance, that the data in register one is to be added to the existing data, this can be achieved as follows:

```
0400    LODI, RØ
81      ADDZ, R1
C87C    STRR, RØ
CC1FC9  STRA, RØ
```

The second part of the Load Immediate instruction is used as 'scratch', so the existing data is loaded into RØ when the first instruction is carried out. The data in R1 is then added, and the new time information is restored in the scratch. Finally, the same new information is transferred to the PVI.
Compare this routine to a more 'normal' one, using address Ø8CØ, say, as scratch-pad memory:

```
ØC08CØ    LODA, RØ
81        ADDZ, R1
CC08CØ    STRA, RØ
CC1FC9    STRA, RØ
Ø8CØ =    scratch
```

Admittedly, the third instruction can be replaced by a 'Store Relative, Indirect' version (C8FB, to be precise) – but even so, this routine is noticably longer than the one given above.

Modifying Absolute addresses

The same trick described above can be used to modify an absolute address as required in the course of a program. The test pattern program on the ESS 007 cassette, for instance, uses this system to load a whole string of initial data into the PVI. The corresponding section of program (with a few modifications, to give a more interesting result!) is given in Table 15.

During each pass through the loop, the following sequence is carried out. First, the second byte of the desired absolute address is retrieved from the 'data store' ('LODA, I-R1') and stored at address Ø9D5 – i.e. the third byte of the STRA instruction. Then the data is retrieved (the second LODA, I-R1 instruction) and stored in the PVI at the address currently specified. Note that this address is *not* 1FØØ, no matter what the listing says: '1Fxx' would be more accurate, where xx is the address data retrieved by the first LODA, I-R1 instruction.

There are, of course, all sorts of variations on the same principle. The thing to realise is that it can be very useful to modify actual instructions in the course of a program. Bearing this in mind, practical examples will be found regularly when developing programs!

Table 15

```
Ø9C7    7620     PPSU, II
Ø9C9    Ø56E     LODI, R1
Ø9CB    ØD49E2   LODA, I-R1     (address)
Ø9CE    C805     STRR, RØ
Ø9DØ    ØD49E2   LODA, I-R1     (data)
Ø9D3    CC1FØØ   STRA, RØ       (Ø9D5 = scratch)
Ø9D6    5973     BRNR, R1
Ø9D8    ØC1E88   LODA, RØ
Ø9DB    F420     TMI, RØ        Return to
Ø9DD    9879     BCFR           monitor if 'PC'
Ø9DF    1FØØØØ   BCTA, UN
```

(continued on next page)

09E2	50 0C	data-address	
09E4	50 1C	data-address	VC 1 . . . 4
09E6	50 2C	data-address	
09E8	50 4C	data-address	
09EA	FE 0D	data-address	
09EC	FE 1D	data-address	VODI . . . 4
09EE	FE 2D	data-address	
09F0	FE 4D	data-address	
09F2	22 0A	data-address	
09F4	42 1A	data-address	HC 1 . . . 4
09F6	62 2A	data-address	
09F8	82 4A	data-address	
09FA	AA C0	data-address	size
09FC	09 C1	data-address	
09FE	09 C2	data-address	colour
0A00	19 C6	data-address	
0A02	00 00	data-address	
0A04	00 01	data-address	
0A06	00 02	data-address	
0A08	74 03	data-address	
0A0A	44 04	data-address	SHAPE 1
0A0C	74 05	data-address	
0A0E	44 06	data-address	
0A10	44 07	data-address	
0A12	77 08	data-address	
0A14	00 09	data-address	
0A16	00 10	data-address	
0A18	00 11	data-address	
0A1A	00 12	data-address	
0A1C	75 13	data-address	
0A1E	45 14	data-address	SHAPE 2
0A20	76 15	data-address	
0A22	45 16	data-address	
0A24	45 17	data-address	
0A26	75 18	data-address	
0A28	00 19	data-address	
0A2A	00 20	data-address	
0A2C	00 21	data-address	
0A2E	00 22	data-address	
0A30	77 23	data-address	
0A32	25 24	data-address	SHAPE 3
0A34	25 25	data-address	
0A36	25 26	data-address	
0A38	25 27	data-address	
0A3A	27 28	data-address	
0A3C	00 29	data-address	
0A3E	00 40	data-address	
0A40	00 41	data-address	
0A42	00 42	data-address	
0A44	70 43	data-address	
0A46	50 44	data-address	SHAPE 4
0A48	60 45	data-address	
0A4A	50 46	data-address	
0A4C	50 47	data-address	
0A4E	50 48	data-address	
0A50	00 49	data-address	

Start address: 09C7. Return to monitor by operating PC key.

11

Using monitor routines

The complete monitor software is stored in ROM, so there is no way to change it. However, it is stored at normal memory addresses, so there is nothing to stop you using monitor subroutines as part of a different program. In most cases, the only restriction is that the monitor routine must end with an unconditional return instruction (RETC, UN = 17). Furthermore, initial data must sometimes be set up correctly before starting the monitor routine. However, even with these restrictions, we have drawn up an extensive list of useful subroutines.

Keyboard scan

A complete keyboard scan routine, including contact debouncing and double-key reject, starts at address Ø181. As it stands, it uses the lower register bank. If this is awkward, the routine can be started at address Ø183 after clearing the 'With Carry' and 'Carry' bits in the Program Status Lower.

Two further points must be noted: the routine must be repeated twice in succession (preferably at consecutive frames, using the VRLE bit); furthermore, memory location Ø89F must be cleared before the first scan. A complete routine is given in Table 16. After the presets and a 'wait for VRLE' routine, the first scan is requested: '3FØ183 BSTA, UN'.
After the scan, the two highest bits in R1 indicate the 'scan status'. If bit 6 is at logic 1, this was the first scan and so a further scan is required; the program branches back to the 'wait for VRLE' routine. After the second scan, bit 6 is at logic Ø and bit 7 indicates whether one key was depressed during the two scans: it is one if this is the case, and zero if no key or two or more keys were operated. Note that 'key operated' (bit 7 is logic 1) corresponds to a negative number, so the condition code will be set to 1Ø.
A further possibility, not used in this routine, is to reset only bit 7 at address Ø89F. Bit 5 in R1 will then indicate if a key is (still) depressed.
To get back to the routine given in Table 16, after the second scan (when reaching address ØFE6, in other words) the lower five bits in R1 give the number of the operated key. The corresponding hexadecimal numbers are listed in figure 29a; the indications at the top left-hand corner correspond to the key indications suggested for the monitor routine. It should be noted that these numbers are *only* valid if bit 7 in R1 is logic 1, as mentioned above; otherwise, '*ØØ*' will appear if the data at address Ø89F was cleared completely, or else the previous key code if only bit 7 was reset.
These key codes can be ideal for many applications. It is particularly useful that the lower four bits are identical for both keyboards, and the fifth bit indicates which keyboard was used. However, in some cases an alternative

Table 16

0FD0	20	EORZ, R0		
0FD1	CC089F	STRA, R0	presets for	
0FD4	7712	PPSL, RS, COM	keyscan	
0FD6	7509	CPSL, WC, C		
0FD8	0CIFCB	LODA, R0		
0FDB	F440	TMI, R0	wait for VRLE	
0FDD	9879	BCFR		
0FDF	3F0183	BSTA, UN	Gosub 'Keyscan	
0FE2	F540	TMI, R1	repeat if	keyboard scan
0FE4	1872	BCTR	1st scan	and decode
0FE6	01	LODZ, R1		
0FE7	1A05	BCTR		
0FE9	0430	LODI, R0	load '30'	
0FEB	7510	CPSL, RS	if no key	
0FED	17	RETC, UN	else load	
0FEE	451F	ANDI, R1	translated code	
0FF0	0D6122	LODA, I/R1	and return	
0FF3	7510	CPSL, RS		
0FF5	17	RETC, UN		
0FF6	3B58	BSTR, UN	(keyscan	wait for key
0FF8	F430	TMI, R0	(no key)	release, then
0FFA	987A	BCFR		return
0FFC	17	RETC, UN	return	

Registers used: R0, R1', R2', R3';
Subroutine levels used: 2 for 'keyboard scan',
3 for 'wait for key release'.

code is more suitable, and this is obtained by the second part of the routine (from address 0FE6 to 0FF5). The translated key codes shown in figure 29b will be transferred into Register 0.

This code has several advantages. For the sixteen 'number keys', the data simply corresponds to the key number. All other keys are distinguished by the fact that bit 7 is logic 1; furthermore, bit 6 is logic 1 for the '+' and '−' keys only. Similarly, bit 5 uniquely identifies the RCAS and WCAS keys. The only disadvantage is that the upper control (UC), lower control (LC) and reset keys (the latter only if the key is wired as part of the keyboard) are all translated as 80, since they are not used in the monitor routines.

Finally, an additional subroutine using the keyboard scan routine is included from address 0FF6 on: 'Wait for key release'. This routine simply repeats the keyboard scan until the indication '30', for 'no key', is obtained.

Some little routines

After the extensive discussion of the keyboard scan routines, it should come as a welcome relief to take a look at some little subroutines.

clear duplicates

The instruction '3F009E' (BSTA, UN, 009E) causes 'FE' to be loaded into the four 'vertical offset duplicate' addresses: 1F0D, 1F1D, 1F2D and 1F4D. The result is that only the basic objects will appear on the screen, without any duplicates.

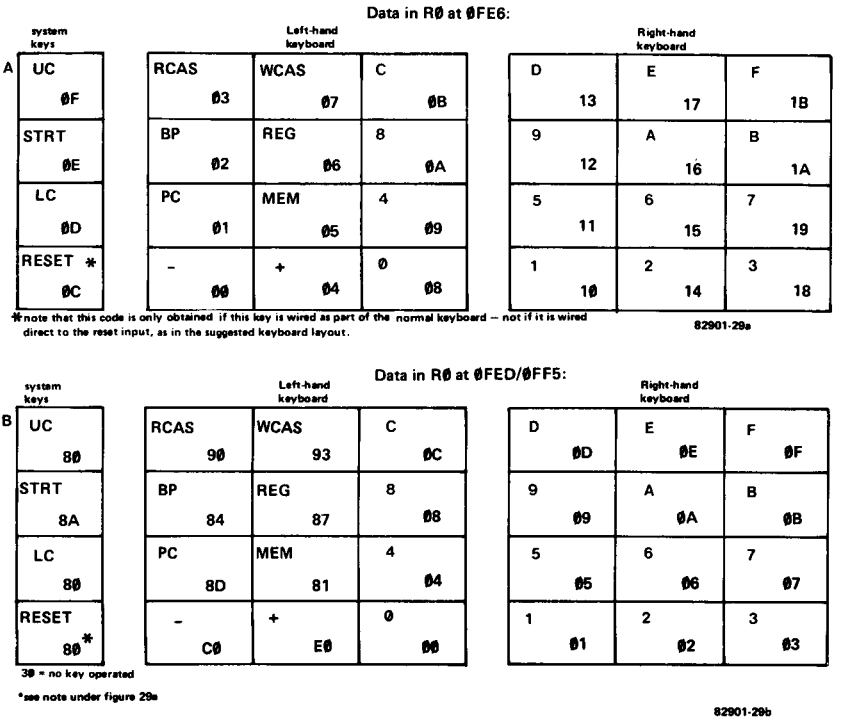


Figure 29. The upper half of this figure shows the data obtained for each key, after the basic keyboard scan; the lower half shows the 'translated' key codes.

Alternatively, any other desired vertical offset can be loaded by first storing it in RØ and then starting the subroutine at address ØØAØ.
Only register zero is used in this routine.

clear objects
All object shape data can be cleared by storing ØØ at all addresses from 1FØØ to 1F4F. This is accomplished by a subroutine starting at address Ø16E. Any other data present in RØ (FF, say) can be loaded into all these addresses by starting the subroutine at address Ø16F.
Registers used: RØ and R2.

split register
The 8 bits in a register can be written as two hexadecimal characters. Sometimes it is useful to actually separate these two characters. A subroutine, starting at address Ø35E, splits the data in R1. If the original data in this register was 'XY', the subroutine will leave 'ØY' in R1 and load 'ØX' into RØ.

Text display routines

There are, of course, several other small subroutines available in the monitor software. However, most of these are closely related to the text display routines, and so it is easier to treat them as a separate group.

initiate PVI

This subroutine (starting at address Ø161) presets the PVI for text display. It has the following effects:

- objects size 2 ('AA' in 1FCØ);
- correct colour (yellow objects, blue screen),
- 'ØØ' in 1FC3 (form/pos);
- sound off;
- disable score ('AA' in 1FC8 and 1FC9);
- clear objects ('ØØ' in 1FØØ ... 1F4F).

Note that all object position data is set to ØØ by this routine! Furthermore, the background data is *not* cleared; the background is merely made 'invisible' by giving it the same colour as the screen.
Registers used: RØ, R1, R2.

message data

When writing a text on the screen, a lot of complicated data must obviously be loaded into the 'object shape' area in the PVI. Fortunately, several characters are pre-programmed in the monitor software, as listed in Table 17. The first 28 (up to and including the 'x' sign) are deliberately programmed; the rest are 'accidental'. A complete scan of all characters

Table 17

character	code	character	code	character	code	character	code
0	ØØ	A	ØA	P	14	?	5F
1	Ø1	b	ØB	r	15	. .	8A
2	Ø2	C	ØC	=	16	n (1)	AA
3	Ø3	d	ØD	space	17	I	BB
4	Ø4	E	ØE	+	18	T	BC
5	Ø5	F	ØF	—	19	I	DF
6	Ø6	G	1Ø	:	1A	: (2)	E6
7	Ø7	L	11	x	1B	•	F7
8	Ø8	I	12			I (3)	A2
9	Ø9	n	13				

Notes:
(1) this n is slightly larger than the 'official' version (code 13), and looks better between capitals.
(2) similarly, this colon is larger than that obtained by code 1A, which can be useful.
(3) the exclamation mark is too small, actually, but no better version exists . . .
(4) the Ø (code ØØ) can be used as the letter O; similarly, a 5 makes a good S and a 2 will pass for a Z.

and other shapes that can be obtained in this way is included as one of the routines in File C on the ESS 007 cassette.
To obtain one line of text on the screen, the codes derived from Table 17 must be loaded into addresses Ø89Ø ... Ø897: eight characters in all for each line. If spaces are required, the code '17' must be stored in the corresponding addresses. In some cases, it may be useful to first store 8 spaces and then store the one or two characters required. There is a subroutine for this, starting at address Ø2D9; it uses RØ and R2.
A program example may serve to clarify the points discussed so far. The routine given in Table 18 (derived from Table 11 in chapter 9) will produce a complete display of the most useful characters.

After the usual 'interrupt inhibit' instruction, the first step is to initiate the PVI, as described above: '3FØ161'. Then R3 and R1 are preset, for the total number of characters (42 = 2A) and the number of characters per line (Ø7) respectively; the desired character codes are stored from address Ø93Ø on.
The 'load 8 spaces' routine is included as the next step ('3FØ2D9'). Not that it is strictly necessary in this case (we're already loading seven characters in each line, and one more space could easily be added), but it serves to illustrate the principle. The following small loop (from Ø90C to Ø912) transfers the first line of code numbers (from address Ø953 on) to the 'message line scratch' (from address Ø89Ø on).
We now come to the next monitor subroutine:

load Mline
This monitor subroutine (at address Ø20E) translates the codes stored in the message line scratch to the corresponding shape data for the four objects, and stores the results in a 'display scratch' (from address Ø8ØØ to Ø88F, for all six lines!).
Since this routine uses all four active registers (RØ ... R3), it would alter the character count data in R3. One solution would be to use the Load Immediate instruction at address Ø9Ø7 as scratch, as described earlier. In this program, an alternative solution is used: the upper register bank is selected before branching to the subroutine.
The next step is to check whether all characters, for all six lines, have been loaded. As long as this is not the case the program branches to address Ø927, bringing us to the next subroutine:

scroll
To be more precise, this subroutine (from address Ø2CF) should be listed as 'scroll and load 8 spaces in Mline'. It has the following effects:
- all object display data in the display scratch is moved up one line, from sixth to fifth, from fifth to fourth, and so on; the data for the first line is lost;
- the code for 'space' (17) is loaded in the eight message line scratch positions.

Since this routine uses registers Ø, 1 and 2, it is again padded by register-bank-select instructions. Unnecessary, in this case, since the only register

Table 18

```
0900        7620        PPSU, II
0902        3F0161      BSTA, UN      (clear/initiate PVI)
0905        072A        LODI, R3
0907    ┌──►0507        LODI, R1
0909    │   3F02D9      BSTA, UN      (load 8 spaces)
090C    │  ┌►0F4930     LODA, I-R3    (messline data)
090F    │  │ CD4890     STRA, I-R1
0912    │  └─5978       BRNR, R1
0914    │   7710        PPSL, RS
0916    │   3F020E      BSTA, UN      (load Mline)
0919    │   7510        CPSL, RS
091B    │ ┌─5B0A        BRNR, R3
091D    │ │►0C1E89      LODA, R0   ┐
0920    │ │ F410        TMI, R0    │  wait for '+' key release
0922    │ │ └1879       BCTR       ┘
0924    │ │   1F0038    BCTA, UN      return to monitor
0927    └─┼►7710        PPSL, RS
0929      │ 3F02CF      BSTA, UN      (scroll)
092C      │ 7510        CPSL, RS
092E      └─1B57        BCTR, UN

0930        5F A2 17 8A 17 E6 F7 sixth line   ┐
0937        02 16 17 18 19 1A 1B fifth line   │
093E        AA 13 00 14 15 05 BC fourth line  │  DATA
0945        0E 0F 10 12 DF 11 BB third line   │
094C        07 08 09 0A 0B 0C 0D second line  │
0953        00 01 02 03 04 05 06 first line   ┘
```

Start address: 0900

Table 19

- change the instruction at address 0924 to '1F095A' (instead of 1F0038);
- add the following section of program:

```
095A    ┌►►0C1FCB     LODA, R0   ┐
095D    │  F440       TMI, R0    │  wait for VRLE
095F    │  └9879      BCFR       ┘
0961    │   0C1E88    LODA, R0   ┐
0964    │   F420      TMI, R0    │  return to monitor if 'PC'
0966    │   1C0000    BCTA       ┘
0969    │   7702      PPSL, COM  ┐
096B    │   3F0055    BCTA, UN   │  display 6 lines
096E    └──1B6A       BCTR, UN   ┘
```

data that must be preserved is that in R3 – but once again included to illustrate the principle.
After this routine, the program branches back to address Ø9Ø7, to load the next line.

Once all six lines have been loaded, the branch instruction at address Ø91B will not be executed: the data in R3 are now zero. An uncommon program ending follows:
- wait for '+' key release – the program is started by operating this key, and the microprocessor is so fast that it will have finished the program before you have time to release the key!
- return to monitor at address ØØ38. This transfers control back to the monitor program in such a way that it takes care of putting the text on the screen, without first writing any message of its own!

In most cases, however, this easy way out will not be possible. A further monitor subroutine is then required to get the message on the screen:

display six lines
The six lines on the screen each consist of all four objects; lines 2 ... 6 are actually the duplicates, of course. To get the desired text on the screen the object shape data for each line must be retrieved from the display scratch at the correct moment, and stored in the object shape areas in the PVI.
The monitor subroutine that does this starts at address ØØ55; it uses registers RØ, R1 and R2. To obtain a correct display; the 'COM' bit in the PSL must be set (instruction: 77Ø2 = PPSL, COM). Furthermore, control must be transferred to this routine at the end of each frame; the return from subroutine will not occur before the sixth line has been displayed. This means that all further program checks or other routines can only be executed just before or during the 'frame end'.
As an illustration, the program given in Table 18 can be modified according to Table 19. All text display routines are now incorporated in the program.

Other routines

The monitor program contains a few more routines that can prove useful on rare occasions. However, they haven't got the 'general-purpose' appeal of the ones described above. For this reason, we will postpone discussing them until we describe the monitor program in greater detail, in a later chapter.

12

Interrupt facility

So far, our advice regarding the interrupt facility could be summed up in three words: Don't use it. However, this is a rather unsatisfactory state of affairs. The time has come to remedy this!

The PVI generates interrupt requests each time an object (or duplicate) is completed, and at the end of each frame. As long as the Interrupt Inhibit bit in the Program Status Upper is not set, all of these interrupt requests will be acknowledged. No matter what caused the interrupt (object 1 complete? duplicate 3 complete? end of frame? or whatever...), the results will be the same: the interrupt inhibit bit is set by the processor, the running program is interrupted, and the program section starting at address Ø9Ø3 is run as a subroutine.

If we assume that only the end-of-frame interrupt is of interest in a program, all others must be ignored. This is not too difficult: the 'sense' bit in the PSU is logic 1 at the end of the frame, so the interrupt subroutine at address Ø9Ø3 can be started as follows:

```
Ø9Ø3    B48Ø      TPSU, sense
Ø9Ø5    36        RETE
```

If the sense bit is not set, the TPSU instruction will result in the condition code 1Ø. The 'return and enable interrupt' instruction (RETE) is then executed, terminating the interrupt subroutine! Only if the sense bit proves to be logic 1, at the end of the frame, will the following interrupt routine be executed. Usually, that is, because there is one minor problem – but we'll come to that in a minute.

A more extensive interrupt select procedure is also possible. In the 'space shoot-out' program on ESS 007, for instance, the program actually starts as follows:

```
Ø9ØØ    1FØ9ØB    BCTA, UN     (to main program)
Ø9Ø3    B48Ø      TPSU, sense
Ø9Ø5    1CØA1Ø    BCTA         (to vertical interrupt routine)
Ø9Ø8    1FØ9D5    BCTA, UN     (to object interrupt routine)
Ø9ØB    762Ø      PPSU, II     (main program starts here)
```

In this case, if the sense bit is set the conditional branch at address Ø9Ø5 will be executed, starting the end-of-frame interrupt routine. Otherwise, this branch instruction will be ignored and the following (unconditional) branch will start the object-complete interrupt routine. The latter starts with a further check routine:

```
Ø9D5    ØC1FCA    LODA, RØ  ⎫ object 3
Ø9D8    F4Ø2      TMI, RØ   ⎭ complete?
Ø9DA    36        RETE         return if not
```

The final result is that only two basic interrupt requests will be acknowledged: frame-end and object 3 (or duplicate 3) complete. All other object or duplicate complete interrupts will be ignored.
When testing this program, one problem was found: Sometimes, the frame-end routine was missed. This error was traced to the fact that an 'object 3 complete' interrupt just before the frame end initiates the corresponding routine – and the latter 'over-runs' the frame end, so that no vertical interrupt was found! The solution, in this case, was simple: make sure that no 'object 3 complete' interrupts can occur just before the end of the frame, by selecting a suitable sequence of 'vertical offset duplicate' values.

Interrupt enable

A closer look at the program section given above (addresses Ø9ØØ to Ø9ØB) will lead to a surprise: the main program starts (at address Ø9ØB) by setting the interrupt inhibit bit! This means that no interrupt requests will be acknowledged – so what's the point of including interrupt routines?
Obviously, at some point in the program the interrupt inhibit bit must be reset. It is, *after* storing all kinds of initial data in the PVI and presetting a whole series of 'scratch' bytes in the program. Then, at address Ø9D1 to be precise, the following two instructions are inserted:

```
Ø9D1  ┌─7420    CPSU, II  ┐ wait for
Ø9D3  └─1B7C    BCTR, UN  ┘ interrupts
```

The processor will go round and round this loop, until an interrupt occurs. The interrupt routine will then be executed (again setting the interrupt inhibit bit, automatically); at the end of the interrupt routine, a 'return' instruction will cause the processor to jump back into this 'wait' loop. Note that the interrupt inhibit bit is reset in the loop, so that it is unimportant whether a 'normal' return instruction (17, say) or a return-and-enable-interrupt instruction is used.

As an illustration of the use of interrupts, a program is given in Table 20. Not that the same results couldn't have been obtained without using this facility! The data given from address Ø961 on corresponds to a series of sixteen words, one for each of the 'number' keys. If other words are required, the data can be derived from table 17. Note that each word must consist of 8 letters or less; if less than 8 letters are used, the remaining positions on each line must be filled with spaces (code 17).

Sharing the workload

In both program examples given so far, the main program peters out into a loop that does nothing more than clear the interrupt inhibit bit and wait for interrupts. This is rather wasteful – it is more logical to share the workload between interrupt routines and main program. One obvious solution is to run frame-related routines (object display and movement, timing, etc.) as interrupt routines, and include score updating, keyscan and so on in the main program.

Table 20

```
0900    1F0958   BCTA, UN
0903    B480     TPSU, sense      vertical interrupts only
0905    16       RETC
0906    B440     TPSU, flag
0908    1808     BCTR
090A    7640     PPSU, flag       set/reset flag on
090C    20       EORZ, R0         alternate frames;
090D    CC089F   STRA, R0         keyboard scan
0910    1B02     BCTR, UN         routine
0912    7440     CPSU, flag
0914    3F0181   BSTA, UN
0917    9A38     BCFR             (no key)
0919    01       LODZ, R1
091A    451F     ANDI, R1         translate key code
091C    0D6122   LODA, I/R1
091F    E4E0     COMI, R0         branch if '+' key
0921    182E     BCTR
0923    F480     TMI, R0          return to monitor
0925    1C0000   BCTA             if control key
0928    C804     STRR, R0         save data in R0
092A    3F02CF   BCTA, UN         and scroll
092D    0400     LODI, R0
092F    D0       RRL, R0
0930    D0       RRL, R0          R0 x 8
0931    D0       RRL, R0
0932    0608     LODI, R2
0934    82       ADDZ, R2
0935    C1       STRZ, R1
0936    0D4961   LODA, I-R1       load Mline
0939    CE4890   STRA, I-R2
093C    5A78     BRNR, R2
093E    3F020E   BSTA, UN
0941    0C1E8A   LODA, R0
0944    6C1E8C   IORA, R0
0947    6C1E8D   IORA, R0         wait for key release
094A    6C1E8E   IORA, R0
094D    44F0     ANDI, R0
094F    9870     BCFR
0951    3F0055   BSTA, UN         display 6 lines
0954    7420     CPSU, II         wait for interrupts
0956    1B7C     BCTR, UN

0958    7620     PPSU, II         clear/initiate PVI
095A    3F0161   BSTA, UN         and set COM bit
095D    7702     PPSL, COM
095F    1B73     BCTR, UN

0961    05 BC 0A 15 BC 17 17 17 data 0
0969    0B 0E 10 12 AA 17 17 17 data 1
0971    0A AA 0F 0A AA 10 17 17 data 2
0979    0D 0E 0B 56 BC 17 17 17 data 3
0981    0E AA 0D 17 17 17 17 17 data 4
0989    0E 12 AA 0D 0E 17 17 17 data 5
0991    0E AA 0D 0E 17 17 17 17 data 6
0999    0F 12 AA 17 17 17 17 17 data 7
09A1    0F 56 AA 17 17 17 17 17 data 8
09A9    11 00 11 17 17 17 17 17 data 9
09B1    05 14 0A 05 05 17 17 17 data A
09B9    15 12 10 00 11 0A 0D 0E data B
09C1    AA 12 0C 0E 17 17 17 17 data C
09C9    0A 0A 15 0D 12 10 17 17 data D
09D1    AA 0E BC BC 17 17 17 17 data E
09D9    10 0E AA BC 12 11 0E 17 data F
```

Table 21

```
0900    1Fxxxx    Jump to main program
0903    C808      STRR,R0  ) Preserve R0 and
   5    13        SPSL     ) PSL, select upper
   6    C809      STRR,R0  ) register bank.
   8    7710      PPSL, RS )
   A  ┌─3B07      BSTR, UN
   C  │ 0400      LODI, R0 ) Restore R0 and
   E  │ 75FF      CPSL     ) PSL – the latter
0910  │ 7700      PPSL     ) selects the lower
   2  │ 37        RETE, UN ) register bank.
0913  └─►....     Interrupt routine starts here. It can be
                  terminated at any point with 14, 15, 16
                  or 17 (RETC).
```

Note: this system requires a minimum of two subroutine levels for the interrupt routine.

The main program typically starts as follows:

```
        7620    PPSU, II
        0402    LODI,R0  ) select lower
        93      LPSU     ) register bank
        ....             ) initialisation
        ....             ) procedure
        7420    CPSU,II
        ....             ) start of main
        ....             ) program loop
```

However, to run both main program and interrupts, it is essential that registers and condition codes do not 'clash'. The problem is that a 'jump to interrupt' can occur at any time: between testing a value, say, and branching according to the result. To avoid problems, the 'Program Status Lower' must be saved when an interrupt routine is initiated; furthermore, the values in the registers must be protected. A simple solution for the latter problem is to use the lower register bank for the main program, and the upper bank for the interrupt routines. This way, only register 0 is common to both; this register must be protected. A suitable routine, catering for all problems, is given in Table 21.

If all registers – both the lower and upper banks – must be saved, things become rather more complicated – as shown in Table 22! It is possible to shorten this routine drastically, by making extensive use of monitor routines. However, this does require a fuller understanding of the monitor software and so we will postpone discussing it for the time being (see chapter 16).

Table 22

0900	1Fxxxx	Jump to main program
0903	C82C	STRR, R0
5	13	SPSL
6	C82D	STRR, R0
8	12	SPSU
9	C823	STRR, R0
B	7510	CPSL, RS
D	C911	STRR' R1
F	CA11	STRR, R2
0911	CB11	STRR, R3
3	7710	PPSL, RS
5	C911	STRR, R1
7	CA11	STRR, R2
9	CB11	STRR, R3
B	3B1A	BSTR, UN
D	7510	CPSL, RS
F	0500	LODI, R1
0921	0600	LODI, R2
3	0700	LODI, R3
5	7710	PPSL, RS
7	0500	LODI, R1
9	0600	LODI, R2
B	0700	LODI, R3
D	0400	LODI, R0
F	92	LPSU
0930	0400	LODI, R0
2	75FF	CPSL
4	7700	PPSL
6	37	RETE, UN
0937	....	Interrupt routine starts here.

13

Joysticks

So far, almost the only mention we have made of the joysticks was in the second chapter: how to connect them! In all practical examples given so far, we have used keys to move the objects. Why?
The main reason is that joysticks tend to vary widely from one TV games computer to another. If everyone used the specified 680k potentiometers, it wouldn't be so bad; in practice, unfortunately, this is not the case. The only solution is to use a 'joystick calibration routine', as will be described in this chapter.

The basic principle of joystick operation is fairly straightforward:
Two addresses in the PVI, 1FCC and 1FCD, correspond to the left-hand and right-hand joysticks, respectively. When the flag is set, the vertical position of each joystick is scanned and the results are stored at the corresponding address; if the flag is not set, the horizontal setting is scanned. The data in the two PVI addresses is only valid at the end of the frame – when the sense bit is at logic 1, in other words.
A low data value in address 1FCC or 1FCD corresponds to 'up' or 'left', depending on the setting of the flag during the previous frame (when the actual A-D conversion took place).
The actual range of values obtained varies from one joystick to another. Unfortunately! This means that it is not easy to write a program that is suitable in all cases. However, as a first step it is worthwhile to key in the program given in Table 23. It reads the data in the two PVI addresses, with the flag both 'on' and 'off', and displays the results on the screen as follows:

FLAG ON	(= vertical)
1FCC 75	(= left)
1FCD AD	(= right)
FLAG OFF	(= horizontal)
1FCC 11	(= left)
1FCD 83	(= right)

The data found at the two addresses is updated on the screen as required. The values given above (75, AD, 11, 83) are just examples, without any special meaning.
If the joysticks are wired as shown in chapter 2, address 1FCC should correspond to the left-hand joystick; 'Flag on' should correspond to vertical movement; and low data values should be obtained at the extreme 'up' and 'left' positions. Now is the time to check – by means of the program given in Table 23 – and resolder the connections to your joysticks if necessary!

Table 23

0900	1F0990	BCTA, UN	
0903	B480	TPSU, sense	vertical interrupts only
0905	16	RETC	
0906	B440	TPSU, flag	
0908	1804	BCTR	flag on
090A	7640	PPSU, flag	alternate frames
090C	1B02	BCTR, UN	
090E	7440	CPSU, flag	
0910	0D1FCC	LODA, R1	
0913	0E1FCD	LODA, R2	save joystick
0916	C90B	STRR, R1	data
0918	CE095C	STRA, R1	
091B	3F0055	BSTA, UN	display 6 lines
091E	0702	LODI, R3	
0920	0602	LODI, R2	
0922	0500	LODI, R1	joystick data! (IFCC)
0924	B440	TPSU, flag	
0926	1802	BCTR	
0928	0604	LODI, R2	
092A	0418	LODI, R0	
092C	CC096D	STRA, R0	
092F	04E0	LODI, R0	
0931	CC0984	STRA, R0	presets for
0934	04CD	LODI, R0	subroutine
0936	CC0985	STRA, R0	
0939	0E4963	LODA, I-R2	
093C	CC0987	STRA, R0	
093F	CC098A	STRA, R0	
0942	3F035E	BSTA, UN	split register
0945	3F0967	BSTA, UN	
0948	0498	LODI, R0	
094A	CC096D	STRA, R0	presets for
094D	040E	LODI, R0	subroutine
094F	CC0984	STRA, R0	
0952	046D	LODI, R0	
0954	CC0985	STRA, R0	
0957	01	LODZ, R1	
0958	3F0967	BSTA, UN	
095B	0500	LODI, R1	joystick data! (IFCD)
095D	FB4B	BDRR, R3	
095F	7420	CPSU, II	wait for interrupts
0961	1B7C	BCTR, UN	
0963	89 71 41 29		address data
0967	7710	PPSL, RS	
0969	0700	LODI, R3	
096B	F401	TMI, R0	preset R3
096D	1802/9802	BCTR/BCFR	
096F	0701	LODI, R3	
0971	440E	ANDI, R0	
0973	C2	STRZ, R2	
0974	D2	RRL, R2	3 x R0
0975	82	ADDZ, R2	

(continued on next page! →

```
0976    0506      LODI, R1
0978    81        ADDZ, R1      set R1, R2
0979    C2        STRZ, R2
097A  ->0E427B    LODA, I-R2
097D    5B04      BRNR, R3
097F    D0        RRL, R0
0980    D0        RRL, R0
0981    D0        RRL, R0
0982    D0        RRL, R0
0983  ->44E0/440E ANDI, R0
0985    CD6829/   STRA/IORA, I/R1
        6D6829
0988    CD6829    STRA, I/R1
098B    F96D      BDRR, R1
098D    7510      CPSL, R5
098F    17        RETC, UN
0990    7620      PPSU, II
0992    3F0161    BSTA, UN      (clear/initiate PVI)
0995    04CC      LODI, R0      address preset
0997    C80F      STRR, R0
0999    0702      LODI, R3
099B  ->0610      LODI, R2
099D    0508      LODI, R1
099F  ->7710      PPSL, RS
09A1    3F02CF    BSTA, UN      scroll
09A4    7510      CPSL, RS
09A6  ->0E49CC    LODA, I-R2
09A9    CD4890    STRA, I-R1    Messline data
09AC    5978      BRNR, R1
09AE    04C4      LODI, R0      address preset
09B0    C876      STRR, R0
09B2    7710      PPSL, RS      load Mline
09B4    3F020E    BSTA, UN
09B7    7510      CPSL, RS
09B9    0504      LODI, R1
09BB    5A62      BRNR, R2
09BD    FB5C      BDRR, R3
09BF    7702      PPSL, COM
09C1    1F095F    BCTA, UN

09C4    01 0F 0C 0D                          basic message data
09C8    01 0F 0C 0C
09CC    0F 11 0A 10 17 00 0F 0F
09D4    0F 11 0A 10 17 00 AA 17
```

Note:
at addresses 096D, 0983 and 0985 either of the alternatives given can be entered. The program modifies these instructions as required!
Start address: 0900.

This particular routine was tried on a large number of different joysticks, with widely varied results... The minimum values found vary between 05 and 28; the maxima were anywhere between 25 and FA. The midrange could be anything between 15 and 7E. Help! What do you do when one

person's minimum is more than someone elses maximum? The only result that was consistent (not surprisingly) was the value obtained without any joystick connected: ØD in all cases.

Against all odds, we think we have a solution that should satisfy everyone. It is based on two conclusions from the results given above:

- If joysticks are to be used, (automatic) calibration is essential.
- Wherever possible, the joysticks are best used as four-way switches (signalling 'up', 'down', 'left' or 'right').

Trying to obtain data that corresponds to all possible positions is virtually doomed to failure, insofar as it is to be compatible with other computers. For strictly 'personal' programs it is no problem, of course.

Now, we come to our 'solution' – or solutions, rather. An automatic

Table 24

```
Ø8CØ  ┌►ØC1FCB   LODA,RØ  }
   3  │  DØ       RRL, RØ  } subroutine
   4  └─9A7A      BCFR     } wait for VRLE
   6     17       RETC,UN  }
Ø8C7     7660    PPSU, II + flag
   9     7518    CPSL, RS + WC
   B     Ø7FF    LODI, R3
   D     3B71    BSTR, UN
   F     3B6F    BSTR, UN
Ø8D1  ┌►3B6D    BSTR, UN
   3  │  Ø6Ø2    LODI, R2
   5  │►ØE5FCC  LODA,I-R2
   8  │  C1      STRZ, R1
   9  │  51      RRR, R1
   A  │  51      RRR, R1
   B  │  453F    ANDI, R1
   D  │  A1      SUBZ, R1
   E  │  CF28EF  STRA, I + R3
Ø8E1  │  81      ADDZ, R1
   2  │  81      ADDZ, R1
   3  │  CF28EF  STRA, I + R3
   6  └─5A6D    BRNR, R2
   8     B44Ø    TPSU, flag
   A     16      RETC
   B     744Ø    CPSU, flag
   D     1B62    BCTR, UN

Ø8EF    ØØ   ØØ   ØØ   ØØ    vertical data
        up   down up   down
        Right     Left

Ø8F3    ØØ   ØØ   ØØ   ØØ    horizontal data
        up   down up   down
        Right     Left
```

Note: this joystick calibration routine can be located at any other point in memory, provided the data at addresses Ø8DE and Ø8E3 are modified to correspond to the desired 'switchpoint data' locations.

joystick calibration routine, with variations depending on the intended application. The basic routine is given in Table 24. As given here, the routine starts at address Ø8C7; it is included in the initialisation procedure as a subroutine: 3FØ8C7. This means that it is run only once, at the first start of the program. The joysticks are assumed to be in their mid positions at that time, and switching points relative to these positions are calculated and stored from address Ø8EF on.
Having calibrated the joysticks, a joystick scan can be incorporated at any point in the main program. The values found at that time are compared to the reference values (from address Ø8EF on) to determine the joystick position. For correct operation, a joystick scan must occur at frame end – after a 'wait for VRLE' loop, for instance. The subroutine at Ø8CØ can be used for this. Furthermore, to scan both horizontal and vertical joystick positions the flag must be set and reset on alternate frames. It is normally preferable to deal with the flag *first* and then check the joystick data – bearing in mind that it refers to the flag status during the *preceding* frame! Furthermore, it is good procedure to grab all relevant joystick information (both 1FCC and 1FCD, if both are required) as soon as possible, and then evaluate them at your leisure. A fairly general-purpose scan routine is given in Table 25. Obviously, it can be modified according to the demands of any particular program.

To illustrate how these routines can be used, a simple demonstration program is given in Table 26. After loading both Tables (24 and 26), the main program is started at ØØØØ. The first two objects, corresponding to 'PC' and '=' respectively, will jump to the centre of the screen ('ØØØØ' remains at the top right). The positions of these two objects can now be

Table 25

```
3FØ8CØ   BSTA, UN   Wait for VRLE
12       SPSU       } set/reset flag on
2440     EORI, RØ   } alternate frames
92       LPSU       }
ØE1FCC   LODA,R2    left-hand joystick data
ØF1FCD   LODA,R3    right-hand joystick data
7702     PPSL, COM
B44Ø     TPSU, flag } branch for horizontal routines
18..     BCTR       } if flag now set
EEØ8F1   COMA,R2
9A..     BCFR
...                 left-hand joystick up routine
EE08F2   COMA,R2
99..     BCFR
...                 left-hand joystick down routine
EFØ8EF   COMA,R3
9A..     BCFR
...                 right-hand joystick up routine
etc.
```

Table 26

```
0900   3F08C7   BSTA, UN          calibration routine
   3   0444     LODI, R0
   5   CC1F0A   STRA, R0
   8   0455     LODI, R0
   A   CC1F1A   STRA, R0          initial
   D   0477     LODI, R0          positions
   F   CC1F0C   STRA, R0
0912   CC1F1C   STRA, R0
   5   3F08C0   BSTA, UN          wait for VRLE
   8   12       SPSU
   9   2440     EORI, R0          set/reset flag
   B   92       LPSU              on alternate frames
   C   0E1FCC   LODA,R2           left-hand joystick data
   F   0F1FCD   LODA,R3           right-hand joystick data
0922   7702     PPSL, COM
   4   B440     TPSU, flag        branch for horizontal
   6   -1829    BCTR              routines if flag now set
   8   0C1F0C   LODA,R0           vertical position object 1
   B   0D1F1C   LODA,R1           vertical position object 2
   E   EE08F1   COMA,R2
0931   -9A02    BCFR
   3   A401     SUBI, R0
   5   →EE08F2  COMA,R2
   8   -9902    BCFR
   A   8401     ADDI, R0
   C   →EF08EF  COMA,R3
   F   -9A02    BCFR              vertical movement
0941   A501     SUBI, R1
   3   →EF08F0  COMA,R3
   6   -9902    BCFR
   8   8501     ADDI, R1
   A   →C8DD    STRR, R0 Ind (1F0C)
   C   C9DE     STRR, R1 Ind (1F1C)
   E   1F0915   BCTA, UN
0951   →0C1F0A  LODA, R0           horizontal position object 1
   4   0D1F1A   LODA, R1           horizontal position object 2
   7   EE08F5   COMA,R2
   A   -9A02    BCFR
   C   A401     SUBI, R0
   E   →EE08F6  COMA,R2
0961   -9902    BCFR
   3   8401     ADDI, R0
   5   →EF08F3  COMA,R3
   8   -9A02    BCFR              horizontal movement
   A   A501     SUBI, R1
   C   →EF08F4  COMA,R3
   F   -9902    BCFR
0971   8501     ADDI, R1
   3   →C8DD    STRR, R0 Ind (1F0A)
   5   C9DE     STRR, R1 Ind (1F1A)
   7   1F0915   BCTA, UN
```

controlled by means of the left-hand and right-hand joysticks.
A few points in both programs are worth mentioning – even though they do not relate to the joysticks as such:

- In Table 24, the 'wait for VRLE' loop uses a Rotate instruction (DØ) to set the condition code for 'negative' if the VRLE bit is set. This is shorter than the Test under Mask instruction 'F44Ø'.
- At the end of the joystick calibration procedure (in other words, when returning from address Ø8EA), the Interrupt Inhibit bit is set and the flag is reset. Furthermore, the lower register bank is selected and the With Carry bit is reset (no carry). There is no need to repeat these presets in the following section of the main program!
- Table 25 starts with the 'wait for VRLE' subroutine. Obviously, this is unnecessary if the routine is included in a 'Vertical interrupt' routine.
- The 'set/reset flag' routine clearly illustrates the value of the EXOR instruction!
- The COM bit is set to ensure 'logical' compare – otherwise, all joystick values greater than 8Ø would be misinterpreted as negative numbers.
- Both routines use relative addressing wherever possible – that is, in almost all cases except those requiring indexed addressing. This means that they can be relocated, elsewhere in memory, with a minimum of effort.

Earlier on, we mentioned the fact that the calibration routine would be given 'with variations', according to the intended application. In fact, two main variations of the routine given in Table 24 are included in the Appendix. The first is a simplified version, intended for use when only one joystick needs calibration – as for single-player games. The second is more fundamental: it stores the results of the calibration routine in the main program (again, it is given for calibration of one joystick only). The idea is to include the results as part of a 'Compare Immediate' instruction: E4xx, say, instead of EC Ø8EF. This can save a byte or two, in some cases...

Proportional control

In all routines discussed so far, we have used the joysticks to indicate 'left', 'right', 'up' or 'down'. We have not attempted to use them to define any particular position – in the sense that the joystick position corresponds to an object position. 'The further left the joystick, the further left the object' has not been tried; we have confined ourselves to 'joystick left? Object moves left' and so on. With good reason: it's much easier that way!
There are a few cases, however, where some kind of proportional control would be useful. This is especially true when rapid object movement must be possible (for moving a 'bat' in a ping-pong game, say). The same basic rule still applies: the joysticks must be calibrated, to cater for different TV Games computers! The question is: how?
A basic solution is given in Table 27. Without going into great detail, this works as follows:

- The calibration routine itself begins at address Ø8CØ. After some initial presets, the joystick value is loaded at address Ø8C6.
- The first step is to calculate a number of left or right shifts that converts the joystick value (assumed to be mid-position!) to the nearest possible value that is less than 6Ø – the mid-value for a horizontal co-ordinate. The number of shifts is stored at address Ø8EA.

Table 27

```
Ø8CØ    7702      PPSL, COM
   2    7518      CPSL, RS/WC
   4    2Ø        EORZ, RØ
   5    C2        STRZ, R2
   6    ØB9F      LODR, R3 Ind (1FCC)
   8    E76Ø      COMI, R3
   A    1903      BCTR
   C    D3        RRL,  R3
   D    D879      BIRR, RØ
   F    C819      STRR, RØ (Ø8EA)
Ø8D1    47FE      ANDI, R3
   3    53        RRR,  R3
   4    Ø3        LODZ, R3
   5    6680      IORI, R2
   7    D2        RRL,  R2
   8    CA1A      STRR, R2 (Ø8F4)
   A    16        RETC
   B    47FE      ANDI, R3
   D    53        RRR,  R3
   E    83        ADDZ, R3
   F    E460      COMI, RØ
Ø8E1    9972      BCFR
   3    A3        SUBZ, R3
   4    1B71      BCTR, UN

Ø8E6    ØC1FCC    LODA, RØ
   9    Ø5ØØ      LODI, R1
   B    44FE      ANDI, RØ
   D    1803      BCTR
   F    D0        RRL,  RØ
Ø8FØ    F97D      BDRR, R1
   2    50        RRR,  RØ
   3    Ø5ØØ      LODI, R1
   5    C2        STRZ, R2
   6    46FE      ANDI, R2
   8    14        RETC
   9    52        RRR,  R2
   A    D1        RRL,  R1
   B    9A79      BCFR
   D    82        ADDZ, R2
   E    1B76      BCTR, UN
```

Note: When calibrating joysticks for vertical operation (flag set), the instructions at addresses Ø8C8 and Ø8DF can be modified to E77F and E47F respectively. Larger values than 7F should not be used for these comparisons!

- Next, a further approximation to the desired mid-value is calculated. This consists of a succession of right shifts (halving the value) and adding; if the result does not exceed the desired mid-value, the corresponding operation is specified in R2. The final value in R2, corresponding to the 'best approximation procedure', is stored at address Ø8F4.
- When a joystick scan is required in the program, the subroutine at address Ø8E6 is run. This routine is now 'calibrated' to convert any joystick value into the desired value in the horizontal range.

As stated above, this is a 'basic solution'. In other words, it can be modified as required! As it stands, it calibrates the left-hand joystick; for the right-hand one, the instruction at address Ø8E6 must be modified to ØC1FCD. If both joysticks must be calibrated, there are two options. In the ideal case, two independent sets of preset values for the joystick scan subroutine should be calculated and stored. In practice, the two joysticks in one machine will normally be fairly similar, so that an evaluation routine that has been preset for, say, the left-hand joystick can also be used for the right-hand one. Provided, of course, that an accurate mid-position with spring-loaded joysticks is not required.
Another point concerns the difference between horizontal and vertical use. As given here, the routine is suitable for deriving a horizontal object

Table 28

Ø9ØØ	Ø42Ø	LODI, RØ	(Ø46Ø for vertical)
2	92	LPSU	
3	3B81	BSTR, UN, Ind (Ø93Ø)	
5	3FØ93Ø	BSTA, UN	
8	3FØ8CØ	BSTA, UN	Joystick calibration
B	Ø4AØ	LODI, RØ	
D	CC1FØC	STRA, RØ	
Ø91Ø	ØC1FCC	LODA,RØ	} Joystick scan and evaluation
3	C3	STRZ, R3	
4	3FØ8E9	BSTA, UN	
7	CC1FØA	STRA, RØ	(CC1FØC for vertical)
A	Ø3	LODZ,R3	} Display joystick value.
B	C1	STRZ, R1	
C	Ø7Ø2	LODI, R3	
E	3FØ354	BSTA, UN	
Ø921	3FØ2ØE	BSTA, UN	
4	Ø578	LODI, R1	
6	Ø6ØØ	LODI, R2	
8	3FØØ7A	BSTA, UN	
B	3FØ93Ø	BSTA, UN	
E	1B6Ø	BCTR, UN	
Ø93Ø	ØC1FCB	LODA,RØ	} Subroutine: wait for VRLE.
3	DØ	RRL, RØ	
4	9A7A	BCFR	
6	17		

position from the joystick value – for the simple reason that that was the first application we had for it... For vertical use, the comparison values at addresses Ø8C8 and Ø8DF can be increased to '7F'. Note that a higher value should not be used!
To illustrate the possibilities – and demonstrate how to include these routines in a program! – a simple program is given in Table 28. After loading both Tables (27 and 28), the program is started at Ø9ØØ. Initially, the Interrupt Inhibit bit is set and the flag is reset (for horizontal scan). Then, after two 'wait for VRLE' routines, the joystick is calibrated. In the main loop, from Ø91Ø on, the modified joystick value is used to determine the horizontal position of object 1. Furthermore, the actual joystick value is displayed by this object.
The same program can be used to demonstrate vertical movement. The only modifications are at addresses Ø8C8 and Ø8DF, as described above, and at Ø9ØØ and Ø917 as shown in Table 28.

Important note
Not all joysticks are spring-loaded – to move them into the centered position when they are not being operated. All calibration routines assume, however, that they are centered initially! This means that owners of 'sloppy' joysticks must ensure that they are centered *before* starting this type of program!

14

Random numbers

In many programs, random numbers play an important part. They can be used to set up the initial positions of objects; to move 'computer-controlled' objects; to start some routine ('fire missile', say) at random intervals; and so on. In this chapter, we will discuss a few suitable 'software' routines as well as a 'hardware' version that can be included in the TV games computer if desired.

When you get down to it, working out a program that gives random numbers is not as easy as you might think. There is one very simple solution, admittedly: a straightforward count. This can be written as a subroutine:

```
0400    LODI, R0
D800    BIRR, R0
C87B    STRR, R0
17      RETC, UN
```

The major disadvantage of this system is that it simply counts up from 00 to FF in sequence and then starts again. In practice, this means that when two 'random' numbers are required in rapid succession, the second is likely to be slightly greater than the first – it is certainly not an 'independent' random number. For this reason, the routine should be run as often as possible – for the duration of all 'wait' loops, for instance.

Table 29

```
..00    0C1FAD  LODA,R0
  3     9802    BCFR
  5     0401    LODI, R0
  7     C1      STRZ, R1
  8     C2      STRZ, R2
  9     0703    LODI, R3
  B     52      RRR, R2
  C     52      RRR, R2
  D     52      RRR, R2
  E     52      RRR, R2
  F     02      LODZ, R2
..10    4401    ANDI, R0
  2     21      EORZ, R1
  3     C1      STRZ, R1
  4     FB78    BDRR, R3
  6     51      RRR, R1
  7     C9E8    STRR, R1   Ind (1FAD)
  9     17      RETC, UN
```

Fortunately, theoreticians have worked out how to make a 'true' random numbers generator by means of EXOR operations, for almost any number of bits. In our case, we are interested in the 8-bit version and this requires three successive EXORs. A suitable routine is given in Table 29. The random number is located at address 1FAD (one of the PVI scratch addresses). At each pass through the routine, a new number – without any obvious relationship to the preceeding one! – is stored at this address. As before, the routine must be kept 'ticking over' by running it at regular intervals; once at each VRLE, for instance.

As it stands, this routine has one minor disadvantage: the random number can never be ØØ. In fact, the basic routine itself would 'stick' if the random number became ØØ for some reason: it would not change it from then on! For this reason, the first few instructions convert any undesirable ØØs into Ø1.

This disadvantage can be overcome by using the more extensive routine given in Table 30. The additional instructions after the main loop ensure that ØØ is converted into 8Ø, whereas Ø1 is converted into ØØ. In effect, this inserts ØØ in the random numbers sequence between Ø1 and 8Ø.

For a general impression of how these routines work, key in the following program:

```
ØØØØ    762Ø      PPSU, II
   2    3FØ8CØ    BSTA, UN
   5    CD1FØC    STRA, R1
   8    ØC1FCB    LODA, RØ
   B    DØ        RRL, RØ
   C    9A7A      BCFR
   E    1B72      BCTR, UN
```

Table 30

```
..ØØ    ØC1FAD   LODA,RØ
  3     C1       STRZ, R1
  4     C2       STRZ, R2
  5     Ø7Ø3     LODI, R3
  7     52       RRR,  R2
  8     52       RRR,  R2
  9     52       RRR,  R2
  A  ┌─►52       RRR,  R2
  B  │  Ø2       LODZ, R2
  C  │  44Ø1     ANDI, RØ
  E  │  21       EORZ, R1
  F  │  C1       STRZ, R1
..1Ø └──FB78     BDRR, R3
  2  ┌──98Ø2     BCFR
  4  │  Ø5Ø1     LODI, R1
  6  └─►E4Ø1     COMI, RØ
  8  ┌──98Ø2     BCFR
  A  │  Ø5ØØ     LODI, R1
  C  └─►51       RRR,  R1
  D     C9E2     STRR, R1  Ind (1FAD)
  F     17       RETC, UN
```

The desired random numbers routine is entered from Ø8CØ on. Note that, when trying the simple count routine given at the outset, the instruction at address Ø9Ø5 should be modified to CC1FØC – the random number is in RØ!

The hardware version

The routines given so far will generate a random number, and they cost no more than a few memory bytes. So what's the point in adding a p.c. board with half-a-dozen ICs to do the job?

The main disadvantage of the software approach is that the random numbers are generated in a fixed sequence, in synchronism with the main program. This means that when random numbers are required at *regular* intervals in the program, the sequence tends to become apparent. Fortunately, it is usually possible to arrange things so that the exact time at which a random number is 'grabbed' depends on some key operation. Since the player is never in precise synchronism with the program, this system ensures that the numbers are sufficiently random. Even so, there are those cases where an asynchronous generator could be useful.

The circuit is given in figure 30. It makes use of an 8-bit shift register (IC2). The outputs Q6 and Q7 are fed back to the input of the shift register via EXOR gates N4 and N3, so that the length of the generated cycle is that of a 7-bit random number generator: 127 steps. The clock pulses for the shift

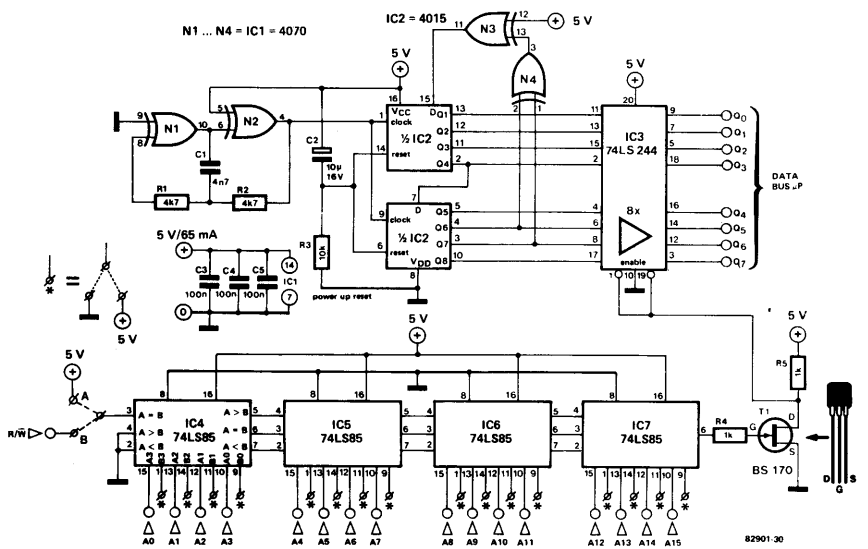


Figure 30. The circuit of the random numbers generator.

register are provided by an astable multivibrator consisting of N1, N2, R1, R2 and C1. The frequency of oscillation with the values of the components shown is approximately 20 kHz. Capacitor C3 and resistor R2 are included to reset the shift register when the unit is first switched on.
This is in fact the entire random number generator. Depending on the particular application, this section can be constructed separately and one or more outputs from the shift register can be used to obtain the random number.
However, in our case this generator must be linked to the rest of the system, and this is where the remainder of the circuit comes in. The outputs

Parts list

Resistors:
R1,R2 = 4k7
R3 = 10 k
R4,R5 = 1 k

Capacitors:
C1 = 4n7
C2 = 10 µ/16 V
C3,C4,C5 = 100 n

Semiconductors:
T1 = BS 170
IC1 = 4070
IC2 = 4015
IC3 = 74LS244
IC4,IC5,IC6,IC7 = 74LS85

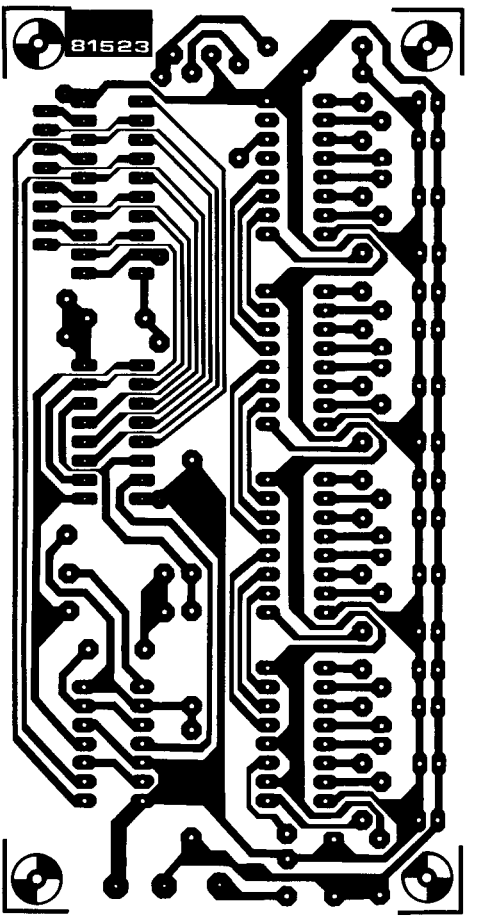


Figure 31. The p.c. board. Note that the A15 input and the corresponding 'B' input are both connected to supply common – there is no A15 in this system!

of the shift register (IC2) are connected to the inputs of the three-state buffer, IC3. The outputs of the buffer are, in turn, connected directly to the data bus of the microcomputer. Care should be taken to ensure that the outputs are in the high impedance condition as long as the processor does not require a random number. In order to detect this, an address decoder consisting of four 4 bit comparators has been included (IC4 ... IC7). The four comparators compare the 16 bit address offered by the microprocessor with the address of the random number generator. When the two are the same, the VMOS transistor, T1, turns on which enables the buffer, IC3, and permits the random number to be output on the data bus. A

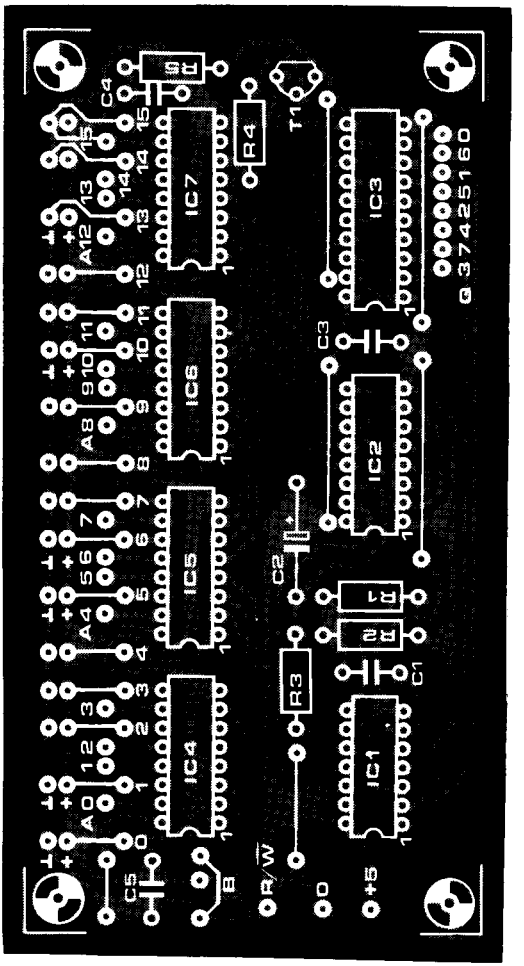


Figure 31.

VMOS transistor was selected for this purpose because of its necessarily short turn-on time.

The desired address for the random number generator can be set up on the printed circuit board (see figure 31) by soldering short wire links between the B inputs and the positive supply rail or ground. A connection between B and ground means a logic zero for that particular address line, and a link between B and the positive rail means logic one. The use of a positive read strobe signal requires a link between the R/$\overline{W}$ input and pin 3 of IC4; a suitable signal is available on pin 17 of the 31-pin connector.

The complete circuit requires a 5 V supply voltage and draws a mere 35 mA. This means that the random number generator can be powered from the existing microprocessor supply.
This circuit was deliberately designed as a versatile unit – in fact, it can be incorporated in virtually any microprocessor system. For this reason, several wire links on the actual p.c. board are shown with two options given as dotted lines. For use in the TV games computer, these links should be mounted as shown in figure 31. Among other things, this sets the 'random number address' to 1FFF – an unused address in the PVI area. Not that we have any programs that need this unit, as yet; but if and when they do materialise, it is highly desirable that everyone uses the same address...

15

Some useful routines

Developing programs the easy way!

Have you tried to develop your own program? And discovered, after the first trial run, that you had omitted a few essential steps somewhere? Welcome to the club! One solution is to laboriously key in the whole program again, with those instructions inserted where they belong. An easier way is to make use of one of the 'block transfer' routines described in this chapter.
That is only one example. Other routines like 'Clear RAM' and 'disassembler' can be equally useful. For that matter, given the basic idea you will soon be writing your own 'donkey-work' routines as you need them!

First things first: shifting program sections up or down in memory is one of the most common requirements. Admittedly, redundant instructions can be deleted by inserting 'NOP's (CØ), but that wastes program space. And additional steps can be added by replacing three of the original instruction bytes by an unconditional branch to an empty memory space, restoring the deleted instruction(s) there and adding the missing steps at that point before branching back. This system works, but it is anything but elegant.
The only alternative is to move up the remainder of the program to make room. Unfortunately, this may well involve moving up several handwritten pages of perfectly good program, by laboriously keying them in again at the new addresses. This is a nuisance, to say the least. A so-called 'block transfer' routine is much easier – in effect, you make the computer do the bulk of the work.
The basic principle is quite simple. Let's assume that one additional 'store absolute' instruction (3 bytes) is to be inserted at address ØAØØ. The remainder of the program, say from ØAØØ to ØAFE, will have to be moved up three places in memory. This can be accomplished as follows:

```
ØØ8CØ     Ø5FF        LODI, R1
     2  ┌►ØØD4AØØ     LODA, I-R1
     5  │  CD6AØ3     STRA, I/R1
     8  └─ 5978        BRNR, R1
     A     1FØØØØ     BCTA, UN
```

When this program is started, at address ØØ8CØ, all instruction bytes are moved up three address positions – one at a time, starting at the 'top'. In practice the computer does the job so fast that the display on the screen hardly flickers. The only modifications that must then be entered by hand are all positions containing absolute address instructions that refer to the

program section that has been moved, and any relative addresses that operate 'across the gap'.

It should be noted that this system can only be used safely when moving a section of program *up* in memory. In that case, the original data at any given address is always stored at the corresponding higher address before new data is written into the lower one. To move a program section *down,* a Load with auto-increment instruction must be used. The data is then moved down, starting at the lowest address:

```
08C0    0500        LODI, R1
   2  ┌►0D2A03      LODA, I + R1
   5  │  CD6A00     STRA, I/R1
   8  └─ 5978       BRNR, R1
   A     1F0000     BCTA, UN
```

Although these two routines work all right, they are not so easy to use. The main problem is that you must first calculate the correct initial value for R1, depending on how many program bytes you intend to move. A further (minor) problem is that you cannot move more than 256 bytes in one go; larger program sections must be moved in a succession of chunks. Both of

Table 31 **Block transfer *up*.**

```
1F60    xxxx        first address, old block
   2    yyyy        last address,  old block
   4    zzzz        last address,  new block

1F66    7660        PPSU,  flag + II
   8    08F8        LODR, R0 Ind
   A    C8F8        STRR, R0 Ind
   C  ┌─1B12        BCTR, UN
      │
1F80  └►B440        TPSU,  flag
   2    1E0000      BCTA
   5    7440        CPSU,  flag
   7    040A        LODI,  R0
   9    93          LPSL
   A    0502        LODI,  R1
   C  ┌►0D5F62      LODA, I-R1
   F  │  A400       SUBI,  R0
1F91  │  CD7F62     STRA, I/R1
   4  │  ED7F60     COMA, I/R1
   7  │ ┌1802       BCTR
   9  │ │ 7640      PPSU,  flag
   B  │ └►596F      BRNR, R1
   D  │   7501      CPSL,  C
   F  │   0502      LODI,  R1
1FA1  ┌►0D5F64      LODA, I-R1
   4  │  A400       SUBI,  R0
   6  │  CD7F64     STRA, I/R1
   9  └─ 5976       BRNR, R1
   B     1F1F68     BCTA, UN
Start address: 1F66
```

these problems can be solved by using the more extensive routines given in Tables 31 and 32.
It may seem strange that these routines are stored in the PVI 'background data' area. However, when working on a program (without running it!) this is one of the few memory areas that is never in use!
To use these routines, the first and last address of the block that is to be moved must be stored at addresses 1F60 and 1F62, respectively. Furthermore, for a transfer *up* the *last* address of the new block position is stored at 1F64; for a transfer *down* the *first* new address is stored here. The routine is then started at PC = 1F66.

Clear (load) RAM

With a few minor modifications, the same routine can be used to stored 00 (or any other value) in a whole block of memory. The program is given in Table 33.
This possibility is more useful than it may appear at first sight. In the first

Table 32 **Block transfer *down*.**

```
1F60    xxxx        first address, old block
   2    yyyy        last address, old block
   4    zzzz        first address, new block

1F66    7660        PPSU, flag + II
   8    08F6        LODR,R0 Ind
   A    C8F8        STRR, R0 Ind
   C    1B12        BCTR, UN

1F80    B440        TPSU, flag
   2    1E0000      BCTA
   5    7440        CPSU, flag
   7    040B        LODI, R0
   9    93          LPSL
   C    0502        LODI, R1
   F    0D5F60      LODA,I-R1
1F91    8400        ADDI, R0
   4    CD7F60      STRA, I/R1
   7    ED7F62      COMA,I/R1
   7    1802        BCTR
   9    7640        PPSU, flag
   B    596F        BRNR,R1
   D    7701        PPSL, C
   F    0502        LODI, R1
1FA1    0D5F64      LODA,I-R1
   4    8400        ADDI, R0
   6    CD7F64      STRA, I/R1
   9    5976        BRNR,R1
   B    1F1F68      BCTA, UN
Start address: 1F66
```

Table 33 Clear (or load) RAM.

```
1F80      xxxx        start address
   2      yyyy        end address
1F84      7660        PPSU, flag + II
   6      0400        LODI, R0
   8   ┌─►C8F6        STRR, R0 Ind
   A   │  B440        TPSU, flag
   C   │  1E0000      BCTA
   F   │  7440        CPSU, flag
1F91   │  770B        PPSL, WC + COM + C
   3   │  0502        LODI, R1
   5   │┌►0D5F80      LODA, I-R1
   8   ││  8400        ADDI, R0
   A   ││  CD7F80      STRA, I/R1
   D   ││  ED7F82      COMA, I/R1
1FA0   ││┌─1802        BCTR
   2   │└┼─►7640        PPSU, flag
   4   │ └►596F        BRNR, R1
   6   └──1B5E        BCTR, UN
```

Start address: 1F84.

Note: if desired, other data can be stored in the memory block by modifying the value specified at address 1F86.

place, it is a good idea to clear out the whole memory before starting to key in the first attempt at a new program. That way, when you start adding all sorts of 'patches' to cure the initial glitches, you can still tell free memory areas by the fact that they consist of a block of ØØs! Furthermore, it is often the case that the initial values in a whole 'scratch' block are all the same – all FF, say.

All three programs given so far are based on the same principles, but since this one is the shortest it is also the easiest to explain. There are only two 'tricks' involved:

- At each pass round the loop, the specified 'start address' is incremented. This is done by means of the carry bit. Initially, this bit is set, to indicate a 'carry', with the result that adding zero (at address 1F98) actually adds one to the lower address byte. If this causes the byte to increment from FF to ØØ, the resulting carry will also increment the upper address byte; otherwise, the upper byte will be left unchanged.
- Before each 'increment address' loop, the flag is cleared. After updating the lower address byte, this is compared with the lower byte of the 'end address'; if the two are not identical, the flag is set. The same test is repeated after updating the upper address byte. The result is that the flag is always set, except when both the lower and the upper address bytes of the new 'start address' and the 'end address' are identical. In other words, when we have reached the last address of the block! After storing the desired value into this address, the 'flag test' at 1F8A will turn out negative, in that case, so that the program returns to monitor.

Disassembler

A useful routine, this! It can be used for scanning existing programs. As shown in Table 34, the first address of the program (section) to be displayed is stored at address Ø8CØ, after which the routine is started at Ø8C2. The addresses and instructions will then roll up the screen automatically. At any time, the display can be 'frozen' by operating the upper control key and then re-started by means of the start key.

If desired, the 'speed' can be altered by modifying the instruction at address 1F9B: '87Ø2' slows it down, and '87Ø8' or '871Ø' speeds things up. Other values are not advised...

When plotting Tables, it can be useful to select a fixed number of bytes per line. This is achieved by modifying the instruction at address Ø8FØ to Ø6Ø4 (single byte), Ø6Ø8 (two bytes) or Ø6ØC (three bytes per line). It should be noted that during normal operation the first few instructions after a series of data values may be misinterpreted, until the routine gets back into synchronism.

One further note applies to all the routines given so far. To return to Monitor, the reset key must be operated. When in the monitor mode, operating the 'start' key will erase all programs that are stored in the PVI area!

Obviously, this program cannot be used for evaluating a program section that is located between Ø8CØ and Ø8F6. That little bit will have to be done by hand! At a later date, when the extension board has been added, this problem will resolve itself: the program can be located in a section of memory that will not be used for normal programs. In fact, extended versions of all the routines described so far are incorporated in an 'edit routine' that fits into that memory area.

When it comes to actually using the disassembler routine, one practical problem remains to be solved. As it stands, this routine gives the addresses and instructions – in machine language! In other words, the routine itself, from 1FØØ on, will appear on the screen as follows:

```
1FØØ
  87Ø4
1F82
  ØD88A4
1F85
  3FØ354
etcetera
```

Interpretation of the instruction codes (87Ø4, ØD88A4, etc.) is left to the user. Fortunately this is a fairly simple matter, thanks to the 'logical' instruction set for the 2650. This is illustrated in the 'decode table' given in figure 32. The first two digits of an instruction determine what it does, in general terms; the rest fill in the details. In figure 32, the first digit is plotted down the left-hand side, and the second runs across the top. For the first digit, even numbers are given in the upper half and odd numbers in the lower.

Instructions can now be decoded without much effort. Take '87Ø4', for instance. The first digit is '8', indicating the ADD row; the second is '7',

Table 34

08C0	xxxx	first address of block:	
2	7620	PPSU,II	
4	0502	LODI,R1	
6	►0D48C0	LODA,I-R1	transfer start address
9	CD68A4	STRA,I/R1	
C	└5978	BRNR,R1	
E	3F02CF	BSTA,UN	scroll
08D1	3F042B	BSTA,UN	
4	0706	LODI,R3	address to
6	►CF488A	STRA,I-R3	MLINE
9	└5B7B	BRNR,R3	and scroll
B	3BF2	BSTR,UN,Ind.(02CF)	
D	0E88A4	LODA,Ind,R2	
08E0	F608	TMI,R2	
2	┌180C	BCTR	
4	F6A0	TMI,R2	
6	┌1806	BCTR	1-, 2- or 3-byte
8	2640	EORI,R2	instruction?
A	F654	TMI,R2	Result in R2 as
C	┌1802	BCTR	01, 02 or 03
E	►8604	ADDI,R2	
08F0	►460C	ANDI,R2	
2	52	RRR,R2	
3	52	RRR,R2	
4	1F1F80	BSTA,UN	

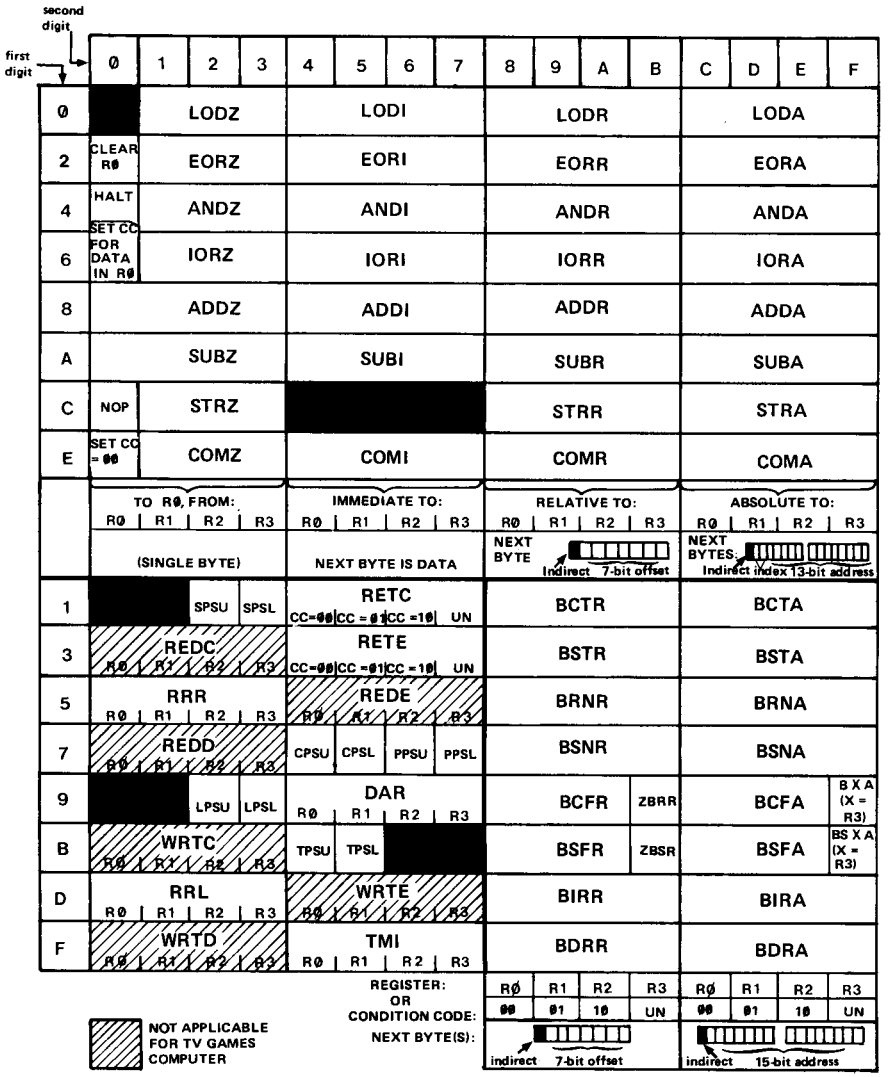
1F80	►8704	ADDI,R3	
2	0D88A4	LODA,R1,Ind	1, 2 or 3 databytes
5	3F0354	BSTA,UN	to display;
8	3F039F	BSTA,UN	increment address
B	└FA73	BDRR,R2	
D	3F020E	BSTA,UN	
1F90	┌►0C1FCB	LODA,R0	
3	D0	RRL,R0	wait for VRLE;
4	└9A7A	BCFR	display 6 lines
6	7702	PPSL,COM	
8	3F0055	BSTA,UN	
B	8704	ADDI,R3	
D	0C1E8B	LODA,R0	SPEED
1FA0	┌9A02	BCFR	stop if upper
2	6701	IORI,R3	control key
4	└►D0	RRL,R0	
5	┌9A02	BCFR	start if
7	47FE	ANDI,R3	start key
9	└►5B65	BRNR,R3	delay (or stop)
B	1F08CE	BSTA,UN	repeat

Start address: 08C2.

Notes:— to decrease the speed, modify the instruction at address 1F9B to 8702; to increase it, use 8708 or even 8710

— for plotting tables, the instruction at 08F0 can be modified to 0604, 0608 or 060C.

Figure 32.



82901-32

which is the highest number in the ADDI section. Looking down a few lines, we find that this corresponds to an 'add immediate to register 3' instruction and that the next byte (Ø4) is data. Therefore, Ø4 is added to the value in R3.

Once the underlying principles of the instruction set have been recognised,

it is a relatively simple matter to memorise the whole table. In fact, this will come almost automatically with practice. A few pointers may help:

- even-numbered first digits define load, store, arithmetic and logic instructions. These are all available (with only a few obvious exceptions) in 'to register zero', 'immediate', 'relative' and 'absolute' versions. In the two latter cases, the highest bit of the next byte is set when indirect addressing is required: ØDØ8A4 for direct addressing, ØD88A4 for indirect. For absolute addressing, the two following bits in the second byte determine the 'index' mode.
- odd-numbered first digits, together with a second digit in the 8...F range, define branch instructions. Depending on the type of instruction, the second digit may determine either a register or a condition code. Thus, '18...' stands for 'branch on condition true' where the condition code (as shown below) must be ØØ. On the other hand, '58...' defines 'branch on register non-zero'; the register in question is RØ. As before, the first bit of the next byte is set for indirect addressing. For absolute addressing, however, no 'indexed' version is normally available: all 15 remaining bits determine the address, from ØØØØ to 7FFF!
- odd-numbered first digits, together with a second digit in the Ø...7 range, correspond to various 'miscellaneous' instructions. Those shown with light shading are not applicable in the TV games computer.

Byte savers

Strictly speaking, these are not routines. They are combinations of instructions, or even single instructions, that are just one or two bytes shorter than 'standard procedure'. This type of trick can be extremely useful when you want to add a few instruction bytes to an existing program (a branch to subroutine, say) without resorting to an extensive 'block transfer' operation.

In this light, we must start with a word of warning. In general, it is not a good idea to make full use of these ideas in the initial version of a program. They should be reserved, for emergency use only, when clearing out the very last imperfections in an otherwise fully-fledged program.

- EORZ, RØ ('2Ø'). This one was mentioned earlier; it is one byte shorter than 'Ø4ØØ'.
- Store indexed to register zero. According to the 2650 manual, this isn't possible. In practice, it is: the data value is used both as data and as index. This is easily illustrated by the following routine:

```
ØØØØ   76ØØ      PPSU, II
   2   ØØ         EORZ, RØ
   3 ┌►CC2AØØ     STRA, I + RØ
   6 └─587B       BRNR, RØ
   8   1FØØØØ     BCTA, UN
```

Having run this routine, you will find that the data stored from ØAØØ on is equal to the lower address byte: ØØ, Ø1, Ø2 and so on. Intriguing – but what is the practical value? In programs, the values ØØ and FF must often be stored somewhere – to preset a scratch byte, disable an object, turn off the sound, or whatever. As an example, let's assume that we must disable

the sound and put object 1 off-screen. A straightforward routine would be:

```
0400    LODI, R0
CC1FC7  STRA, R0
04FF    LODI, R0
CC1F0A  STRA, R0
```

Ten bytes in all. Now we can use the two tricks described above, to obtain the same result:

```
20      EORZ, R0
CC1FC7  STRA, R0
CC5E0B  STRA, I-R0
```

Believe it or not, the third instruction will cause FF to be stored in 1F0A! Since it is an autodecrement, the value in the specified index register (R0!) is first decremented from 00 to FF; then this index is used to determine the absolute address: 1E0B + FF = 1F0A. All in all, this saves three bytes – enough to insert a Branch-to-subroutine, for example.

- Re-use existing data values. One of the most common examples is the situation after running a 'loop'. Normally, this is concluded when the data in one of the registers becomes zero. It is rather pointless, in this case, to continue as follows:

```
.....
5976    BRNR, R1        end of loop
0400    LODI, R0
CC1FC7  STRA, R0
```

Instead, you can save two bytes:

```
.....
5976    BRNR, R1
CD1FC7  STRA, R1
```

After all, the value in R1 must be zero at the end of the loop!

- Make full use of existing condition codes. Any operation that transfers data to or modifies data in a register will set the condition code. Again, a common example: testing for operation of the upper control key. First the obvious way:

```
0C1E8B  LODA, R0
F480    TMI, R0
18..    BCTR
```

Now stop a minute, and think it over. You want to know whether the highest bit in 1E8B is set or not. If it is, the value is equivalent to a negative number! Therefore, the following routine must do exactly the same job:

```
0C1E8B  LODA, R0
1A..    BCTR
```

Enough for now. The examples given above are the most common ones, although the list is by no means complete. Suffice it to say that, when you need to squeeze in those last few bytes, it will often pay to take a closer look at the surrounding section of program. In the majority of cases, a little trickery will make just enough room!

A completely different category of 'byte-savers' is contained in the monitor program. But that is the subject of the next chapter.

16

A closer look at the monitor program

Knowing the monitor software can be extremely useful when developing your own programs. Some examples were given in chapter 11, where we explained the keyboard scan and text routines. Even though those explanations were fairly detailed, they are by no means complete; the routine used in Table 28 to display the joystick value, for instance, was not included.
However, the number of potentially useful monitor routines is so extensive that it just isn't feasible to give 'instructions for use' for every possible application of each routine. The only solution is to explain the routines themselves, so that interested readers can work out their own variations as required.

In order to explain the monitor program, the first requirement is to reproduce the complete 'listing'. There is another way, of course, as many readers have discovered: use a disassembler routine, like the one given in Table 34, and work the whole thing out by hand. However, this is extremely time-consuming when you are talking about a complicated and lengthy program of this nature. (Indeed, it is fortunate that Philips, the originators of the monitor software, supply it in a 2616 ROM that plugs straight into the board. We shudder to think how many disenchanted readers there might be if everyone had to enter it by hand into EPROM!)

A further problem, when you try to decode a program by hand, is that you have no advance information on what the various sections are supposed to do. To avoid all these problems, Philips have kindly granted us permission to reprint the complete original documentation for the purposes of this book. A quick look at the relevant section of the Appendix will illustrate the points made above: just try to visualise decoding all that instruction and program data by hand!

The complete program can easily be broken into individual sections. In the following, we will deal with these in the order in which they occur. First, however, we need a brief explanation of how to read the monitor program as given in the Appendix. After a short summary of the abbreviations used, the start of the actual program is given as shown in Table 35.

Table 35

```
LOC  OBJECT   ADDR  SOURCE  LINE

0000 1F000A   000A  RESET   BCTA,UN START

                          *INTERRUPT ADDRESS H'0003'
0003 1F88B9   08B9  INTRR   BCTA,UN *INTADR VIA RAM ADDRESS

                          *VECTORS FOR BREAKPOINT ENTRY
0006 0594           BVEC1   ACUN BREAK 1
0008 05B2           BVEC2   ACUN BREAK 2

                          *AT RESET THE SYSTEM ENTERS THE MONITOR
                          *INITIATION IS DONE AFTER POWER UP OR IF RESET+INITIATE KEY

000A 7620           START   PPSU II
000C 0C08B9   08B9          LODA,R0 INTADR    WAS USER ACTIVE?
000F E409                   COMI,R0 H'09'
0011 3C05CD   05CD          BSTA,EQ BRKSBR    THEN SAVE STATUS AND CLR BREAKPOINTS
0014 20                     EORZ R0
0015 CC089A   089A          STRA,R0 MFUNC     SET NO FUNCTION
0018 0518                   LODI,R1 24                LAST LINE OF MON. DISPLAY ME
001A 6D4878   0878          IORA,R0 MONOB+(NLINES-1)*24,R1,- IS TESTED FOR POWER UP
001D 597B     001A          BRNR,R1 $-3
001F 4411                   ANDI,R0 H'11'     THESE POSITIONS ARE ZERO IF RUNNING
0021 1815     0038          BCTR,Z START1
0023 7660           INIT0   PPSU FLAG+II      INITIATE SYSTEM STATUS
0025 20                     EORZ R0
0026 05C0                   LODI,R1 > (PC-MONOB+2)
0028 CD5F00   1F00  INIT1   STRA,R0 RAM,R1,-  INITIATE PVI
002B CD6800   0800          STRA,R0 MONOB,R1  INITIATE SCRATCH
002E 5978     0028          BRNR,R1 INIT1
0030 0628                   LODI,R2 40        MESSAGE 'IIII'
0032 3F0602   0602          BSTA,UN WCAS3
0035 3F0161   0161          BSTA,UN SAVE1     INITIATE FOR MONITOR DISPLAY
0038 044B           START1  LODI,R0 > MONINT  SET MONITOR INT ADDRESS
003A CC08BA   08BA          STRA,R0 INTADR+1
003D 0400                   LODI,R0 < MONINT
003F CC08B9   08B9          STRA,R0 INTADR
0042 CC089F   089F          STRA,R0 MKBST     ASK KEY ENTRY
0045 7702           MWAIT0  PPSL COM
0047 7427           MWAIT1  CPSU II+SP        CLEAR II
0049 1B7C     0047          BCTR,UN MWAIT1    WAITING FOR INTERRUPTS
```

The information for each line is given in the following sequence:

- LOC (for 'location'): this is the address, in memory, where the instruction is located.
- OBJECT: the instruction, data or whatever at that address.
- ADDR (for 'address'): this is the address (if any) referred to in the instruction. The very first line, for instance, is a Branch to 000A.
- SOURCE: this is a 'label' for that particular line, which is used when referring to it at other points in the program. At 000A, for example, the 'source name' is START; the instruction at 0000 is therefore described as 'BCTA, UN START' (short for: branch to START).
- LINE: the mnemonics for the instruction, or description of the data on that line. Although these are presented in a slightly different way than the system we have been using so far, the intention is clear. A few pointers may help:
  - H'09', say, indicates the hexadecimal value 09;
  - STRA,R0 RAM, R1- (at line 0028) indicates that the value in R0 is

stored in RAM as indicated by the value in index register R1, after decrementing the latter. So far, we have indicated this as 'STRA, I-R1'.

- Intermediate comments, clarifying the following section, are indicated by the * sign.

Now, having laid the groundwork, we can take a closer look at the monitor software as it is given in the Appendix.

Initialisation

The program starts with a branch to 'start', followed by an indirect branch to the start of the interrupt routine. The actual 'interrupt address' (ØØ4B) is stored at Ø8B9 and Ø8BA; when starting a 'user program', the monitor stores 'Ø9Ø3' here. Then come the two breakpoint addresses, Ø594 and Ø5B2, after which the actual main program starts.

This program is quite short (from ØØØA to ØØ4A). It first checks to see whether a 'user program' was running (in that case, Ø9 is stored at Ø8B9 for the interrupt address); if so, the subroutine at Ø5CD is run to save the register values and program status and clear the breakpoints. This, by the way, explains why 'Ø9' is always found in RØ when returning to monitor via address ØØØØ.

The next step is to test the last line in the monitor display area to see whether this contains valid text. If not, a short initialisation routine is run: set flag, clear PVI, clear monitor scratch, display message 'IIII'. Then the monitor interrupt address (ØØ4B) is stored in Ø8B9 and Ø8BA; Ø89F is cleared in preparation for key entries; the COM bit is set and Interrupt Inhibit is cleared. The program now peters out into a loop 'waiting for interrupts'.

Text display

The interrupt routines start at ØØ4B with a check for 'vertical interrupts only'. Then the keyboard scan subroutine at Ø181 is run; if a key is operated, the program branches to ØØBD for further evaluation. If no key is operated, on the other hand, the next step is to set the object positions for the text display.

Then, after a 'wait for *end of* vertical reset' loop (at ØØ71), the actual text display is executed. In essence, this consists of a wait loop ('for top of display'); then a data transfer loop, to load the object shapes for the first line; then the delay loop is repeated (delay between lines), followed by the data transfer for the next line of text; and so on, until all six lines of text have been displayed. It may be helpful to note that this routine uses R1 as index register for retrieving the display data from the RAM area from Ø8ØØ on, while R2 is used as the index for storing the data in the first six lines of each object. This means that 'R2 = Ø6' indicates that all data for one text line has been loaded, whereas 'R1 = 8F' signals the end of the last text line. Finally, the vertical offset for all duplicates is set to FE to disable the objects. The object position data for all four objects during this routine is given from ØØAD to ØØBC.

At this point, it is interesting to note the major (if unintentional!) advantage of this monitor program layout. Since virtually the entire program runs as interrupt routines, nearly all major sections are terminated with a Return instruction. This means that they can easily be incorporated in any program by means of a Branch to subroutine instruction. In this particular case, and depending on the application, subroutines can be started at ØØ55 (for a complete text display, after setting the COM bit), at ØØ7A (for a partial text display, after setting R1 and R2 as required) or at ØØ9E (to clear the duplicates).

Key evaluation

As mentioned above, this section is reached after the 'vertical interrupt' when the keyscan shows that a key is operated. At this point the data in R1 indicates the key, as described in chapter 11 (figure 29a in particular). Although this section (from ØØBD on) is of little practical value for 'user programs' (it merely initiates monitor routines as required) we can run through it quickly.
First, the 'function code' is loaded into R3 (this indicates which monitor routine is currently selected) and the key code is 'translated' into the data given in figure 29b. The translation table for this conversion is stored between Ø122 and Ø13D. Then, depending on the key entry, one of three things happens:

- for digit keys (Ø...F), the routine from ØØC7 stores the corresponding value in Ø8A1 and then, after a few presets, runs the selected monitor routine: 'Branch to subroutine, indexed, absolute' (at ØØD1). The index is the function code in R3; this selects either the return instruction at Ø1ØC or one of the branch instructions between Ø1ØD and Ø121.
- for function keys, the routine from ØØE2 clears the 'function scratch' (from Ø8AØ on) and branches back to ØØCA to run the selected monitor routine as described above.
- for + or − keys, the routine from ØØFD again presets a few values before branching back to ØØCD to run the monitor routine.

Addresses Ø1ØC to Ø13D contain data, as outlined above.

Save status

This routine, from Ø13E on, starts by saving the Program Status values and register contents at addresses Ø8AC...Ø8B4. Then the PVI control data for the monitor mode are retrieved from Ø177 on and stored from 1FCØ on. Finally, the routine starting at Ø16E clears all object data. On the odd occasion, this last routine can prove useful.

Keyboard scan

The 'instructions for use' of this routine were given in chapter 11. There is little point in going into detail on how it works. Suffice it to say that it consists of a main control loop from Ø181 to Ø1BF, a key evaluation routine from Ø1CØ to Ø1ED and a set-key-code routine from Ø1EE to Ø2ØD. It is

unlikely that any of these routines will prove of practical use, except for their intended application: keyboard scan.

Preparation for text display

The routines from Ø2ØE on are of greater interest, since they set the 'object shape' data for a text display. It is often possible to obtain interesting text variations by jumping to these routines at some intermediate point, with different preset values than those used by the monitor program. For this reason, it is worth going into slightly greater detail on this part of the monitor.

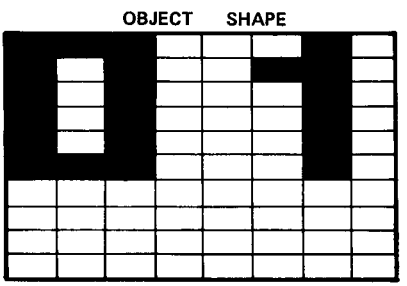
It was explained above, under 'text display', how the shape data for six display lines is transferred from the RAM area (from Ø8ØØ to Ø8FF) to the object areas of the PVI. At some earlier stage, the desired text must obviously have been converted into this 'shape data'; this is done by the routine that starts at Ø2ØE. In order to understand this section, it is essential first to study the basic principles.

Each of the four objects is used to display two characters in each line. This means that each byte that defines one line in an object can be broken up as follows: the three highest bits (the left-hand three) correspond to the first character; the next bit is zero, for the space between characters; the following three bits belong to the second character; and, finally, the extreme right-hand bit is zero. The same system is used for packing the basic data into the 'image table', from Ø27B to Ø2CE, so this can be used as an illustration. The first six bytes in this table give the data for the characters '0' and '1'. As illustrated in figure 33, the left-hand half of each byte contains the data for the '0' shape and the right-hand half is used for '1'. The lower four lines in all objects are left blank, for the space between lines.

In order to preset the 'shape data' area for one object, the six 4-bit data groups for the right-hand character must first be retrieved from the image table, and stored as the four right-hand bits in the corresponding six shape

Figure 33

IMAGE TABLE ADDRESS	DATA HEX	DATA BINARY
Ø27B	E2	1110 0010
Ø27C	A6	1010 0110
Ø27D	A2	1010 0010
Ø27E	A2	1010 0010
Ø27F	A2	1010 0010
Ø280	E2	1110 0010



82901 - 33

bytes. Then a similar operation must be executed to retrieve the 4-bit data groups for the left-hand character and add them to the data already stored in the shape bytes. This is done as follows.
At Ø2ØE, the first step is to preset R1 for the desired number of characters per line – 8, for monitor routines. This value will be decremented to point to each character in turn, from right to left along the line. Then, since R1 is even (this indicates that a new pair of characters must be converted), the RAM area from Ø878 to Ø87D is cleared. The next character code is retrieved from the 'monitor line' (Ø89Ø...Ø897), at the address indicated by R1, after which R1 is stored in Ø89B. The next step is to preset R2 to indicate the current situation: Ø2 or Ø3 for a right-hand character in the image table and ØØ or Ø1 for a left-hand character; the odd numbers (Ø1 and Ø3) indicate that the 4-bit data retrieved from the image table must be shifted four positions before storing it as 'shape data'. This situation occurs when a 'right-hand' character in the image table must be stored in the left-hand object position, or vice versa.

This brings us to Ø23A: a branch to the subroutine at Ø24B that calculates the index required to retrieve the character data from the image table. The sections from Ø23C to Ø24A and Ø255 to Ø25A load the indicated data byte, mask out the desired 4-bit group, and branch either to Ø25B (if the data must be shifted) or to Ø25F. The data is 'merged' into that present in Ø87D, and the loop is repeated, from Ø23C, until all data groups for this character have been stored from Ø878 to Ø87D.
The 'character position in line' index is now retrieved; if this is zero, all eight characters are in position so the routine ends. Otherwise, the shape data is moved up six positions to make room for the next 'merge', if necessary, and the whole routine is repeated from Ø212.

What have we achieved, so far? Eight data bytes in the 'monitor line', indicating the eight characters for one text line, have been converted into shape data and stored from Ø878 to Ø88F. The 'text display' routine described earlier will display this as the lowest line on the screen. An essential step (and a few little ones) is still missing: a routine to move the text up the screen as each new line is added. This is indeed the next routine – after the image table data from Ø27B to Ø2CE.

Scroll

This routine, starting at Ø2CF, first moves all data from Ø818...Ø88F down to Ø8ØØ...Ø877. This moves lines 2 to 6 up one position on the screen. Note that the sixth line is not erased: it now appears twice, in the fifth and sixth positions.
After this the code '17', corresponding to 'space', is loaded into the eight 'monitor line' positions. Running the routine from Ø2ØE at this point would therefore erase the bottom line. Alternatively, two or three characters can be specified first; all other positions will remain blank.

Output new message

Putting one of the monitor messages on the screen is a simple matter. R2 is preset, to indicate the desired message: Ø4 for 'PC =', Ø8 for 'AD =' and so on. Then the subroutine from Ø2E3 is run.

This routine first executes the 'scroll' procedure outlined above – if this is undesirable, the routine can be started at Ø2EA. Then the four indicated character codes are retrieved from the 'message line table' (from Ø2F5 to Ø31C) and loaded into the four left-hand positions of the 'monitor line'. Now, running the routine from Ø2ØE will place the text on the screen.

Assemble data

The message line table, mentioned above, is followed by an 'assemble data' routine that starts at Ø31D. Before using this routine, ØØ must first be stored at address Ø8A7. This indicates 'first digit', and ensures that the two data assembly locations (Ø8A2 and Ø8A3) are cleared initially. Up to four digits (ØØ to ØF) can now be stored at Ø8A1; running the 'assemble' routine between each entry will cause them to be entered and shifted left in addresses Ø8A2 and Ø8A3.

The routine is interesting in itself, since it makes good use of the carry bit. At address Ø326, the subroutine at Ø346 is run; this loads the new digit into RØ and the lower and upper assembly bytes (Ø8A3 and Ø8A2) into R1 and R2, respectively. The new digit is then shifted left so that Ø5, say, becomes 5Ø. At this point, we reach the actual assembly loop (starting at Ø33Ø). First RØ is shifted left; if the left-hand bit is '1', this will cause the carry bit to be set. Then R1 shifts left; if the carry bit is set, due to the RØ shift, the lowest bit in R1 will now be set to '1'. In the same way, bits are shifted out from the left-hand end of R1 and into the right of R2. This procedure is repeated, four times in all, until the original data in R1 and R2 has moved one digit to the left and that from RØ has been added at the right-hand end. Finally, the end result is stored in Ø8A2 and Ø8A3.

To see this routine in action, operate the PC key and then start hitting digit keys (Ø to F)!

Output double byte to monitor line

The routine described above assembles a 'double-byte'. This can be loaded into the monitor line, for display on the screen, by means of the routine starting at Ø34E. As before, R1 must contain the lower byte and R2 the upper; R3 determines the position of the data in the display. Note that R3 is preset to Ø8 at Ø34E – for a display in the right-hand positions – but if this is undesirable, a different value can be selected, after which the routine is started at Ø35Ø.

In fact, the routine makes use of a subroutine at Ø354 that outputs a single byte to the monitor line. For double-byte operation, this routine is run twice; the byte that must be transferred to the monitor line is contained in R1. The data in R1 is copied into RØ and there shifted right four places; masking out the lower four bits in RØ and R1 leaves the two 'separated digits' in these registers. The results are stored in the monitor line positions specified by R3.

The following routine, from Ø368 to Ø376, is of little interest; it uses the routines described above to 'output a new address'. With leading spaces, since it runs on to the next section:

Output of spaces

Quite simply, this routine (starting at Ø377) loads the code for 'space' into the monitor line. The position of the first space is set by R3 (or R2, if you start at Ø379) and the number of spaces is determined by R1.

Store data in memory

Running from Ø381 to Ø39E, this routine is of no practical value for user programs. Even so, there are three points worth noting:

- Storing data into the monitor scratch area (Ø8ØØ to Ø8BF) by means of the monitor routines is blocked. In monitor mode, this data can only be read.
- The 'test if store was successful' routine is disabled from address 18ØØ on. In the extended version, to be described later, this area contains RAM. Not that we have ever known a 'store' to be 'unsuccessful'.
- At address Ø399, the instruction 'EØ' (= COMZ, RØ) is used. Since the data in RØ is obviously equal to the data in RØ, this sets the condition code to ØØ, for 'equals'. Another useful trick!

Increment double byte (address)

This routine (from Ø39F) can be more useful in practice than would appear at first sight. It adds one to the two data bytes contained in Ø8A2 and Ø8A3. For example, 74FF will be incremented to 75ØØ. Again, good use is made of the carry bit.

Data display and alter

The section from Ø3BØ to Ø4ØB is of little interest. It displays and alters the register and program status values for the user routine, as stored from Ø8AC to Ø8B4. The same applies for the 'memory display and alter' routines from Ø4ØC to Ø4A8, and the 'breakpoint set and clear' routines from Ø4A9 to Ø5ØD. While we are at it, we can also include the PC routine (from Ø5ØE to Ø535) in this list.

However, all these routines have one useful feature in common: they all make use of the text routines, and sometimes provide the possibility to combine several of these into a single 'branch to subroutine' instruction. To give just a few examples:

- the section from Ø47Ø will output the two bytes in Ø8A4 and Ø8A5 as the first four digits on the screen, followed by two spaces and then the byte from Ø8A3 in the last two positions. An 'address-data' display mode, in other words.
- the section from Ø52F can be considered as an extension of the 'assemble data' routine, from Ø31D, described above. As before, Ø8A7 must be preset to ØØ initially; then up to four digits can be entered (shifting left). They will be displayed in the first four positions on the bottom line.

Start up

The locations from Ø537 to Ø53F contain data for the 'start up' routine – this is the routine that starts a program, after you operate the PC key followed by '+'. Locations Ø537 and Ø539 contain the ZBRR instructions for the breakpoints; the data from Ø53B is actually used for the last few instructions before branching to the user program, as we will see.

The actual 'start up' routine begins, at Ø54Ø, by copying the new PC value into Ø8BE and Ø8BF. Then, if breakpoints are specified, the data at these addresses is saved in RAM and replaced by the ZBRR instructions mentioned above. The next step, from Ø562, is to transfer the data from Ø53B on to Ø8B9...Ø8BD. In combination with the PC value that was transferred earlier, this leaves the following instructions from Ø8B9:

```
Ø8B9    Ø9Ø3        (interrupt address!)
   B    77ØØ        PPSL
   D    1FØ9ØØ      BCTA, UN        (to new PC address)
```

The original user's 'Program Status, lower' data is now retrieved from Ø8B3 and inserted at Ø8BC. After this, the original register values and PSU data are also restored; finally, the PSL is cleared and the program branches to... Ø8BB! As we have just seen, this will restore the original PSL value after which a further branch takes us to the specified 'new PC address'.

We have gone into this routine in slightly greater detail, since it offers a few interesting possibilities. In the first place, it is possible to specify a different 'interrupt address', by modifying the data in Ø8B9 (provided you disable the interrupts while you're doing it!). This makes it possible to specify different interrupt addresses for different program sections, which may come in handy.

Table 36

```
Ø9ØØ    1Fxxxx    Jump to main program
Ø9Ø3    CCØ8AC    STRA, RØ
   6    13        SPSL
   7    CCØ8B3    STRA, RØ
   A    12        SPSU
   B    CCØ8B4    STRA, RØ
   E    751Ø      CPSL, RS
Ø91Ø    CDØ8AD    STRA, R1
   3    CEØ8AE    STRA, R2
   6    CFØ8AF    STRA, R3
   9    771Ø      PPSL, RS
   B    CDØ8BØ    STRA, R1
   E    CEØ8B1    STRA, R2
Ø921    CFØ8B2    STRA, R3
   4    ┌─3BØ9    BSTR, UN
   6    │ Ø42E    LODI, RØ
   8    │ CCØ8BF  STRA, RØ
   B    │ 1FØ562  BCTA, UN
   E    │ 37      RETE, UN
Ø92F    └─►....   Interrupt routine starts here.
```

A further possibility is to use the 'restore status' section to make a slightly shorter interrupt routine than that given in Table 22 (chapter 12). If only the 'save status' routine at Ø13E was equally accessible! The result is shown in Table 36. There are a few points worth noting, when using this routine:

- As it stands, use is made of the fact that the main program starts at Ø9ØØ and the desired return address from the 'restore' routine is Ø92E. This means that only the lower byte must be changed before branching to Ø562.
- If the routine is located from Ø8CØ on (Ø9Ø3 becomes 1FØ8CØ), nearly all absolute addresses can be replaced by relative versions. This saves ten bytes – more than enough to make up for the four required to modify Ø8BE as well.

It is even possible to go a step further: an even shorter routine is given in Table 37.

Breakpoint entries

Another section of little interest. When breakpoint one is found, the ZBRR instruction at that point initiates a jump via ØØØ6 to Ø594. Similarly, the second breakpoint results in a jump via ØØØ8 to Ø5B2. In both cases, the register data is saved, the original data at the breakpoint is restored, and the 'program counter' address is set to that of the breakpoint.
The only point that might prove useful is the section from Ø5C4. This stores the data in R2 and R1 into Ø8BE and Ø8BF, respectively, before running the routine at Ø34E that displays these two bytes in the last four positions of the lower text line.

Table 37

Ø8CØ	C86A	STRR, RØ
2	13	SPSL
3	C86E	STRR, RØ
5	12	SPSU
6	C86C	STRR, RØ
8	751Ø	CPSL, RS
A	C961	STRR, R1
C	CA6Ø	STRR, R2
E	CB5F	STRR, R3
Ø8DØ	771Ø	PPSL, RS
2	C95C	STRR, R1
4	CA5B	STRR, R2
6	CB5A	STRR, R3
8	3Fxxxx	BSTA, UN to interrupt routine
B	Ø437	LODI, RØ
D	C85E	STRR, RØ
F	1FØ56C	BCTA, UN
	....	
	....	
Ø9ØØ	1Fxxxx	Jump to main program
Ø9Ø3	1FØ8CØ	Jump to interrupt routine

Cassette routines

WCAS is a long and complicated routine (from Ø5E8 to Ø7Ø8); RCAS is no better – it runs from Ø7Ø9 to Ø7FF (the actual start address is Ø758). A full explanation would require several pages of flow charts, pulse diagrams and so on. Suffice it to say that these routines work! The only interesting point, as far as the actual operation goes, is a comparison between the WCAS output and what RCAS will accept as input:

- Pulse width: each output pulse is approximately 110 μ s wide. The RCAS routine will accept any pulse width between 60 μ s and 180 μ s. In other words, the nominal pulse width plus or minus 50%!
- Sync pulse code: WCAS outputs 9 pulses; RCAS will accept 8, 9 or 10.
- Logic level code: WCAS outputs 3 pulses for a 'Ø' and 6 for a '1'; RCAS accepts 2...4 and 5...7, respectively.

As before, it is a good idea to run through these routines briefly, on the lookout for useful subroutines. In fact, there are a few:

- The 'output new message' routine was described earlier: preset R2 for the desired (monitor) message and branch to the subroutine at Ø2E3. If this is to be followed by actually putting the message on the screen (using the subroutine at Ø2ØE), it is just as easy to start at Ø6Ø2.
- The routine from Ø6F7 can be used as a delay routine: it gives just over 200 μ s delay, multiplied by the value in R3. For R3 = Ø1, this is approximately one-hundredth of the picture height.

Monitor scratch

The RAM area from Ø8ØØ to Ø8BF is used by the monitor. It can be useful to know how the various areas are used – both when using this scratch area for programs and when modifying monitor routines. An overview is given in Table 38.

Table 38

Ø8ØØ . . . Ø817	shape data, first text line
Ø818 . . . Ø82F	shape data, second text line
Ø83Ø . . . Ø847	shape data, third text line
Ø848 . . . Ø85F	shape data, fourth text line
Ø86Ø . . . Ø877	shape data, fifth text line
Ø878 . . . Ø88F	shape data, sixth text line
Ø89Ø . . . Ø897	'monitor line': eight character codes for display
Ø898	BP function indicator/RCAS silence counter
Ø899	+/– enter key memory
Ø89A	monitor function index
Ø89B, Ø89C	scratch for keyboard routine
Ø89D	right keyboard status
Ø89E	left keyboard status
Ø89F	combined keyboard status
Ø8AØ . . . Ø8AB	function scratch, used by memory and data routines
Ø8AC . . . Ø8B4	save registers, PSU and PSL area
Ø8B5, Ø8B6	breakpoint address 1
Ø8B7, Ø8B8	breakpoint address 2
Ø8B9 . . . Ø8BF	startup program (start address at Ø8BE, Ø8BF)

Summary of monitor routines

In this chapter, we have discussed a large number of monitor routines that can prove useful in programs. A brief summary will help programmers to locate the routine they are looking for in a particular application. This is given in Table 39.

Table 39

Useful monitor routines

Routine description	Presets, initial data; comments	Start address	Registers used R0	R1	R2	R3
keyboard scan	clear 089F; see chapter 11	0181	x	x	x	
translate key codes	451F, 0D6122; see figure 29	—	x	x		
clear duplicates	VOD 1 . . . 4 is 'FE'	009E	x			
load VOD duplicates	data in R0	00A0	x			
clear objects	1F00 . . . 1F4D is '00'	016E	x		x	
load objects	data in R0 to 1F00 . . . 1F4D	016F	x		x	
split register	R1 = XY; then R0 = 0X, R1 = 0Y	035E	x	x		
increment double byte	data in 08A4, 08A5	039F	x	x		
delay	delay is (R3 data) x 200 µs	06F7		x		x
assemble data	clear 08A7; enter successive digits (00 . . . 0F) in 08A1; result shifts left in 08A2 (= R2) and 08A3 (= R1)	031D	x	x	x	x
Text routines:						
initiate PVI		0161	x	x	x	
display six lines	set COM bit; wait for VRLE	0055	x	x	x	
display last line only	set COM bit, VC objects, R1 = 78, R2 = 00	007A	x	x	x	
translate MLINE to display	see Table 17 for codes	020E	x	x	x	x
load 8 spaces to MLINE		02D9	x		x	
load spaces to MLINE	first position in R2, number of spaces in R1	0379	x	x	x	
scroll, 8 spaces to MLINE		02CF	x	x	x	
output single byte to MLINE	data in R1, position in R3	0354	x	x		x
output single byte to screen	data in R1, position in R3	0480	x	x		x
output double byte to MLINE	data in R2/R1, position in R3	0350	x	x	x	x
output double byte to MLINE	data in R2/R1, last positions	034E	x	x	x	x
output double byte to screen	data in R1/R2; last positions	0529	x	x	x	x
output three bytes to screen	data in 08A4, 08A5 to first positions; 08A3 to right	0470	x	x	x	x
address + data to MLINE	address in 08A4, 08A5	042B	x	x	x	x
output message to MLINE	R2 is message code (note 1)	02EA	x		x	x
scroll, message to MLINE	R2 is message code (note 1)	02E3	x	x	x	x
scroll, message to screen	R2 is message code (note 1)	0602	x	x	x	x
assemble data on screen	see 'assemble data', above	052F	x	x	x	x

Note 1: message code data:

04: 'PC='
08: 'AD='
0C: 'R='
10: 'BP1='
14: 'BP2='
18: 'BEG='
1C: 'END='
20: 'SAD='
24: 'FIL='
28: 'IIII'

17

An improved text routine

A complete alphabet

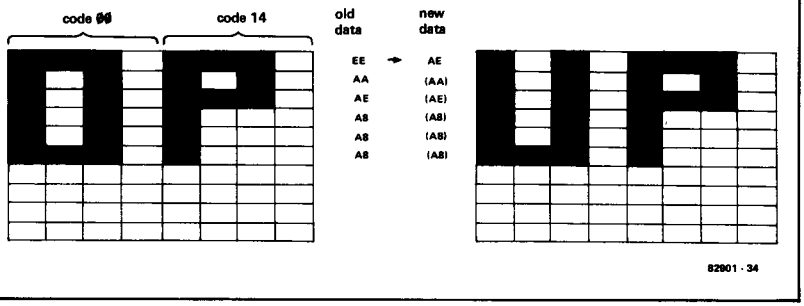
The monitor text routines, as described in chapters 11 and 16, have one drawback: they do not provide a complete alphabet. This can present quite a problem, when using them in a program: you have to work out a text that only uses existing letters. To get around this, several solutions have been incorporated in programs that we received. Some readers store the complete text as 'shape data' for the objects, in an extensive section of RAM. Others have copied the monitor routines, and extended the 'image data table' to include the missing characters. An even better solution is described in this chapter.

The monitor text routines were explained in the previous chapter. These can be summed up briefly as follows:

- each character is defined by a unique 'character code' – ØØ for a zero, ØA for the letter A, and 18 for a + sign, to name a few.
- for a single line of text, eight of these codes are first loaded to the 'monitor line', from Ø89Ø to Ø897.
- then, using the subroutine at Ø2ØE, the corresponding shape data bytes for the eight characters are retrieved from the 'image table' and stored from Ø878 to Ø88F (the data area for the bottom line).
- finally, the display routine retrieves this data and transfers it to the object shape bytes in the PVI as required for the actual display.

As far as the monitor is concerned, once the subroutine at Ø2ØE has been run the correct data is stored from Ø878 to Ø88F. When it comes to displaying the text on the screen, it will blindly load this data into the PVI. In other words, if the data is modified *after* running the 'translate MLINE' subroutine, the display routine will put the modified data on the screen!
This provides an easy way of extending the monitor character set. You select a character that is as similar as possible to the one you really want and load it in the normal way. Then: modify one or two bytes in the text display area to convert it into the desired shape! As an example, assume that you need a 'U'. This is very similar to 'O', so you start with the character code ØØ. Then, having run the Ø2ØE routine, the 'top' data byte is modified to delete the central bit. When doing this, you must of course ensure that the other character's data in the same byte remains unaltered. This is illustrated in figure 34.

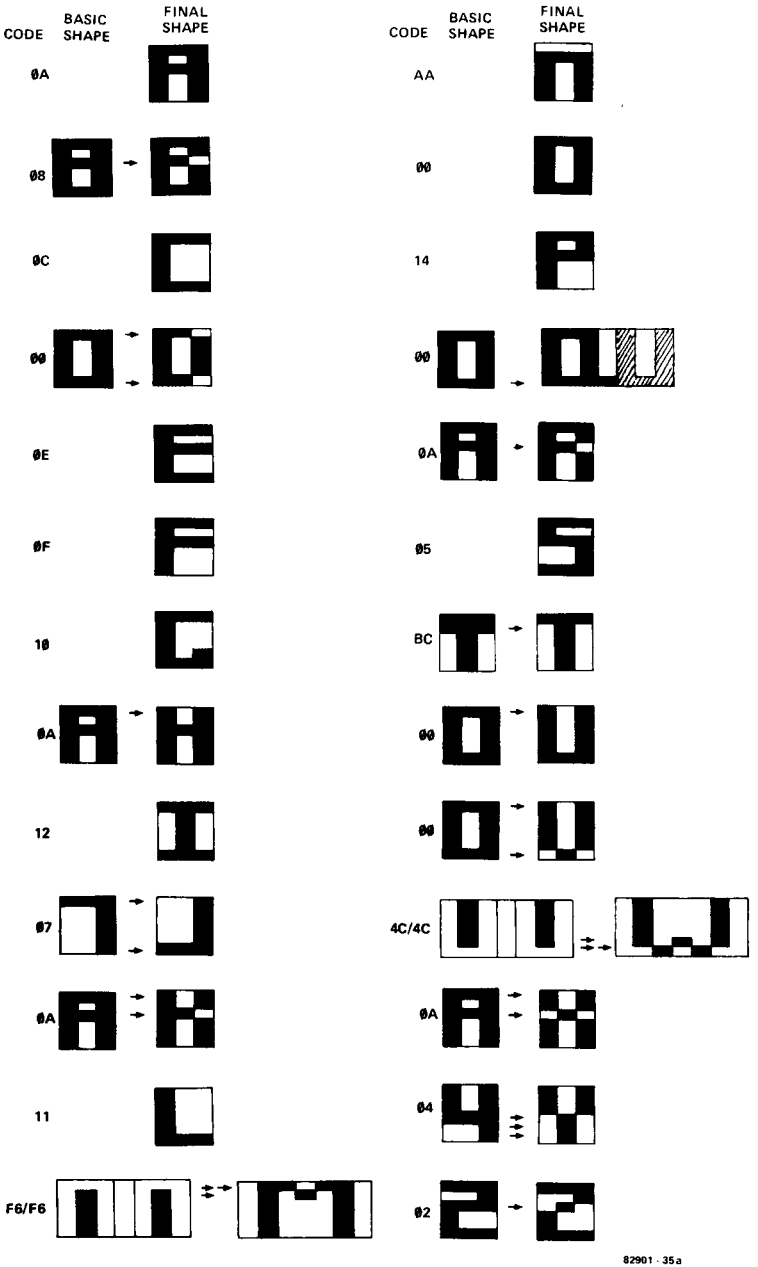
Figure 34



A complete set of conversions is given in figure 35, for capital letters and a few other odds and ends. The same system can be used, of course, to obtain any other desired shapes. In this connection, it is interesting to note that two (or more) character positions can be used for a single shape. In fact, this possibility is used for the 'M' and 'W' shapes.
Converting theory into practice is only a small step. To simplify matters, figure 36 clearly shows the address of each shape byte, immediately after running the Ø2ØE subroutine. Given a text, therefore, the necessary character codes and modifications can be derived from figure 35, and figure 36 shows where to enter the modified data. The only outstanding requirement is a suitable program subroutine.
While we're at it, we may as well go one step further. Table 40 gives a complete program, that illustrates the principles outlined above. At the same time, it shows how several texts can be selected and how a score result (or other data) can be included in the text display.

The program itself starts, as usual, at Ø9ØØ. The main loop consists of little more than a keyscan: the RCAS, WCAS, C and upper control keys correspond to texts 1...4, respectively. When one of these keys is operated, the subroutine at Ø925 is run. Initially R1 indicates which key was operated, and this information is used to select the corresponding 'first address' (from Ø919 on) and preset the subroutine at Ø921.
A fairly straightforward text routine is now executed, from Ø936 to Ø95Ø. The only peculiarity is the 'data grab': under certain conditions (which we will explain later) the subroutine at Ø963 is inserted at Ø949. This causes a data byte (the left-hand joystick, in this case) to be inserted in the two right-hand positions of the MLINE. Note that this only occurs when the text is first put on the screen; moving the joystick afterwards will not modify the display. The final step is to modify the text shape data, as required.
To see exactly how this system works, we can examine line 3 in text two in greater detail. The first step, at address Ø936, is to 'scroll and load 8 spaces to MLINE'. Then R2 is preset to 'F8', and the subroutine at Ø921 is executed. At this point, the instruction at Ø921 will read 'ØD29EØ' (for text

Figure 35


CODE
BASIC SHAPE
FINAL SHAPE
0A
08
0C
00
0E
0F
10
0A
12
07
0A
11
F6/F6
CODE
BASIC SHAPE
FINAL SHAPE
AA
00
14
00
0A
05
BC
00
00
4C/4C
0A
04
02
82901 - 35a

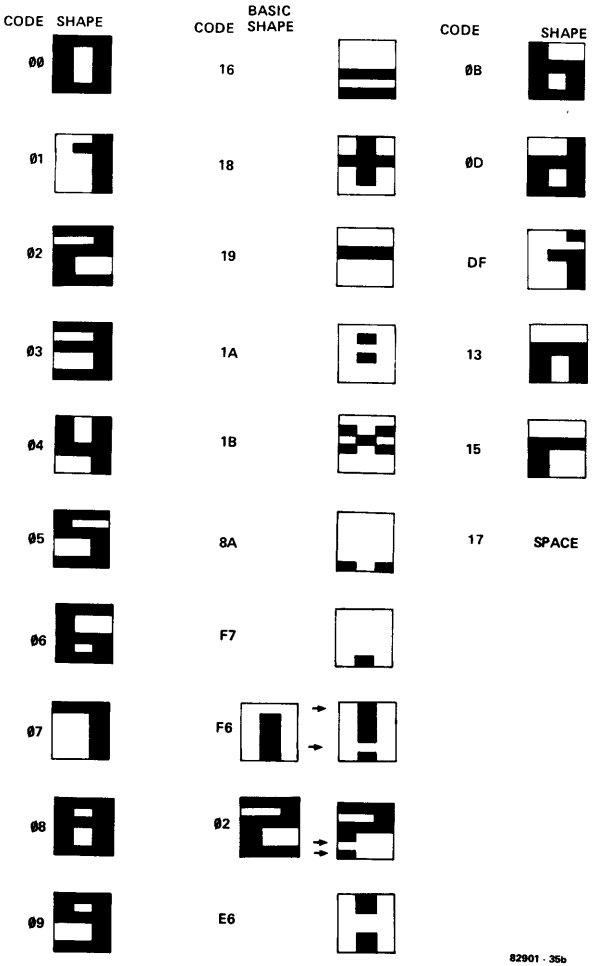

CODE SHAPE
CODE BASIC SHAPE
CODE SHAPE
ØØ
16
ØB
Ø1
18
ØD
Ø2
19
DF
Ø3
1A
13
Ø4
1B
15
Ø5
8A
17
SPACE
Ø6
F7
Ø7
F6
Ø8
Ø2
Ø9
E6
82901 - 35b

Figure 36

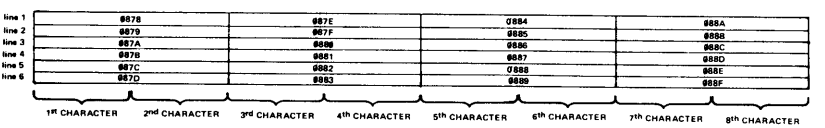

line 1
line 2
line 3
line 4
line 5
line 6
0878
0879
087A
087B
087C
087D
087E
087F
0880
0881
0882
0883
0884
0885
0886
0887
0888
0889
088A
088B
088C
088D
088E
088F
1st CHARACTER
2nd CHARACTER
3rd CHARACTER
4th CHARACTER
5th CHARACTER
6th CHARACTER
7th CHARACTER
8th CHARACTER

Table 40

```
0900    7620      PPSU,II
   2    7517      CPSL,RS
   4    0501      LODI,R1
   6    3B1D      BSTR,UN
   8    12        SPSU
   9    9A7D      BCFR
   B    0504      LODI,R1
   D    0D7E87    LODA,I/R1
0910    1A74      BCTR
   2    F979      BDRR,R1
   4    3F0055    BSTA,UN
   7    1B6F      BCTR,UN

0919    0976      first address, text 1
   B    09E0      first address, text 2
   D    0976
   F    0976

0921    0D2976    LODA,I+R1
   4    17        RETC,UN

0925    D1        RRL,R1
   6    0D4919    LODA,I-R1
   9    C878      STRR,R0
   B    0D4919    LODA,I-R1
   E    6420      IORI,R0
0930    C870      STRR,R0
   2    0706      LODI,R3
   4    05FF      LODI,R1
   6    7710      PPSL,RS
   8    3F02CF    BSTA,UN
   B    7510      CPSL,RS
   D    06F8      LODI,R2
   F    3B60      BSTR,UN
0941    CE6798    STRA,I/R2
   4    DA79      BIRR,R2
   6    3B59      BSTR,UN
   8    C2        STRZ,R2
   9    3A18      BSTR
   B    7712      PPSL,RS+COM
   D    3F020E    BSTA,UN
0950    7510      CPSL,RS
   2    02        LODZ,R2
   3    180B      BCTR
   5    3B4A      BSTR,UN
   7    C804      STRR,R0
   9    3B46      BSTR,UN
   B    CC0878    STRA,R0
   E    FA75      BDRR,R2
0960    FB54      BDRR,R3
   2    17        RETC,UN
```

presets

load new text

wait for VRST

keyscan

display text

'load data' subroutine

preset 'load data' subroutine

preset R3,R1

scroll

load MLINE

preset R2; data grab
if MSB set

translate MLINE to display

modify text shape data

(continued on next page! →

```
0963    467F        ANDI,R2          clear MSB
   5    C90D        STRR,R1          save R1
   7    0D1FCC      LODA,R0          load data
   A    3F035E      BSTA,UN  }
   D    CC0896      STRA,R0  }       split register, load to MLINE
0970    CD0897      STRA,R1  }
   3    0500        LODI,R1          restore R1
   5    17          RETC,UN

0976    0A 08 0C 00 0E 0F 10 0A  }
   E    04                       } line 1
   F    7A EC 7E EC 83 EC 8A EA  }
0987    12 07 0A 11 F6 F6 AA 00  }
   F    06                       } line 2
0990    78 E2 7D EE 7E A8 80 C8  }
   8    84 6C 85 54              }
   C    14 00 17 0A 05 BC 00 00  }
09A4    05                       } line 3   text 1
   5    7D 8F 80 0C 85 84 8A AA  }
   D    8F E4                    }
   F    4C 4C 0A 04 02 17 00 01  }
09B7    08                       } line 4
   8    7C 54 7D 28 7E AA 80 4A  }
09C0    81 A4 82 A4 83 A4 86 40  }
   8    02 03 04 05 06 07 08 09  } line 5
09D0    00                       }
   1    16 18 19 1A 1B F7 02 F6  }
   9    03                       } line 6
   A    8A E4 8E 00 8F 84        }

09E0    0B 0D DF F6 13 15 8A E6  }
   8    01                       } line 1
   9    7E 24                    }
   B    8A 8A 8A 8A 8A 8A 8A 8A  } line 2
09F3    00                       }
   4    00 0A BC 0A E6 17 17 17  }
   C    83                       } line 3   text 2
   D    78 CE 7D CA 7F 4A        }
0A03    8A 8A 8A 8A 8A 8A 8A 8A  } line 4
   B    00                       }
   C    1B 1B 1B 1B 1B 1B 1B 1B  }
0A14    04                       } line 5
   5    7A 55 80 55 86 55 8C 54  }
   D    93 93 93 93 93 93 93 93  }
0A25    04                       } line 6
   6    7A 11 80 11 86 11 8C 10  }
```

2) and R1 will have incremented to point to address 09F4: the first data byte for line 3. In the following loop, the first 8 bytes for this line are therefore transferred to the MLINE scratch from 0890 on – bearing in mind that 0798 + F8 = 0890.

At 0946 the subroutine at 0921 is called again, to retrieve the data byte at 09FC: '83'. This byte indicates two things: the '8' means that the MSB is set

so that a 'data grab' must be executed (note that this is the only line where this byte does not start with a 'Ø'!). The '3' corresponds to the number of shape bytes that must be modified for this line. After running the data grab routine, the MLINE codes are converted into shape data at Ø94D.
Finally, since R2 is not zero, the text shape data must be modified. The loop from Ø955 on will first retrieve the lower address byte '78' and preset the instruction at Ø95B accordingly, after which the data 'CE' is stored at the indicated address. This procedure is carried out three times, as indicated by R2, to store CE in Ø878, CA in Ø87D and 4A in Ø87F.
To experiment with this system, data for a third text can be stored from ØA2E on, after which 'ØA2E' can be stored at Ø91D. There are a few points to watch, however:

- The 'modify' loop from Ø955 on is unprotected, in the sense that any data value from ØØ to FF that is retrieved at Ø955 will be used for the address. This means that any address from Ø8ØØ to Ø8FF can be modified! All data retrieved at this point should be in the 78 to 8F range, to avoid problems. For this reason, great care must be taken to avoid programming errors, and the addresses given from Ø919 on must correspond to a correctly programmed area. This is why 'Ø976' was used for the two 'spare' text addresses.
- The subroutine assumes, at address Ø932, that there are six lines of text. If you want to try out fewer lines, the data for R3 must be reduced accordingly at this point.

It should be stressed that this program is given as an example. The two main tricks are the 'data grab' and 'modify text shape data' routines. In actual programs, all kinds of variations may prove useful. A text data block may be preceeded by an indication of the number of lines required; the 'data grab' results can be stored in any other position in MLINE, or this routine may even prove redundant; and so on. Furthermore, as we have discovered in practice, it is not a bad idea to insert a little check between the instructions at Ø955 and Ø957:

```
Ø955    3B4A      BSTR, UN
   7    E48F      COMI, RØ
   9    19Ø7      BCTR
   B    C8Ø4      STRR, RØ
```

This protects all data from Ø89Ø on!

18

TV games extended!

Another 3K of memory, and sound effects!

The basic TV games computer has nearly 2K of program memory. This has proved to be adequate for a lot of interesting games, without the programs getting completely out of hand – bearing Parkinson's Law in mind: 'Computer programs grow until all available memory space is filled'. However, it doesn't take dedicated TV games computer fans long to reach the point where they can put a greater memory range to good use. In this chapter, we will describe an extension board that almost triples the available memory. Also included on the board are two 'Programmable Sound Effects Generators' (PSGs).
The extension board contains one further interesting feature: provision was made for plugging in games cartridges that are intended for some of the commercial machines!

The best way to illustrate the new features is by means of a memory map as shown in figure 37. In the basic version, the monitor occupied the first 2K of memory and this was followed by 2K of RAM (1¾ K for the user). The address range from 1ØØØ was virtually unused – it only contained a few input/output lines and the PVI. Now, a further 3K RAM is added, from 1ØØØ to 1BFF, and two Programmable Sound Generators (PSGs) are located at addresses 1DØØ to 1D1F.
The area from 18ØØ on gets rather confused, owing in part to a minor error in the original monitor ROM. The Read Cassette address is located at 19BF, instead of 1DBF where it belongs. Since the addresses in this range were not fully decoded, this made no difference at the time. Now, it complicates matters. A facility must be added to disable the RAM at 199C...199F and 19BC...19BF. This is selected by setting S3 to position 'x' and S5 to position 'B'.
The total 'memory map' from 18ØØ on is shown in greater detail in the right-hand half of figure 37. First, the RAM with the two 'input' ranges mentioned above; then a gap; then the sound generators, at 1DØØ, followed by some input and output ranges (which include cassette read and write). At 1E88, the keyboard; and, finally, from 1FØØ on, the PVI.
Readers who feel so inclined can, of course, add other features. A hardware random numbers generator at 1D2Ø? Other custom-designed input/output devices in the 1Dxx range? A commercial-style explosion sound generator at 1E8Ø? They're all possible, under one proviso: they musn't be used in programs submitted to us for including in the ESS service!

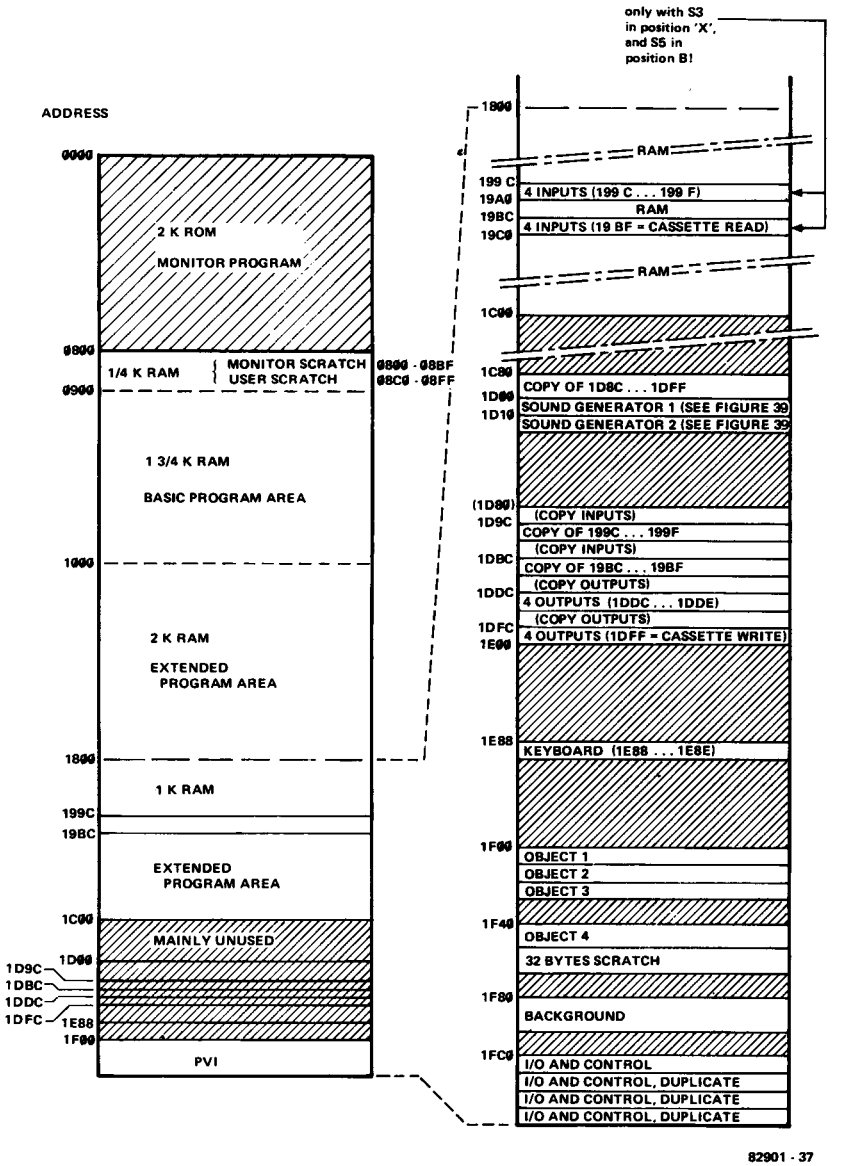


Figure 37. Memory map of the extended TV games computer. The final section, from 1800 to 1FFF, is shown in somewhat greater detail at the right.

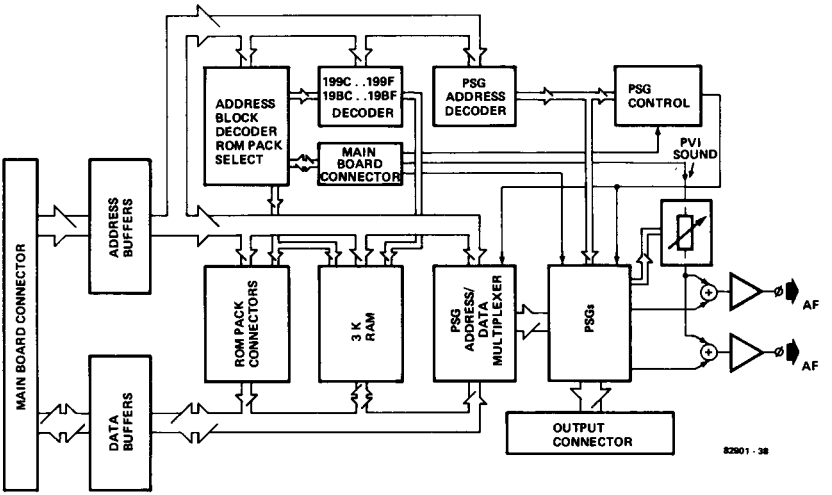


Figure 38. Block diagram of the extension board.

The extension board

A block diagram of the extension circuit is given in figure 38. It is fairly straightforward. To cater for the additional load, the address and data lines from the main board must be buffered. The actual extensions are shown in the lower half of the figure: ROM connectors, for commercial game cartridges; RAM extension; Programmable Sound Generators, with their associated audio outputs. Above these are the control circuits: the address block decoder, with switches for selecting commercial ROM game cartridges as required; the decoder for addresses corresponding to the (erroneous) input blocks; the PSG address decoder and control circuit. A further connector to the main board, drawn centrally in the figure, carries a few 'odd' signals like 'clock' and 'PVI audio output'.

The PSGs

The programmable Sound Generators are the most noteworthy part of the extension board. These ICs are quite complicated, as illustrated in the block diagram given in figure 39. Each IC contains 16 registers, corresponding to addresses 1DØØ...1DØF or 1D1Ø...1D1F, as shown. These registers are controlled by an address/data demultiplexer in the IC – unfortunately, in our case, since it means adding an external address/data multiplexer as shown in figure 38!

The data stored in the registers controls three tone generators, a noise generator and associated mixers; an envelope generator, with associated D/A converters; and two input/output ports. To start with the latter: in the normal case, each input/output port is 8 bits wide, and can be used at will – independently from the rest of the PSG, provided it is enabled by the

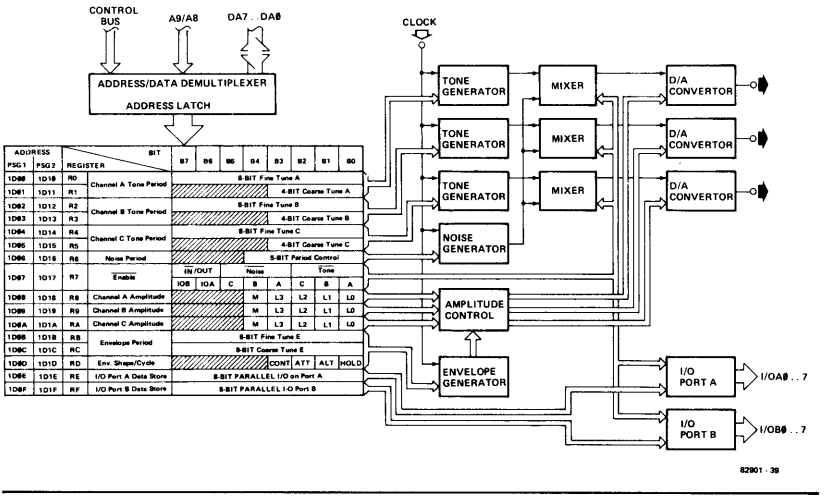


Figure 39. Block diagram of a Programmable Sound Generator IC. The 16 registers control three tone generators, a noise generator, envelope generator, mixers and output ports.

corresponding bit in R7. In the application described here, however, they can only be used as outputs. Furthermore, the four least significant bits of port A in PSG 1 (corresponding to address 1DØE) are used for amplitude control of the PVI sound output.

The basic frequency of each tone generator can be set anywhere in the whole audio range – the appendix includes a table for selecting any note in an eight-octave range. The basic noise generator 'frequency' can be determined. The desired outputs can be enabled individually. The output amplitude for each channel can be set, or else controlled by the 'envelope generator'; in either case, the result is determined by a 4-bit D/A converter, corresponding to 16 amplitude levels.

The envelope generator can be set for single-shot attack or decay effects, like explosions, or periodic amplitude effects like tremolo or machine-gun fire. It should be noted that disabling a tone or noise generator by means of R7 (address 1DØ7 or 1D17) is not sufficient to completely 'kill' it. For complete silence the 'amplitude level' of all outputs must be set to zero, by storing ØØ in registers R8...RA. This we have discovered after a lot of frustration...

All in all, a phenomenal range of sound effects are available. The original application note gives an 'explosion', 'gun shot', 'European siren', 'laser effect', 'whistling bomb', 'wolf whistle', and 'racing car'. In no time at all, we added a rattling chain and several polyphonic wedding marches. Great fun!

In the next chapter, we will go into much greater detail on these units. First, however, it is more important to see how the circuit works, and how it should be connected to the existing main board.

Circuit details

The complete circuit is given in figure 40. The main point to note is that it conforms to the block diagram given in figure 38! This is a great help when studying the operation.

Starting at the left, the large connector brings the address and data lines up from the main board. The address lines are buffered by IC1 and IC2 before going up to the address decoders at the top left. There, a few gates (N1, N2, N12 and N16) decode the block in which the PVI is located and pass the 'PVI SEL' signal back down via the connector. Immediately below these gates, IC15 does a more extensive decoding job. The lower half divides the available address range into four equal chunks:

- 0000...07FF. Switch S2 and gates N14/N8 and N10 determine what is to happen in this range: either the monitor ROM is selected, via the little connector, or else a plug-in ROM cartridge can be enabled. This will be discussed later.
- 0800...0FFF. In this case, switch S1 and gates N15/N9 and N11 determine the function: the RAM on the main board or a plug-in ROM cartridge. Note that S1 and S2 must be used in conjunction with S4 for ROM cartridges: this switch selects either of the two ranges mentioned above, or else the total range from 0000 to 0FFF.
- 1000...17FF. This area is further subdivided, by means of N13, N5 and N6, to enable IC4/IC5 or IC6/IC7. This range is therefore always available as RAM.
- 1800...1FFF. This is where confusion sets in, as illustrated in figure 37! The lower half of the range, from 1800 to 1BFF, is selected by N4. When S5 is in position A, this simply enables IC8/IC9 so that the complete range is available as RAM. In position B, however, two small sections of RAM (199C...199F and 19BC...19BF) are disabled via N18-N19-N20-N24-N26. This makes room for the 'cassette read' address, which is selected when S3 is set to position X.

  At this point, it is worth noting that S3 and S5 are not really needed, as switches. They can just as easily be hard-wired, depending on the monitor ROM (or EPROM) used. If the instruction at address 073D reads 0D19BF, switch S3 must always be set in position X and S5 in position B. If the instruction reads 0D1DBF, both switches can be set in the other position. In either case, two wire links will do the job just as well.
- 1C00...1FFF. For the upper half of this range, IC15 is blocked by the PVI, via N3 and pin 1 of the DIL connector. This leaves the range from 1C00 to 1DFF, which is subdivided by the second half of IC15. To be more precise, output 0 (pin 12) corresponds to two address blocks: 1C00...1C3F and 1D00...1D3F. In the same way, output 1 selects IC40...1C7F and so on; output 2 is for 1C80...1CBF etc. and output 3 corresponds to 1CC0...1CFF and 1DC0...1DFF. Each set of address blocks is further decoded. To start with an easy one:

– the highest blocks (1.C0...1.FF) simply go down to the main board, to enable the outputs. This incomplete address decoding leads to a profusion of 'copy output' addresses in the 1CC0...1CFF and 1DC0...

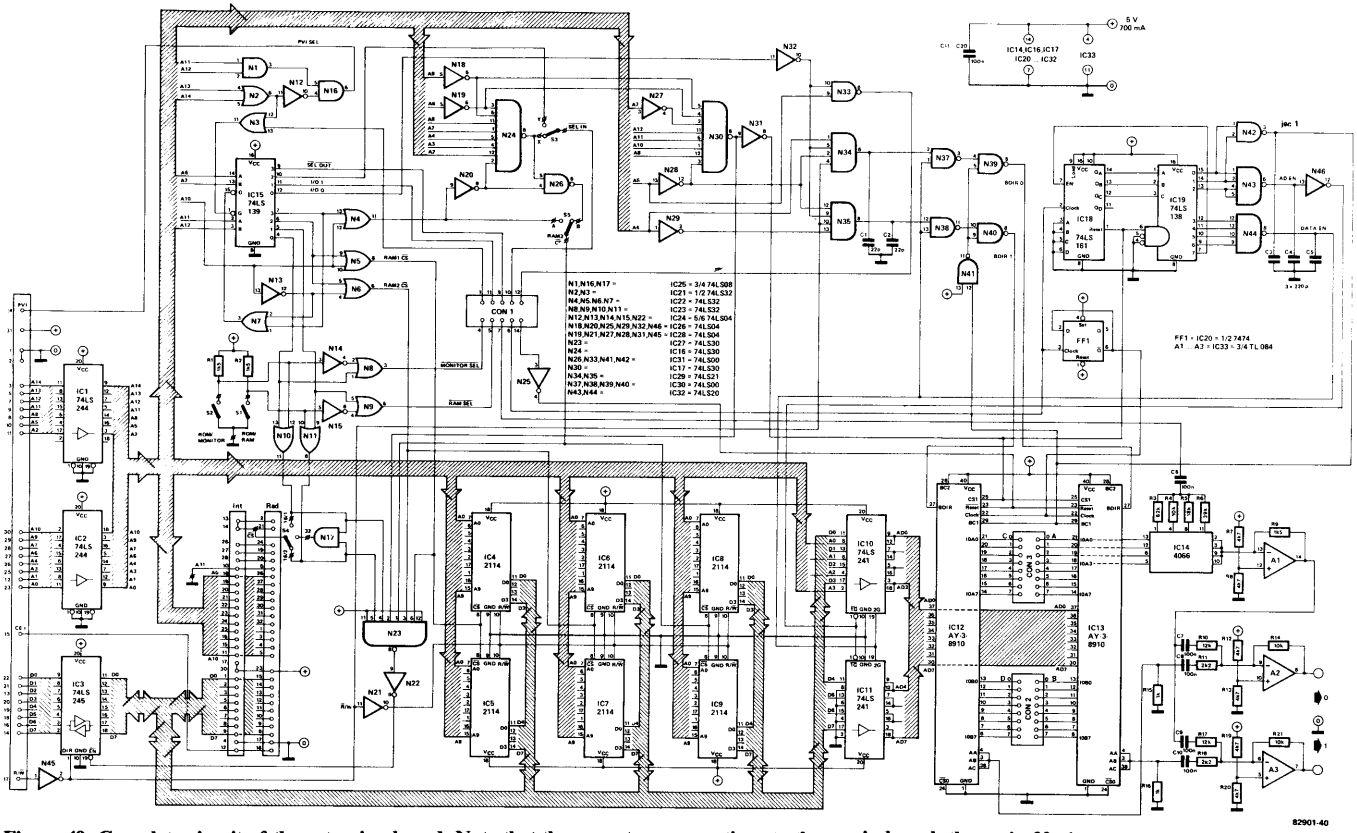


Figure 40. Complete circuit of the extension board. Note that there are two connections to the main board: the main 32-pin connector at the left, and a 14-pin connector in the centre of the drawing.

1DFF ranges, as can be seen in figure 37. However, since those addresses aren't used for anything else, who cares?

- a similar story applies for the blocks from 1.8Ø...1.BF. Depending on the setting of S3, these blocks are either unused (S3 in position X) or else they again lead to a profusion of input addresses in the 1C8Ø...1CBF and 1D8Ø...1DBF ranges.
- the next set, from 1.4Ø...1.7F, is the easiest of all. It doesn't do anything at all! In the basic version, this range was reserved for an input/output area ('I/O 1'). Readers who have made use of this will find that it still works; for everyone else, ourselves included, it is still a blank.
- finally, the lower ranges: 1.ØØ...1.3F. In the PSG address decoder (upper centre in figure 40), N30 selects the range from 1DØØ...1D1F. Via N31, it enables the two sound generator ICs (IC12 and IC13) and the PSG control circuit. N34 and N35 decide which of the two PSGs is actually to be used.

The PSG control circuit (upper right-hand corner) and PSG address/data multiplexer (IC10 and IC11) work hand-in-glove to drive the PSGs, IC12 and IC13. Exactly how this part of the circuit works is of academic interest. Suffice it to say that, when data is to be transferred to one of the PSGs, this lot makes sure that it arrives in the correct register... We can give a thumbnail description of the basic principles: as stated above, the output of N39 and N40 is logic 1 when the corresponding PSG must be enabled; the output of N42 (BC1) must be logic 1 to latch the address information into the PSG and logic Ø to load the data. This is achieved by feeding a high-frequency clock signal into IC18; together with IC19 (when the latter is enabled) this converts the clock signal into eight successive output pulses. During the first two N42 pulls BC1 high, and during the first three N43 and N46 cause IC10 and IC11 to select the address information for the PSGs. This is the 'latch address' step. After this, N44 switches IC10 and IC11 to the data select mode, loading the selected PSG register. The final step is a 'hold' mode, which is maintained until IC18 and IC19 are reset at the end of the current instruction cycle (that is, when the address data is no longer present).

Finally, the output from the PSGs. In fact, there are two distinct types, as illustrated in figure 39. The Input/Output ports, corresponding to addresses 1DØE, 1DØF, 1D1E and 1D1F are brought out to two connectors. Note that, in this circuit, these can only be used as outputs. Furthermore, the four lower bits of 1DØE are used to control IC14, which sets the audio output level for the PVI sound. Opamps A1...A3 combine this signal with the audio outputs from the two PSGs.

In effect, there are now two audio outputs: one from each PSG, with the PVI sound mixed into both. Although this offers the possibility of 'stereo' sound effects, we found it easier to combine the whole lot into a single audio output by means of two 4k7 mixer resistors.

Construction

The printed circuit board is shown in figure 41. This, in itself, is fairly straightforward. Figure 42, the wiring diagram, is more important. This shows the main board underneath and the extension board on top. As can be seen, there are two connections between the boards: the main 31-pin connector and a 14-pin DIL connector on the extension board that runs to a 16-pin version in the original IC6 position on the main board. This IC, a 74LS139, is now used on the extension board as IC15.

More precise details are given adjacent to figure 42. As can be seen, three copper tracks must be broken on the main board and IC6 is moved to the extension board. On the main board, three connections must be brought out to the 31-pin connector and three to the original IC6 socket. The actual connections between the boards can hardly miss.

A full complement of switches is shown in figure 42. As mentioned earlier, S3 and S5 may just as well be replaced by wire links, according to the monitor ROM used. S1, S2 and S4 must always be used in conjunction, so a three-deck four-way switch can be used instead. The four positions would then correspond to 'basic version', 'ROM cartridge Ø000...Ø7FF', 'ROM cartridge Ø800...ØFFF' and 'ROM cartridge Ø000...ØFFF'.

Parts list

Resistors:
R1,R2,R9 = 1k5
R3 = 82 k
R4,R14,R21 = 10 k
R5 = 18 k
R6 = 39 k
R7,R8,R12,R13,R19,R20 = 4k7
R10,R17 = 12 k
R11,R18 = 2k2
R15,R16 = 1 k

Capacitors:
C1,C2 = 22 p
C3 . . . C5 = 220 p
C6 . . . C26 = 100 n

Semiconductors:
IC1,IC2 = 74LS244
IC3 = 74LS245
IC4 . . . IC9 = 2114
IC10,IC11 = 74LS241
IC12,IC13 = AY-3-8910
IC14 = 4066
IC15 = 74LS139*
IC16,IC17 = 74LS30
IC18 = 74LS161
IC19 = 74LS138
IC20 = 74LS74
IC21 . . . IC23 = 74LS32
IC24,IC26,IC28 = 74LS04
IC25 = 74LS08
IC27 = 74LS30
IC29 = 74LS21
IC30,IC31 = 74LS00
IC32 = 74LS20
IC33 = TL 084
**removed from main board (see IC6)*

Switches:
S1,S2,S5 = SPDT
S3 = SPDT or wire link
S4 = single-pole three-way

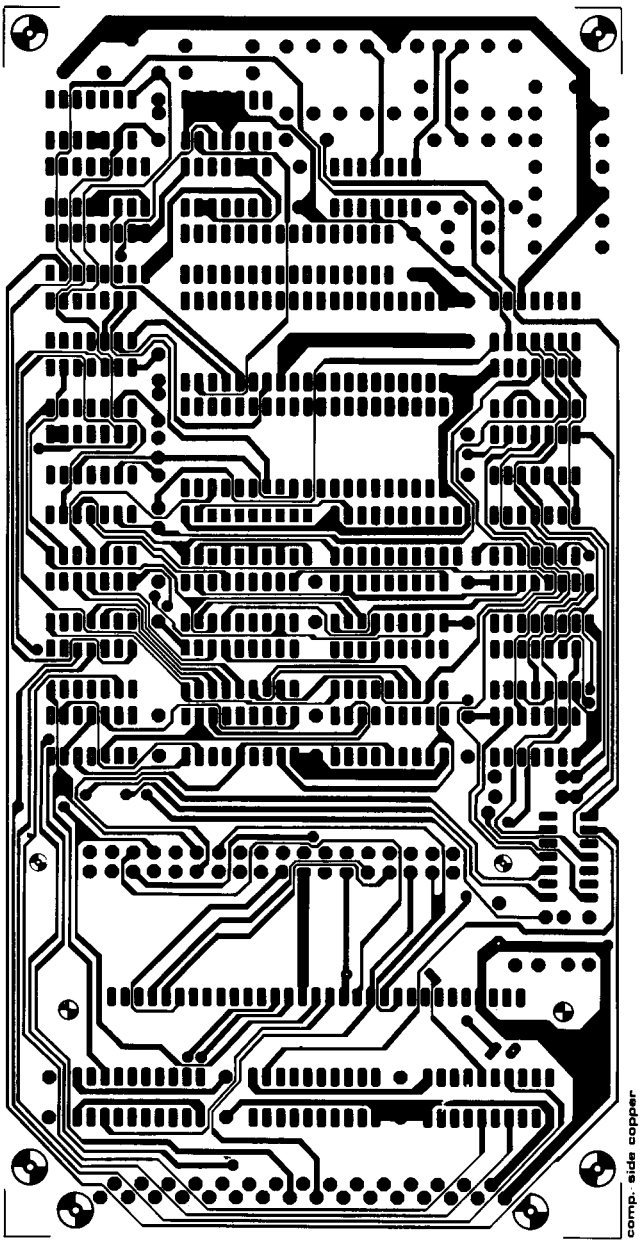


Figure 41. The printed circuit board and component layout. Note that it is shown here at 70% of its true size.

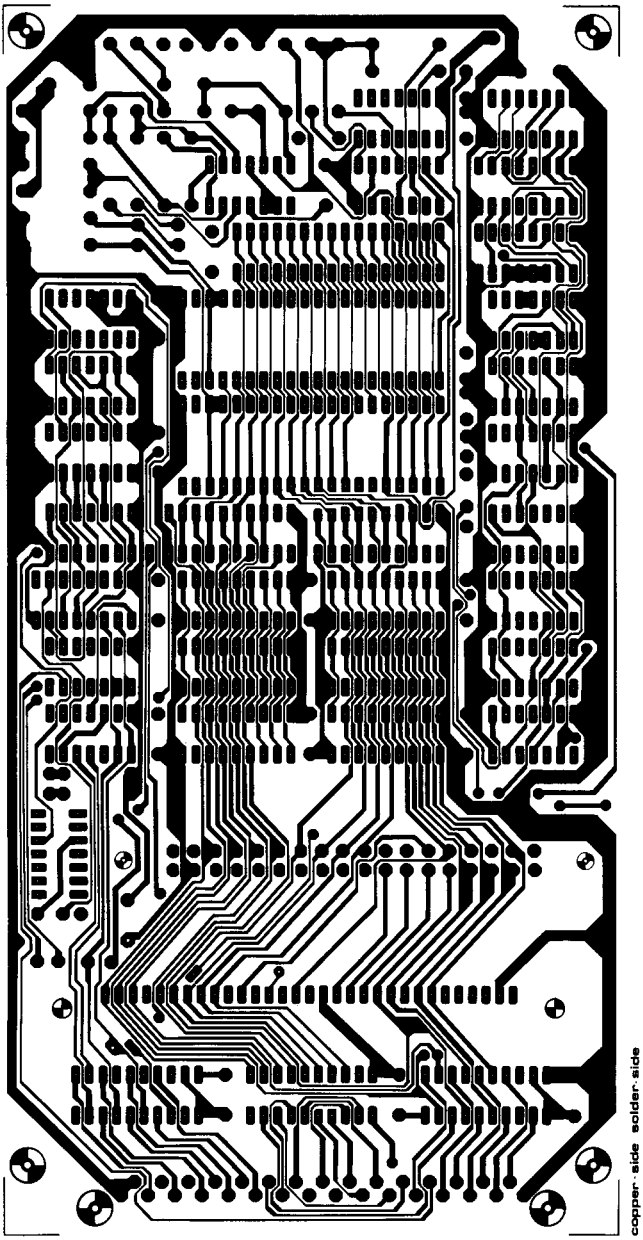


Figure 41. Continued.

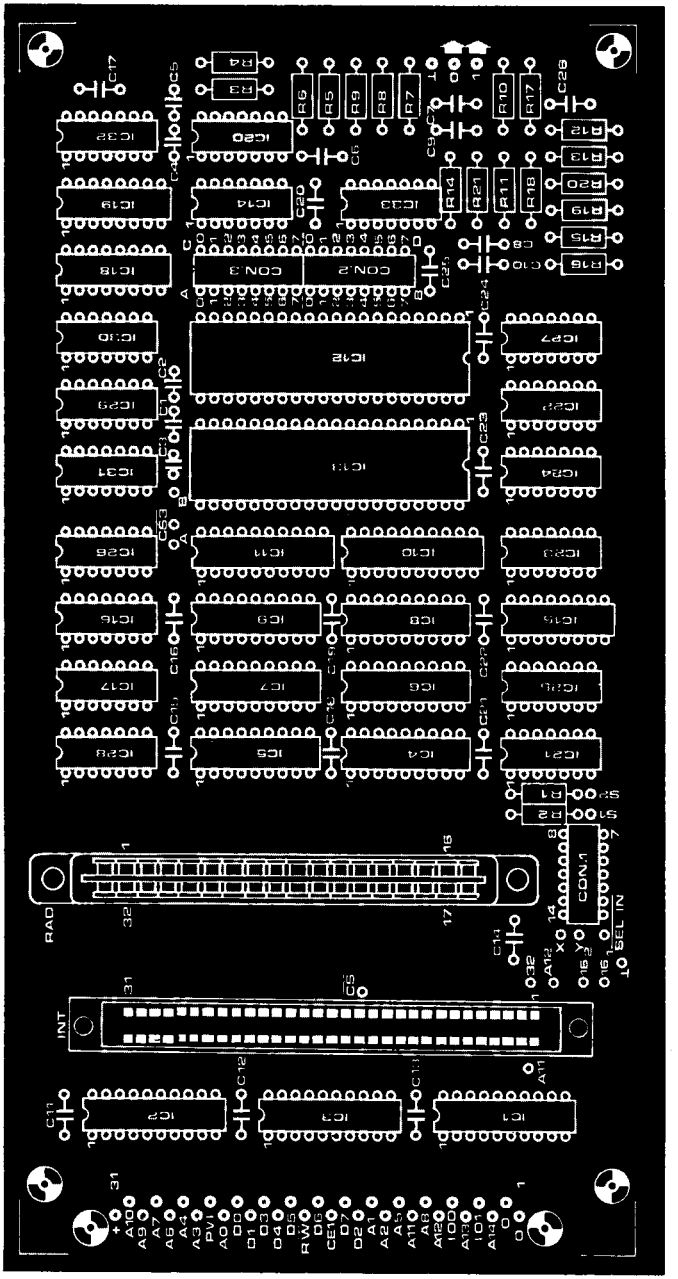


Figure 41. Continued.

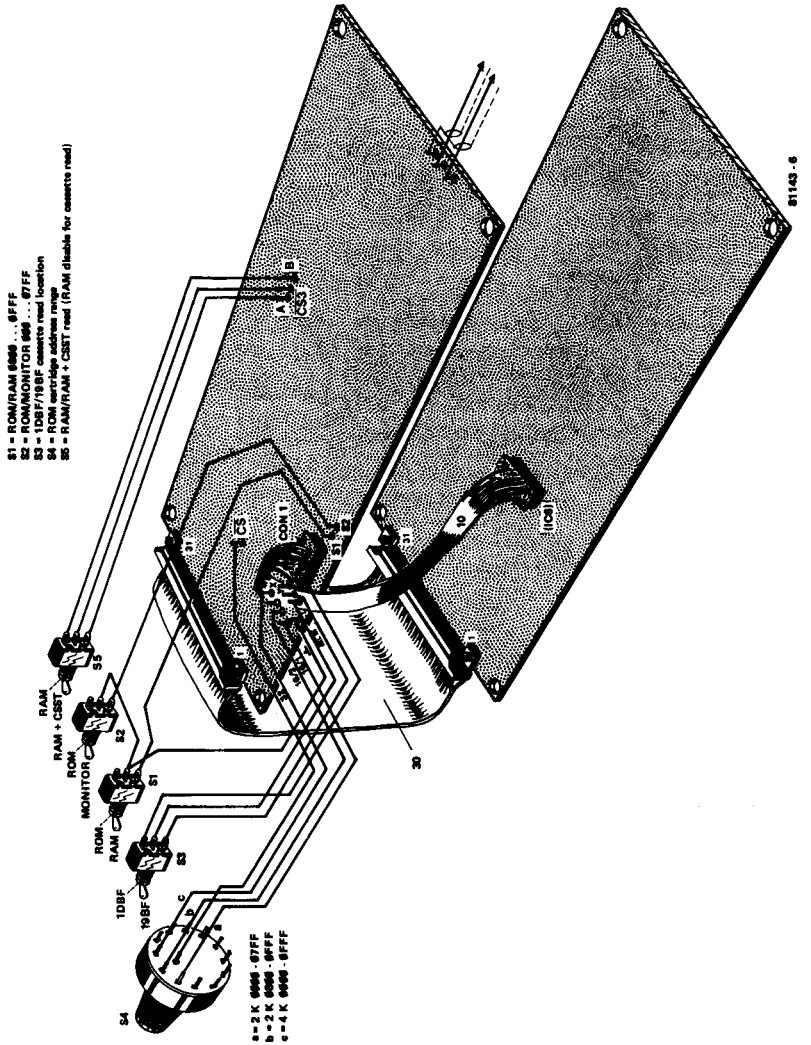


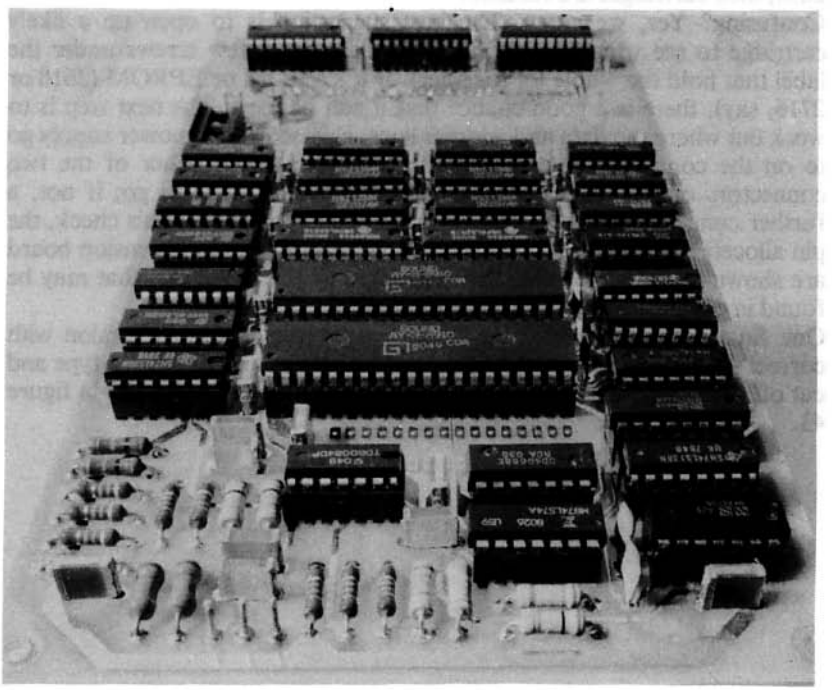
Figure 42. The wiring to the extension board is not quite as complicated as it looks. The various switches and connectors are clearly shown here, and the necessary operations on the main board are detailed on the next page.

Modifications to the main board:

1. Break the copper track leading to pin 15 of IC3 (address line 12 to PVI), at a point near this IC.
2. Break the copper track leading from pin 5 of the 31-pin connector to pin 6 of IC6 (EXP MEM to LS139) at a point near to the connector.
3. Break the copper track leading to pin 14 of IC6 (address line 6 to LS 139), on the component side of the board near IC6.
4. Remove IC6 (LS 139); it can be re-used as IC15 on the extension board.
5. Connect pin 18 of IC1 (address line 14 from the CPU) to pin 3 of the 31-pin connector.
6. Connect pin 19 of IC1 (address line 13 from the CPU) to pin 5 of the 31-pin connector.
7. Connect pin 15 of IC3 (PVI-SEL for the PVI) to pin 24 of the 31-pin connector.
8. Connect pin 12 of IC4 (CLK from the USG) to pin 6 of the IC6 socket.
9. Connect pin 22 of IC3 (audio out from the PVI) to pin 7 of the IC6 socket.
10. Connect the reset lead (R) to pin 14 of the IC6 socket.

Connections between the boards:

1. The two 31-pin connectors are simply wired pin 1 to pin 1, pin 2 to pin 2, etc. Note that pins 4 and 6 are not used.
2. A 16-pin DIL connector is inserted in the IC6 position on the main board and pins 1, 4...7, 9...12 and 14 are connected to the same pin numbers on CON 1 on the extension board. Note that the latter is a 14-pin connector, so pin 8 is opposite pin 7 and not adjacent to it!



A final word on the power supply. If this was built according to the description given earlier, there should be no problems. However, this extension board is a considerable extra load and so the fact that the supply was adequate for the basic version is no guarantee for the present situation. The main components to check are the bridge rectifier and the power transistor. Touch them with a moist finger: if it sizzles, they are running too hot! In that case, they need a larger heatsink, preferably mounted outside the case.

ROM cartridges

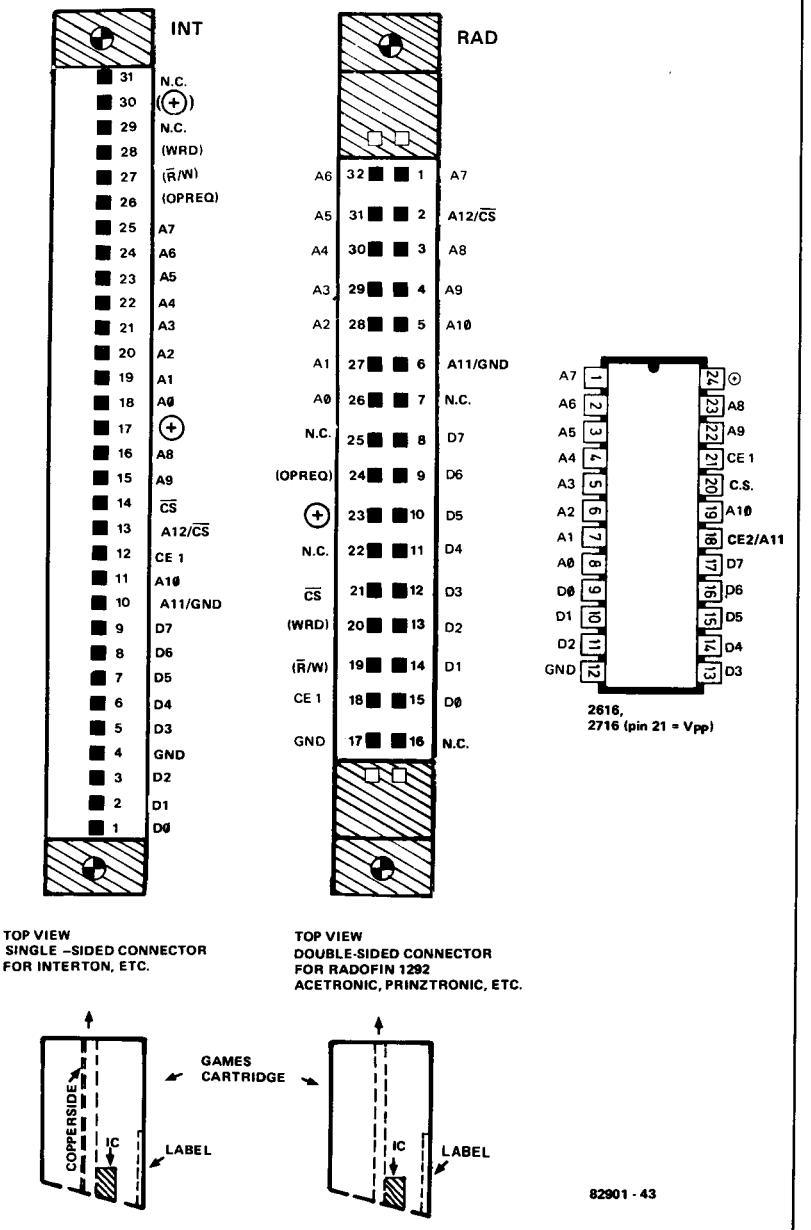
The extension board includes positions for two edge connectors, marked 'INT' and 'RAD' (for 'Interton' and 'Radofin', respectively). However, this is not to say that any Interton cartridge or any Radofin cartridge can be used; nor, on the other hand, does it limit the range of possible cartridges to these manufacturers. Things are rather more complicated than that.

In principle, any game cartridge that is intended for use with a 2650-based system can be used. Both manufacturers mentioned above do market a game of this type, and therefore they also supply game cartridges for it. However, they also have other TV games units that have nothing to do with the 2650; cartridges for these machines are useless. On the other hand, there are other manufacturers who also market systems that do use the 2650; their cartridges are suitable.

Confusing? Yes, we agree. The best suggestion is to open up a likely cartridge to see what's inside (usually, there are a few screws under the label that hold the whole lot together). If it's a ROM or EPROM (2616 or 2716, say), there is a good chance that it can be used. The next step is to work out where the data and address lines, chip select and power supply go to on the connector, and check whether these match either of the two connectors on the extension board. If so, you're ready to go; if not, a further connector can be wired up as required. To simplify this check, the pin allocations of the 'INT' and 'RAD' connectors on the extension board are shown in figure 43, together with some of the ROM ICs that may be found in game cartridges.

One final note, concerning the 'RAD' connector. A 32-pin version with correct spacing is not easy to obtain, and so we have used a 36-pin type and cut off the ends. This is illustrated by the dotted line and shading in figure 43.

Figure 43


INT
31 N.C.
30 ⊕
29 N.C.
28 (WRD)
27 (R/W)
26 (OPREQ)
25 A7
24 A6
23 A5
22 A4
21 A3
20 A2
19 A1
18 AØ
17 ⊕
16 A8
15 A9
14 CS
13 A12/CS
12 CE 1
11 A1Ø
10 A11/GND
9 D7
8 D6
7 D5
6 D4
5 D3
4 GND
3 D2
2 D1
1 DØ
RAD
A6 32 1 A7
A5 31 2 A12/CS
A4 30 3 A8
A3 29 4 A9
A2 28 5 A1Ø
A1 27 6 A11/GND
AØ 26 7 N.C.
N.C. 25 8 D7
(OPREQ) 24 9 D6
⊕ 23 10 D5
N.C. 22 11 D4
CS 21 12 D3
(WRD) 20 13 D2
(R/W) 19 14 D1
CE 1 18 15 DØ
GND 17 16 N.C.
A7 1 24 ⊙
A6 2 23 A8
A5 3 22 A9
A4 4 21 CE 1
A3 5 20 C.S.
A2 6 19 A1Ø
A1 7 18 CE2/A11
AØ 8 17 D7
DØ 9 16 D6
D1 10 15 D5
D2 11 14 D4
GND 12 13 D3
2616,
2716 (pin 21 = Vpp)
TOP VIEW
SINGLE –SIDED CONNECTOR
FOR INTERTON, ETC.
TOP VIEW
DOUBLE-SIDED CONNECTOR
FOR RADOFIN 1292
ACETRONIC, PRINZTRONIC, ETC.
COPPERSIDE
IC
LABEL
GAMES
CARTRIDGE
IC
LABEL
82901 - 43

19

Programmable sound generators

Organ, explosion or wolf whistle...

The most intriguing feature on the extension board is the extremely versatile sound effects facility offered by the two PSGs. Each of these units is almost a miniature electronic music synthesizer in its own right. Even quite simple subroutines can produce the most varied effects, as will be demonstrated in this chapter.
However, to put these units to their fullest use, it is essential first to have a fair idea of how they work.

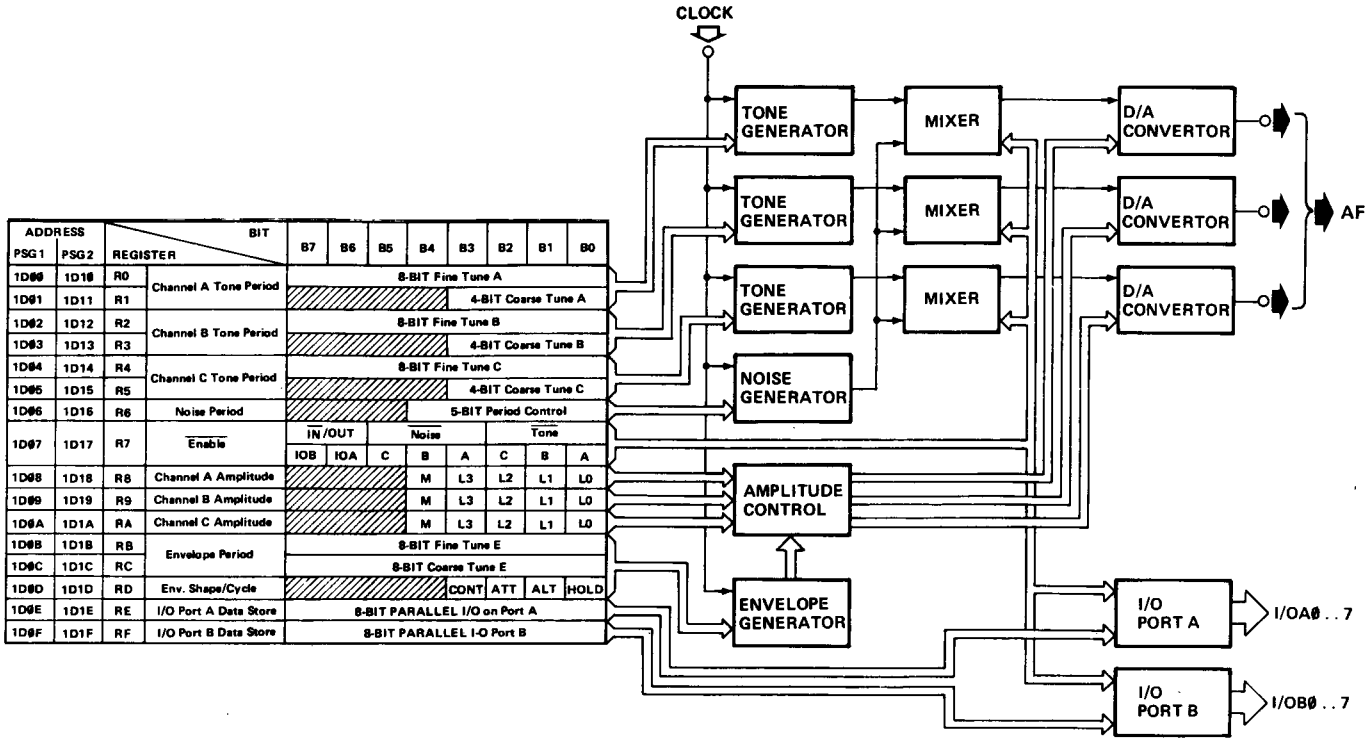
Since the complete hardware has been taken care of, we can deal with the two PSGs as 'units, located in the address block from 1DØØ...1D1F'. In other words, we can go straight into the question of what is the 'effect' of each address on the final sound. After that, we can demonstrate how certain specific sound effects can be achieved. The block diagram of one PSG (figure 44) will prove a useful guide.

- **The tone generators.** The data in the first six addresses control the frequency of the tone generators. For each of the three generators, two locations (8-bit fine tune and 4-bit coarse tune) define a 12-bit 'tone period value', as illustrated in figure 45. The lowest value, ØØØ, turns the tone generator off; ØØ1 gives the highest frequency and 111 the lowest. This is just what we were used to when programming the PVI for sound. The actual frequency obtained is equal to:

$$f_O = \frac{110837}{TP} \text{ Hz}$$

where TP is the *decimal* equivalent of the 12-bit tone period value. In the Appendix, one of the Tables gives a complete set of data values for an equal tempered chromatic scale over a range of eight octaves. To give one example: ØØ in the 'coarse' register and 64 in the 'fine' register gives a tone period value Ø64. The decimal equivalent for this is 100 (which is why it was chosen as an example!) and so the output frequency must be 110837 ÷ 100 = 1108.37Hz. In the Appendix, this will be found as C# in octave 6.

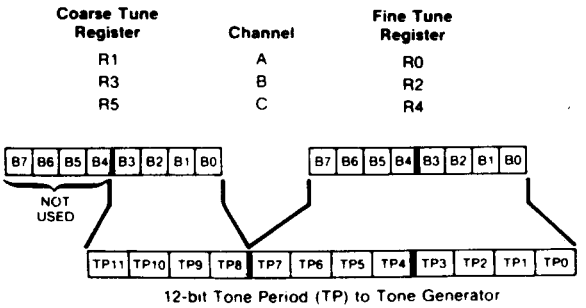
- **The noise generator.** The basic period time for the noise generator is determined in exactly the same way as described above. The only difference is that a 5-bit value (at address 1DØ6 or 1D16) is used. For this reason, the 'noise frequency range' is approximately 3.5kHz to 110kHz. The noise bandwidth can therefore be set between approximately 1.5kHz and 50kHz – a good range, in practice.



ADDRESS PSG1	ADDRESS PSG2	REGISTER	BIT	B7	B6	B5	B4	B3	B2	B1	B0
1D00	1D10	R0	Channel A Tone Period	8-BIT Fine Tune A							
1D01	1D11	R1						4-BIT Coarse Tune A			
1D02	1D12	R2	Channel B Tone Period	8-BIT Fine Tune B							
1D03	1D13	R3						4-BIT Coarse Tune B			
1D04	1D14	R4	Channel C Tone Period	8-BIT Fine Tune C							
1D05	1D15	R5						4-BIT Coarse Tune C			
1D06	1D16	R6	Noise Period				5-BIT Period Control				
1D07	1D17	R7	Enable	IN/OUT		Noise			Tone		
				IOB	IOA	C	B	A	C	B	A
1D08	1D18	R8	Channel A Amplitude				M	L3	L2	L1	L0
1D09	1D19	R9	Channel B Amplitude				M	L3	L2	L1	L0
1D0A	1D1A	RA	Channel C Amplitude				M	L3	L2	L1	L0
1D0B	1D1B	RB	Envelope Period	8-BIT Fine Tune E							
1D0C	1D1C	RC		8-BIT Coarse Tune E							
1D0D	1D1D	RD	Env. Shape/Cycle					CONT	ATT	ALT	HOLD
1D0E	1D1E	RE	I/O Port A Data Store	8-BIT PARALLEL I/O on Port A							
1D0F	1D1F	RF	I/O Port B Data Store	8-BIT PARALLEL I-O Port B							

Figure 44. Block diagram of a Programmable Sound Generator IC. The 16 registers control three tone generators, a noise generator, envelope generator, mixers and output ports.

Figure 45



• **Enable outputs.** These locations (1DØ7 and 1D17) select the signals (tone and/or noise) for each of the three output channels of each PSG. For instance, when the right-hand bit is logic Ø tone generator A is connected to output A (see figure 44). Note that disabling tone and noise does not turn off a channel: this can only be accomplished by writing ØØ into the corresponding amplitude control register (1DØ8...1DØA etc.). The two left-hand bits in the output control registers determine the function of the I/O ports: logic 1 for output, logic Ø to disable (since there is no 'read' facility in the extension board circuit!).

• **Amplitude control.** These locations (1DØ8...1DØA and 1D18...1D1A) determine the output amplitude for each channel. There are two possible modes, as selected by the 'M' bit:

– M = Ø: the amplitude is determined by the value of the four right-hand bits, corresponding to sixteen (logarithmic) level steps; thus, ØØ in one of these registers turns the corresponding channel off.
– M = 1: the remaining bits are now ignored, and the channel output amplitude is determined by the envelope generator.

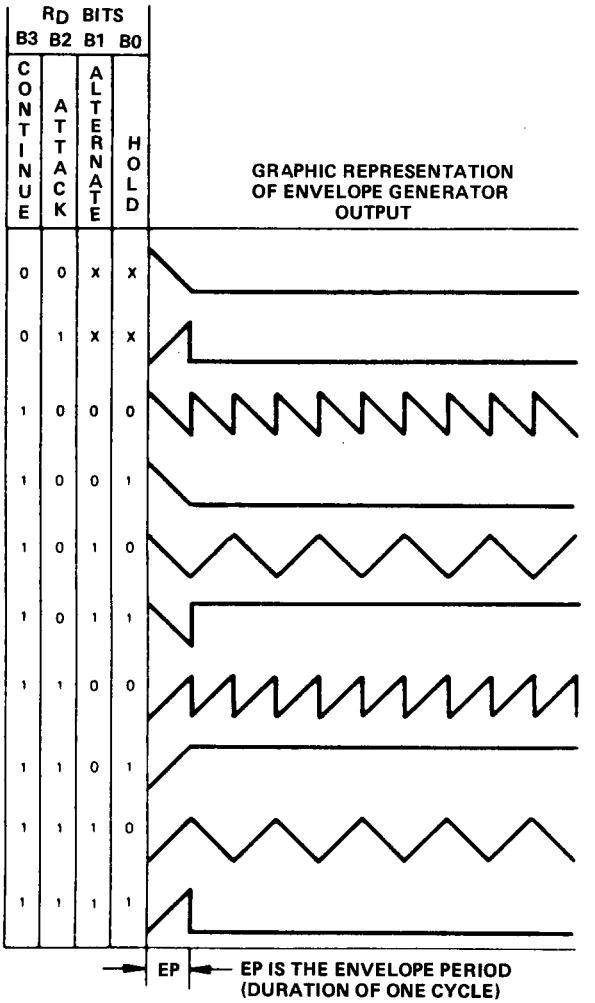
• **The envelope generator.** Two registers, corresponding to addresses 1DØB and 1DØC (or 1D1B, 1D1C) determine the envelope period; a third (1DØD or 1D1D) selects the envelope shape, as shown in figure 46. The envelope period is determined by the 16-bit total value that is given in the 'coarse' and 'fine' registers. As before, the shortest period value is ØØØ1 and the longest is FFFF. The actual value is again obtained from the decimal equivalent of the envelope period value (EP), as follows:

period = EP x 145 μs

For a one-second period time, for example, the decimal equivalent of the period value is approximately 6900, giving 1A for the coarse register and F4 for 'fine'. Another way of looking at this is to say that the envelope *frequency* is:

$$f = \frac{6900}{EP}$$

Figure 46



It should be noted that these values are all approximations – but good enough for all practical purposes. Perfectionists can calculate the period value as a multiple of 144.356 μ s, and the frequency as 6927.3 ÷ EP (Hz). The function of the envelope shape/cycle bits is best derived from figure 46.

• **I/O ports.** In this application, these locations (1DØE, 1DØF, 1D1E, 1D1F) are output ports only. As far as TV games are concerned, they will all remain unused – with one exception: 1DØE. When this port is enabled

as output (storing 40, say, in 1D07), its four right-hand bits determine the output level of the PVI sound: 0F for maximum level, 00 for sound off, or any in-between value. Alternatively, disabling this output port (00 in 1D07) will automatically set the PVI sound to maximum level. This is actually how the PVI sound gets through when running programs that were intended for the basic version of the TV games computer: at 'reset', 00 is written into all PSG registers.

Making noises

The best way to find out how the PSGs work is to try some actual sound effects. For 'dynamic' effects, the data in the PSGs must be varied at regular intervals. 'Explosions' and the like are often simpler, since it will suffice to store the basic data for the PSGs in one go. Several examples of both kinds of sound effect are given in Tables 41...44.

For the *siren* (Table 41), the preset routine from 0915 selects 'tone only, channel A' and sets this channel to maximum level. Then, after loading initial values into the registers, control is passed to the interrupt routine. Here, R0 is used to count off 17 frames (340ms); at the end of each count, the two EOR instructions switch the data in R2 and R1 from 01/56 to 00/FE or back. This data is used to set the frequency.

The *gunshot/explosion* (Table 42) is a noise-only effect. In this case, the initial presets select noise only on all three channels, under control of the envelope generator; the envelope period time is set to a suitable value. Each time the start key is operated, 00 is loaded into 1D0D. As can be seen in figure 46, this initiates a single decaying pulse: Bang!

The *wolf whistle* (Table 43) is an example of a dynamic effect using both noise and tone. The noise period is set to minimum; tone is selected, at maximum level, on channel A and noise at a somewhat lower level on channel B. After this, the first loop sweeps the tone period value from 40 down to 20; there is a brief delay; a further loop sweeps the period from 40 down to 30; finally, the last loop sweeps the period rapidly up to 68.

The *wedding march* (Table 44) is a much more extensive program. After a few presets, the main loop starts at 0912. At VRLE, the value for R0 at 0918 is decremented; when it reaches zero, data is retrieved from the 0A00 area as indicated by the (incrementing) value in R1. There are now several possibilities, depending on the left-hand bits of the retrieved data. The various 'branch if negative' (MSB set!) and 'rotate left' instructions select one of the following:

- data is 80: stop. The program jumps from 0921 via 092F and 0930 to 0943. At that point, the value in R1 is decremented by 2, causing a 'back step' in the data area.
- data is C0: overflow. In this case, the jumps are from 0921 via 092F...0933 to 0937. The various 'LODA, I+R1' instructions are incremented from 0D2A00 to 0D2B00, to point to the next section of the data area; R1 is incremented to FF and the program jumps back to 091E for the next data fetch.
- data is E0: delete. Via 0921 and 092F...0935, the program simply increments R1 and returns to 091E for the next data fetch. Undesired

Table 41: siren.

```
0900    1F0915    BSTA,UN     branch to preset routine
0903    B480      TPSU        } vertical interrupts only
   5    36        RETE        }
   6    F80C      BDRR,R0
   8    25A8      EORI,R1     }
   A    CD1D00    STRA,R1     } switch
   D    2601      EORI,R2     } tones
   F    CE1D01    STRA,R2     }
0912    0411      LODI,R0     set R0 for delay
   4    37        RETE,UN

0915    7620      PPSU,II
   7    04FE      LODI,R0     }
   9    CC1D07    STRA,R0     }
   C    040F      LODI,R0     }
   E    CC1D08    STRA,R0     } preset PSG
0921    05FE      LODI,R1     } and registers
   3    0600      LODI,R2     }
   5    0401      LODI,R0     }
   7    7420      CPSU,II     }
   9    1B7E      BCTR,UN     wait loop
```

Table 42: gunshot/explosion.

```
0900    7620      PPSU,II
   2    040F      LODI,R0     }
   4    CC1D06    STRA,R0     }
   7    0407      LODI,R0     }
   9    CC1D07    STRA,R0     }
   C    0410      LODI,R0     } PSG presets
   E    CC1D08    STRA,R0     }
0911    CC1D09    STRA,R0     }
   4    CC1D0A    STRA,R0     }
   7    0410      LODI,R0     }
   9    CC1D0C    STRA,R0     }
   C    0C1E8B    LODA,R0     }
   F    D0        RRL,R0      }
0920    9A7A      BCFR        }
   2    20        EORZ,R0     }
   3    CC1D0D    STRA,R0     } fire when start
   6    08F5      LODR,R0, ind. } key operated
   8    D0        RRL,R0      }
   9    1A7B      BCTR        }
   B    1B6F      BCTR,UN     }
```

For explosion sound, modify data at address 0902 to '041F' and at 0917 to '0438'.

Table 43: wolf whistle.

```
0900   7620    PPSU,II
   2   0401    LODI,R0
   4   CC1D06  STRA,R0
   7   042E    LODI,R0
   9   CC1D07  STRA,R0
   C   040F    LODI,R0
   E   CC1D08  STRA,R0
0911   0409    LODI,R0
   3   CC1D09  STRA,R0
   6   0440    LODI,R0
   8 ┌►CC1D00  STRA,R0
   B►  073C    LODI,R3
   D   3F06F7  BSTA,UN
0920   F800    BDRR,R0
   2   E420    COMI,R0
   4 └─9872    BCFR
   6   0507    LODI,R1
   8 ┌►0C1FCB  LODA,R0
   B   D0      RRL,R0
   C   9A7A    BCFR
   E └─F978    BDRR,R1
0930   0440    LODI,R0
   2 ┌►CC1D00  STRA,R0
   5   077D    LODI,R3
   7   3BE5    BSTR,ind
   9   F800    BDRR,R0
   B   E430    COMI,R0
   D └─9873    BCFR
   F ┌►CC1D00  STRA,R0
0942   071E    LODI,R3
   4   3BD8    BSTR,ind
   6   D800    BIRR,R0
   8   E468    COMI,R0
   A └─9873    BCFR
   C   20      EORZ,R0
   D   CC1D08  STRA,R0
0950   CC1D09  STRA,R0
   3 ┌►0C1E8B  LODA,R0
   6 │ D0      RRL,R0
   7 └─9A7A    BCFR
   9   1F090C  BCTA,UN
```

PSG presets

12 ms delay

140 ms delay

25 ms delay

6 ms delay

set amplitude to zero for both channels

repeat if 'start' key

data in the 0A00 area can be 'deleted' in this way: it is replaced by 'E0 00'.

- data is 40: set duration. In this case, the program runs on through 0921...0925 and then jumps to 0945. At that point, the next data byte is retrieved and then stored in 0919. This sets the next count value: nothing more will happen until after the indicated number of frames.
- data is less than 40: PSG data. The program simply runs from 0921 to 092D: the second data byte of each pair is stored at the PSG address indicated by the first. Thus, the first data line, from 0A00 on, will cause B7 to be loaded into 1D00; 03 into 1D01; etc.

Table 44: wedding march.

```
0900    7620      PPSU,II
   2    04F8      LODI,R0
   4    CC1D07    STRA,R0
   7    CC1D17    STRA,R0
090A    04FF      LODI,R0
   C    C80F      STRR,R0
   E    0401      LODI,R0
0910    C807      STRR,R0
0912    0C1FCB    LODA,R0
   5    D0        RRL,R0
   6    9A7A      BCFR
   8    0401      LODI,R0
   A    F82E      BDRR,R0
   C    05FF      LODI,R1
   E    0D2A00    LODA,I+R1
0921    1A0C      BCTR
   3    C2        STRZ,R2
   4    D0        RRL,R0
   5    1A1E      BCTR
   7    0D2A00    LODA,I+R1
   A    CE7D00    STRA,I/R2
   D    1B6F      BCTR,UN
   F    D0        RRL,R0
0930    9A11      BCFR
   2    D0        RRL,R0
   3    9A02      BCFR
   5    D967      BIRR,R1
   7    086F      LODR,R0
   9    8401      ADDI,R0
   B    C862      STRR,R0
   D    C869      STRR,R0
   F    C805      STRR,R0
0941    D95B      BIRR,R1
   3    A502      SUBI,R1
   5    0D2A00    LODA,I+R1
   8    C953      STRR,R1
   A    C84D      STRR,R0
   C    0C1E8B    LODA,R0
   F    D0        RRL,R0
0950    9E0912    BCFA
   3    08F7      LODR,R0 ind
   5    D0        RRL,R0
   6    1A7B      BCTR
   8    1F090A    BCTA,UN
```

presets

wait for VRLE

80 = stop, C0 = overflow, E0 = delete

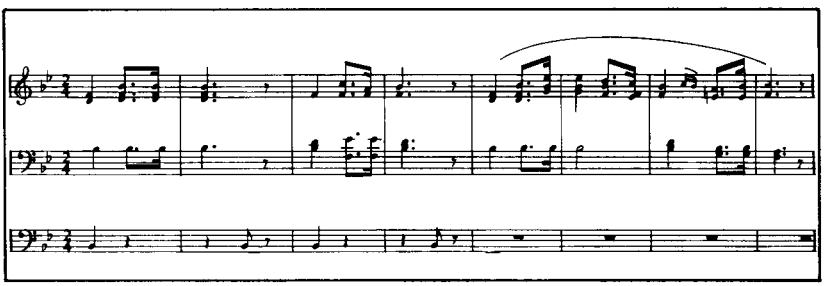
repeat if 'start' key operated

```
0A00    00B7 0103 023D 0301 0479 0501 10DC 1101
  10    0808 0908 0A08 1808 401F 07FF 4001 07F8
  20    00EE 0100 4017 07FF 4001 07F8 4007 07FF
  30    4001 07F8 4020 12B7 1303 1908 4010 0800
  40    0900 0A00 1800 1900 4010 0908 0A08 1808
  50    1908 4020 00D4 0464 0808 107B 1102 1900
  60    4017 07FF 4001 07F8 00FC 4007 07FF 4001
```

Continued on next page. ⟶

```
70      07F8 00EE 0479 10DC 1101 4020 1908 4010
80      0800 0900 0A00 1800 1900 4010 0908 0A08
90      1808 4020 0808 4018 021B 04B2 0500 4007
A0      07FF 4001 07F8 4020 04BD 023D 4018 00D4
B0      0464 0501 4008 00EE 0479 4016 00D4 4005
C0      00EE 4005 00FC 0250 0435 0502 4017 07FF
D0      4001 07F8 00EE 4008 00D4 023D 047B 10F8
E0      4030 0800 0900 0A00 1800 80
```

Notes:
- the first byte in each data group specifies the PSG register (e.g. 00 = 1D00) and the second is the corresponding data.
- alternatively, if the first byte is 40, the second byte determines the duration of the preset tone(s).
- the tune ends when the first byte is 80.
- to delete a group, replace it with E000.
- when the address range 'overflows', from 0AFF to 0B00 say, the last group in the current range should be (0AFE =) C000.
- the data values for the tone generator registers can be derived from the Table in the Appendix.



Adding sound effects to a program

From now on, you're on your own... Although the routines described above may prove useful, if you happen to need a siren or a wolf whistle, it is more than likely that you will have to 'create' an effect by ear. It then becomes a question of guessing what might sound right, listening to the result, and modifying it as seems necessary. Trial and error, in other words.

When adding sound effects to a program, there is one Golden Rule: don't overdo it! A continuous succession of beeps and bangs tends to get extremely irritating in a very short time. The effects should enhance the highlights of the game; they should not form a permanent background.

One final remark: most commercial TV games units contain a sound effects generator that produces a few sounds (explosions and the like) according to data stored in 1E80. Programs intended for these systems will not make use of the PSGs, so the explosions won't come out!

20

Writing programs...

...for use by others

When you've worked out a good program, it's nice to be able to pass it on to other TV games computer owners. Since everyone uses the same cassette interface, this would seem a simple matter: copy your program onto tape and pass it on. In practice, it is not always that easy... This chapter gives a brief summary of some of the points that bear watching.

With the information given in this book, a lot of practice and a liberal dose of inspiration, any TV games computer owner can come up with interesting programs. Several general-purpose routines are given throughout the book and most of these are repeated in the Appendix. One of the Golden Rules for computer programmers is: 'Don't re-invent the wheel – steal the plans!' However, when a 'home-brew' program is intended for use by others, there are some further points that must be kept in mind.

Joystick calibration

As stressed in chapter 13, this is absolutely essential. If joysticks are used, one of the routines described (or a suitably modified version) must be incorporated in the program.

Basic versus extended versions

As mentioned earlier, programs tend to expand until they fill all available memory. In the basic version, this isn't too bad: 1¾K is still a manageable amount of memory. However, when the extended version is available things start to look different. A program that spreads over nearly five Kbytes must be either highly sophisticated – or rather clumsy!
It is a good idea to make every effort to squeeze the final program into the basic memory area (Ø8CØ...ØFFF), since this makes it suitable for all other owners of the system. When working on the program, this leaves the whole extended memory area free for temporary patches, edit routines and so on.
A second approach can be even better, in some cases. A basic version of the game is contained in the memory range up to ØFFF; somewhere in the initialisation routine it includes the following check:

```
ØC1ØØØ    LODA, RØ
BAFC       BSFR, ind
```

If extended memory is not available, the value retrieved from address 1ØØØ will be FF. Since this is negative, the branch to subroutine will not be

Table 45

```
1000    071E        LODI, R3
   2    0620        LODI, R2
   4    A703        SUBI, R3
   6    A602        SUBI, R2
   8    0503        LODI, R3
   A    0F5400      LODA, I-R3   }
   D    C80C        STRR, R0     } load destination
   F    0F5400      LODA, I-R3   } (base) address
1012    6440        IORI, R0     }
   4    C804        STRR, R0
   6    0E5400      LODA, I-R2   }
   9    CD4000      STRA, I-R1   } transfer data
   C    5978        BRNR, R1     }
   E    5B64        BRNR, R3
1020    ..........              further initialisation
1400    0A5D        1F1240      'game end'      }
   5    0A84        1F1170      'VRLE'          }
   A    095F        1F11F0      'dragon killed' } new data for
   F    096B        3F1200      'knight killed' } basic program
1414    094A        1F1238      'held'          }
   9    0B49        1F1220      'raise gate'    }
        dest.       new         function
        address     data
```

executed, and the basic game will run. The only minor imperfection is that the 'Read cassette' routine will be interrupted with the message 'Ad = 1000'. At that point, 'PC = 0900' must be entered by hand. When extended memory is available, and provided the byte at 1000 is in the 00 to 7F range (not 'negative'), the subroutine will be executed. This can then 'convert' the basic program by inserting branches to extended memory as required. As an example, Table 45 shows the routine used in the 'Maze adventure' game on the Elektor Software Service tape ESS 009.

Sound routines

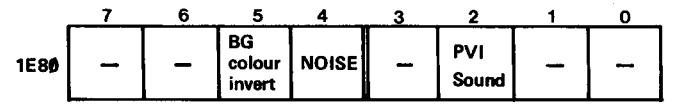
Three distinct types of sound generator may or may not be available: PVI sound, PSG sound and '1E80' sound. All TV games computers include PVI sound, so it is a safe bet to use this option. The extended version contains the PSGs, so it seems logical to use these when programs run into extended memory. Note that they should not be used – or only as further enhancement – when a program is located in the basic memory area.
Finally, those readers who have modified a commercial unit may also have a relatively simple 'explosion generator' at their disposal, located at address 1E80. This should not be used in games intended for other users! However, to accommodate these 'pirate' versions, it is advisable to store '04' in 1E80 to turn on this sound unit if it is present. Also, it may be useful to know

what this unit does – if only for modifying 'commercial' programs! – and so the function of the various bits is shown in figure 47. A 'conversion' routine can be inserted wherever the original program stores data in 1E8Ø, as shown in Table 46. This will drive the PSGs accordingly. An alternative is to add a sound generator at 1E8Ø – provided you can locate the necessary components!

Text routines

By their very nature, text routines suffer from a language problem. English texts or not ideal for Spanish readers! For this reason, symbols are often preferable. However, words can be much more interesting in certain cases.

Figure 47



82901 - 47

Table 46

Original instruction	Function	New instruction	PSG preset *
Ø4ØØ CC1E8Ø	sound off	Ø4F7 CC1DØ7	1DØE = ØØ
2Ø CC1E8Ø	sound off	2Ø CC5CØ8	1DØE = ØØ
Ø4Ø4 CC1E8Ø	PVI sound on	Ø4 B7 CC1DØ7	—
Ø41Ø CC1E8Ø	noise on (explosion)	Ø4ØØ CC1DØD	1DØ6 = 1F 1DØ7 = F7 1DØ8 = 1Ø 1DØC = 17
Ø41Ø CC1E8Ø	noise on	Ø4ØF CC1DØ8	1DØ6 = 1F 1DØ7 = F7
↓ Ø4ØØ CC1E8Ø	↓ noise off	↓ Ø4ØØ CC1DØ8	—

* After 'RESET' the data in 1DØE is ØØ, so this need not be included as a preset. The other preset data can be stored in the PSG when starting the program, if it is required at all.

46

There is no real harm in this, provided one of the routines given in this book is used. A reference to the corresponding routine and information on where the data is located will then be sufficient for readers who wish to translate the texts.

Recording level

Although the cassette routines used are quite reliable, the same can not be said, unfortunately, for cassette recorders. This is understandable, since these machines are intended for audio work – not for transmission of digital data. To play it safe, it is advisable to make several recordings at different levels: −6dB, 0dB and +6dB, for instance. Furthermore, Dolby, DNL or whatever should not be used: these noise reduction systems cause havoc with a digital data stream.
Furthermore, it is highly advisable to include a copy of the program. That way, even a tape that will only load with a few 'glitches' can be cleaned up by hand. For that matter, documentation is always useful: it gives the recipient the opportunity to modify the program according to his personal taste!

Programs for the ESS

Programs submitted to us for inclusion in the Elektor Software Service are always welcome – provided they satisfy the conditions outlined in this chapter! It should be noted that payment for suitable programs is not based on their length, since this tends to encourage 'messy' programming. Instead, factors like 'originality', 'effect', 'sophistication' and 'interest to other TV games owners' are considered.

21

Plug-in EPROM programmer

Program 2716s the easy way!

The circuit described here can be used to load programs or other data into 2716 EPROMs. For this fairly complicated job it uses only a single 555 timer and one TTL IC, plus a handful of little components. Impossible? Not when you get the μP itself to do most of the work!

Loading data into a 2716 is a fairly simple job, as shown in figure 48. You need a 25 V programming voltage, and a programming pulse at ordinary TTL level and lasting for 50 ms. To be more precise, the $\overline{OE}$ pin must be pulled high and then the $\overline{CE}$ input is also set to logic 1. This situation must be maintained for 50 ms, after which $\overline{OE}$ and $\overline{CE}$ go low again. To check whether this programming cycle was successful, the data can now be read out without first having to switch off the 25 V supply.
A normal EPROM programmer is quite a complicated machine, since it must be capable of retrieving the desired data from another EPROM or whatever; putting the various signals onto the new EPROM's pins in the correct sequences; checking for a successful 'store' and repeating the process if necessary. A self-contained unit to do this can't possibly be a 'simple' circuit. However, it helps a lot when you hit on the idea of letting an existing microprocessor system do most of the work. The system described here takes this idea one step further.
In many cases, the EPROM will be intended for use in the microprocessor system. This means that there will be a socket for it on the board. A small

Table 47: 2716 operating modes.

PINS / MODE	$\overline{CE}$/PGM (18)	$\overline{OE}$ (20)	V_{PP} (21)	V_{CC} (24)	OUTPUTS (9-11, 13-17)
Read	V_{IL}	V_{IL}	+5	+5	D_{OUT}
Standby	V_{IH}	Don't care	+5	+5	High Z
Program	pulsed V_{IL} to V_{IH}	V_{IH}	+25	+5	D_{IN}
Program Verify	V_{IL}	V_{IL}	+25	+5	D_{OUT}
Program Inhibit	V_{IL}	V_{IH}	+25	+5	High Z

auxiliary circuit can be plugged into this socket, after which the EPROM is plugged into the top of this. A piggy-back arrangement, in other words. Four flying leads go to the auxiliary circuit, after which the 2716 can be programmed by means of an ordinary write operation to the corresponding address! This is followed by a normal 'read' to check whether the data was transferred correctly. It is interesting to note that any single location or group of locations can be programmed in this way. In other words, starting from an erased EPROM, bits and pieces can be loaded at different times – you don't have to fill the whole thing in one go.

The circuit

In the TV games computer, the $\overline{\text{OPACK}}$ signal stops the processor 'in its tracks' as it were, and the data and address lines remain unchanged. This makes it an easy matter to keep all the address and data bits stable during the 50 ms programming cycle. The necessary $\overline{\text{OPACK}}$ signal can be generated quite easily by means of a 555 timer.

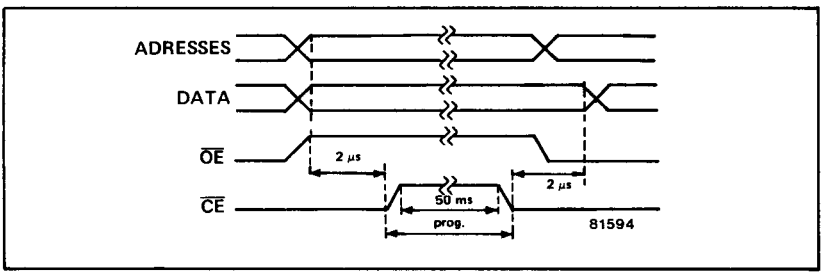


Figure 48. A 2716 is remarkably easy to program. Provided the programming voltage (25 V) is connected, you only have to ensure that both $\overline{\text{OE}}$ and $\overline{\text{CE}}$ are pulled high at the correct times.

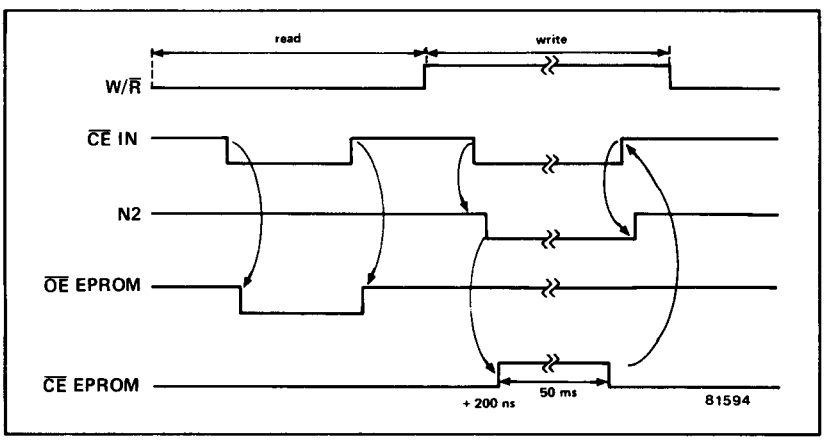


Figure 49. The circuit converts the W/$\overline{\text{R}}$ and $\overline{\text{CE}}$ signals from the processor into the necessary $\overline{\text{OE}}$ and $\overline{\text{CE}}$ pulses for the EPROM. Note that $\overline{\text{OE}}$ must be low for Read and high for Write.

The basic requirements are given in figure 49. The EPROM is enabled by pulling the $\overline{CE}$ input low. For a read cycle, nothing special happens. As can be seen in figure 50, the R/W input (at logic Ø for Read!) will cause the output of N2 to go to logic 1; this, combined with the logic 1 from N1, will pull the $\overline{OE}$ pin low via N3. The data is now read out from the EPROM in the normal way; its $\overline{CE}$ input remains low throughout.
For a write operation, things must obviously become slightly more complicated. The $\overline{CE}$ input to the circuit is logic Ø in this case, so the output from N3 ($\overline{OE}$ for the EPROM) is maintained at logic 1 throughout.

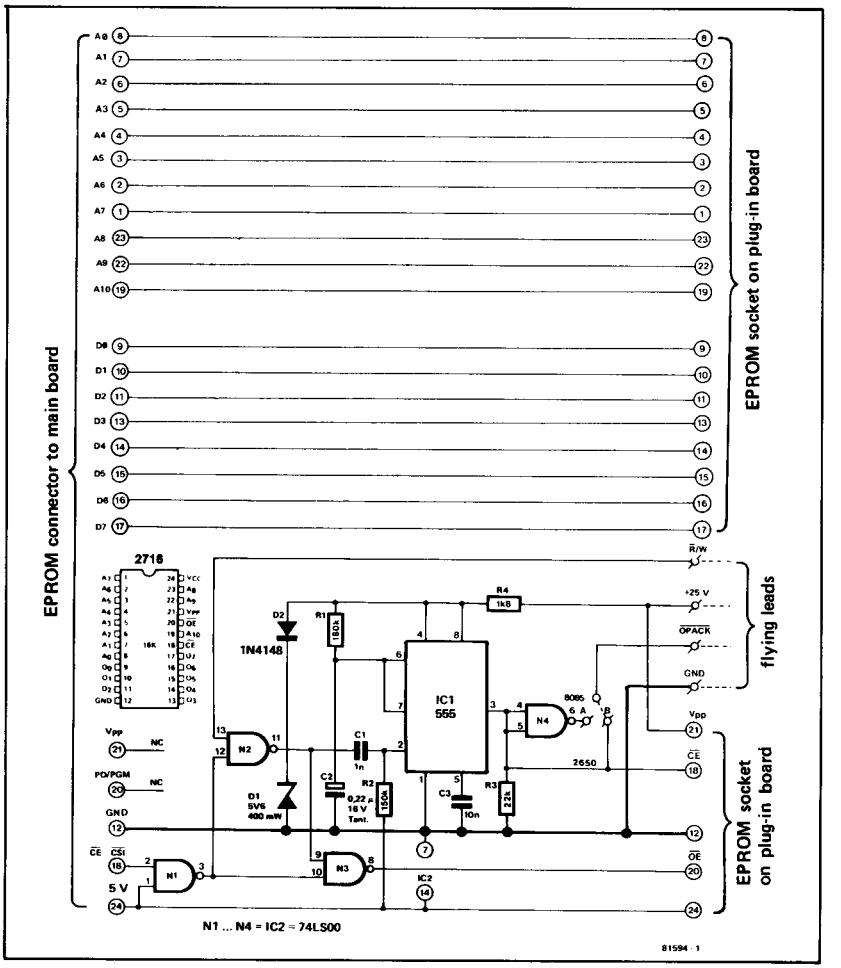


Figure 50. The complete circuit. Most of the signals that are present at the original EPROM socket can be left unmodified for programming. The original $\overline{CE}$ signal is combined with R/W (the latter is connected via a flying lead) to produce $\overline{OE}$ and $\overline{CE}$ for the EPROM to be programmed and $\overline{OPACK}$ for the μP. The supply to the 555 is derived from the 25 V programming voltage.

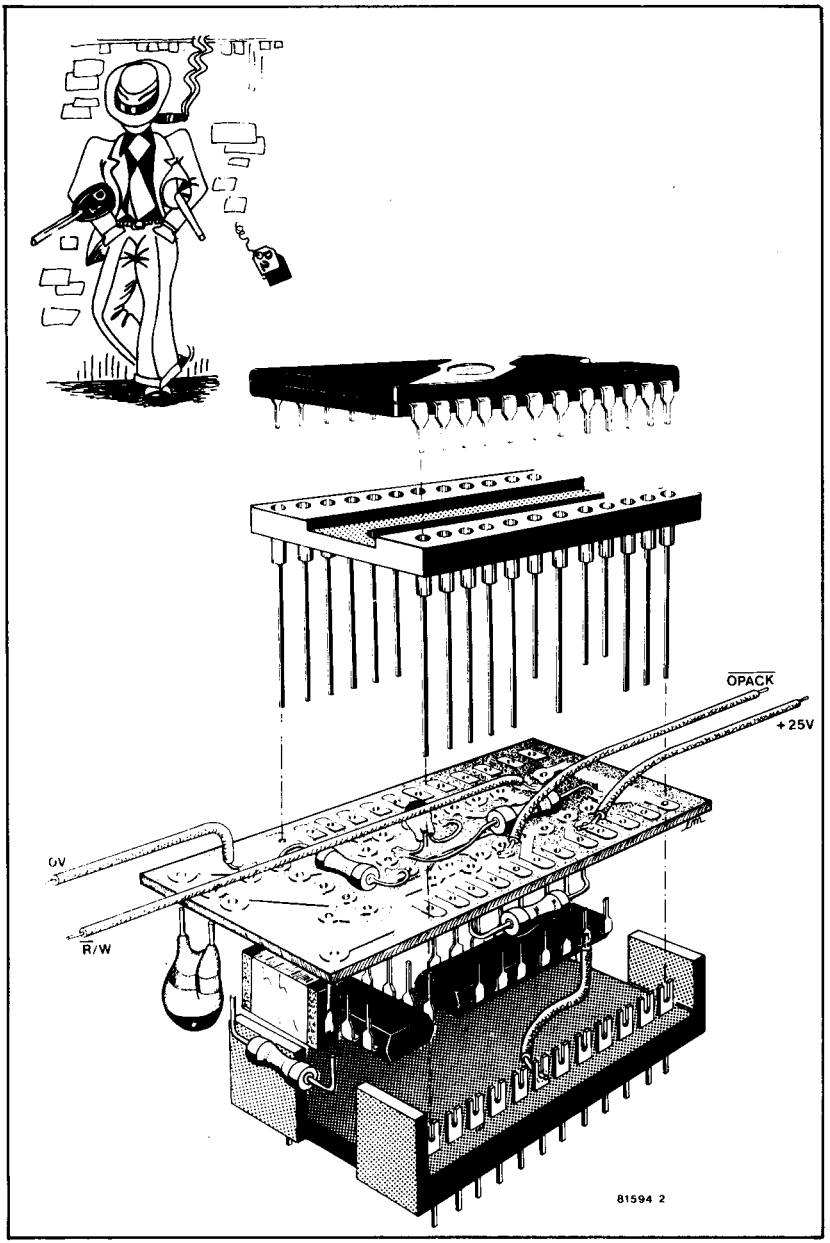
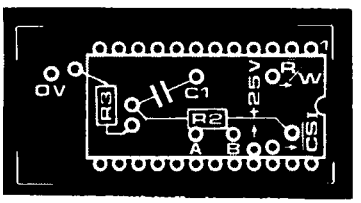
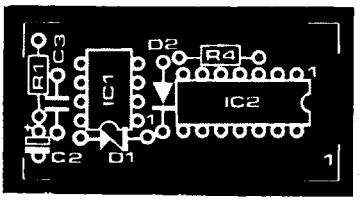
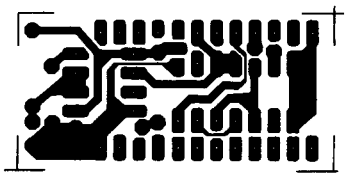


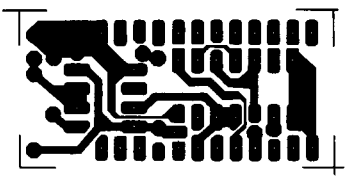
Figure 51. This drawing clearly illustrates the construction. After mounting all components on the p.c. board, short and stiff wires are soldered through the corresponding holes for mounting the DIL connector underneath and the IC socket on top. Alternatively, a wire-wrap socket can be used at the top; its pins are long enough to go right through the board to the connector underneath. Note that D1 and D2 are not shown in this drawing.

Furthermore, the negative-going edge at the output of N2 triggers the timer (IC1) so that its output goes high. This is the $\overline{CE}$ signal for the EPROM. As can be seen in figure 48, both $\overline{OE}$ and $\overline{CE}$ for the EPROM being at logic 1 corresponds to the programming mode! During the 50 ms output period from the timer, the $\overline{OPACK}$ output is also held at logic 1 to stop the processor – for an 8085 μP this signal can be inverted via N4, but in our application this gate is bypassed by taking the $\overline{OPACK}$ output from point B.
R1 and C2 give a calculated 555 period of 45 ms, when C2 has its nominal value. This is on the short side, but in practice this type of capacitor has an effectively larger capacitance in this very-low-frequency application. If measuring equipment is available, R1 can be tailored until the period is exactly 50 ms; however, the values given have proved reliable in all our tests.









Parts list

Resistors:
R1 = 180 k
R2 = 150 k
R3 = 22 k
R4 = 1k8
all resistors are 1/8 watt types

Capacitors:
C1 = 1 n
C2 = 0.22 μ/16 V tantalum
C3 = 10 n

Semiconductors:
D1 = 5.6 V/400 mW zener diode
D2 = 1N4148
IC1 = 555
IC2 = 74LS00

Miscellaneous:
24 pin IC socket
24 pin IC connector

Figure 52. The printed circuit board and component layout. Components are mounted on both sides of the board. Note that pins 20 and 21 of the DIL plug are not connected to the board, and that pin 18 is linked by means of a separate wire to pin 2 of IC2. This is indicated as 'CSI' on the board.

After each programming cycle, the circuit must be given time to 'settle' (2.5 ms at least). In the program example given, a delay is included for this. Perfectionists may have noticed that the programming signals for the EPROM are not quite right: there is no 2 μs delay between the moment that the address and data is applied and the start of the $\overline{CE}$ pulse. In practice, this has never caused us any trouble.
One final point: databooks do not seem to agree entirely on which 2716 pin is the $\overline{CE}$ and which is $\overline{OE}$! However, if you use the pin *numbers* given in the circuit (18 for $\overline{CE}$ and 20 for $\overline{OE}$) the circuit will work.
The printed circuit board is shown in figure 52.

Connection to the TV games computer

In principle, there are two positions worth considering for programming an EPROM: the monitor ROM on the main board and the plug-in cartridge connectors on the extension board. Using the monitor ROM position is not such a good idea, since in practice this would involve removing the ROM and plugging in the programmer board while the whole unit is switched on. For this reason, we will only consider the second option.
Assuming that the INT connector is used – since this is suitable for single-sided extension boards – the $\overline{R}$/W signal is available on pin 1 of IC3, and this can be connected to pin 27 of the INT connector. That takes care of one 'flying lead'. The 0V and +25V leads must be brought out to an external power supply, as this voltage is not available in the TV games unit. Finally, the $\overline{OPACK}$ line (from point B on the programmer board) must be connected to a suitable point on the main board. The correct track runs from pin 36 of IC1 to one end of R26 (and crosses over to the other side of the board and back to reach pin 17 of IC3). However, at this point we hit a snag: the PVI also makes use of the processor's $\overline{OPACK}$ input! This means that the two signals will have to be added, as shown in figure 53. The actual circuit is given in figure 53a: an OR gate and a pull-up resistor are added, and one input of the OR gate is brought out to a miniature jack plug with built-in switch. The latter ensures that both inputs of the gate are driven by the PVI when the programmer is not plugged in. The construction details are shown in figure 53b. A single track under IC1 (but on the other side of the board) must be broken, and four connections are brought out as shown. Reading from top to bottom, these are supply common; the PVI output; the CPU input; positive supply.
So much for the hardware. The software side is fairly straightforward: a suitable program is given in Table 48. The 'wait for key' loops at the beginning and end of the program are included to facilitate switching from monitor to EPROM module. The screen colour is used to indicate the program status and possible errors. The first step is to check whether the designated address block is clear; if not, the screen colour is switched to white and the program stops. If all goes well, however, the screen remains blue and the actual programming starts. After each programming cycle, the location is checked for correct store. If an error is detected, a second or even third attempt will be made; if these also fail, the screen colour is

Figure 53.

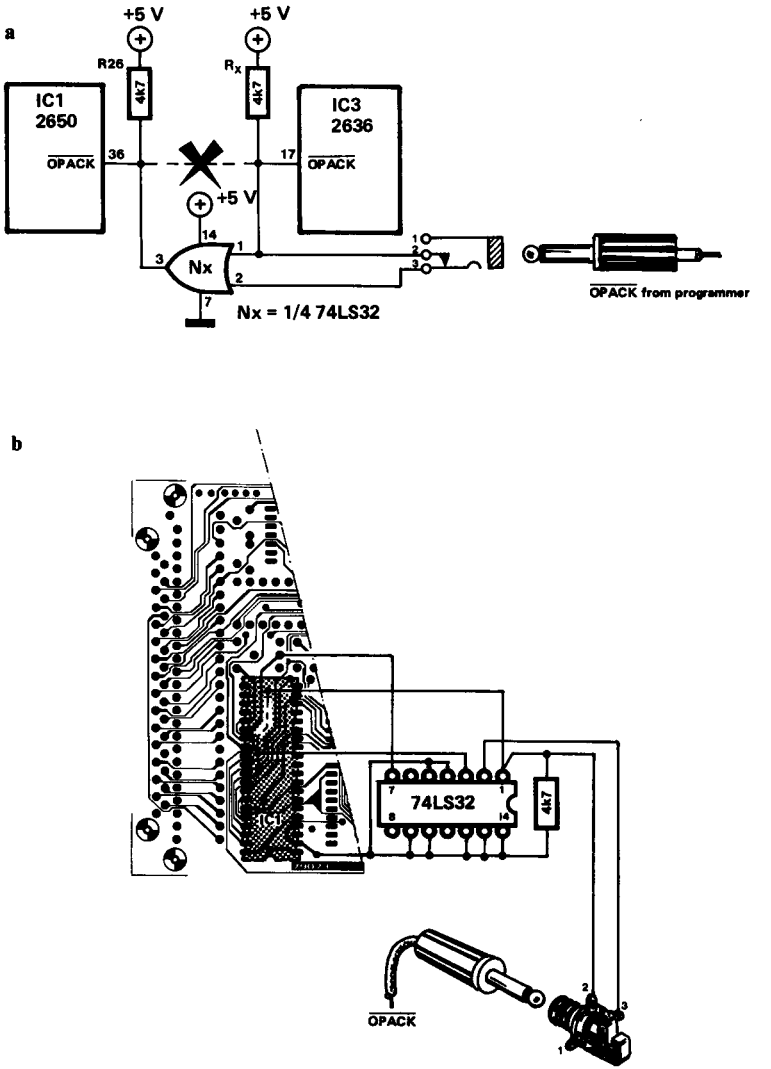


Figure 53. Providing an $\overline{\text{OPACK}}$ connection for the EPROM programmer involves a minor modification to the main board. One copper track must be broken, and four leads are brought out to an OR gate and a miniature jack plug. In this way, normal operation of the unit is maintained at all times.

Table 48

```
1900    xxxx        first address, data
   2    yyyy        first address, EPROM
   4    zzzz        last address,  EPROM

1906    7620        PPSU, II
   8  ▶0C1E89       LODA, R0  } wait for WCAS
   B  └─9A7B        BCFR      }
   D    07FF        LODI, R3
   F    0A71        LODR, R2
1911    0970        LODR, R1
   3  ▶CA03         STRR, R2
   5    C902        STRR, R1
   7    0C0000      LODA, R0
   A    E4FF        COMI, R0
   C  ┌─9824        BCFR
   E  │ EA64        COMR, R2    } is EPROM empty?
1920  │ 9804        BCFR          if not, error.
   2  │ E961        COMR, R1
   4  │ 1808        BCTR
   6  │▶7709        PPSL, WC + C
   8  │ 8500        ADDI, R1
   A  │ 8600        ADDI, R2
   C  │ 1B65        BCTR, UN
   E  └▶074C        LODI, R3
1930  ▶ 0603        LODI, R2
   2  │ 0D9900      LODA, R1 ind.
   5  │▶CD9902      STRA, R1 ind.
   8  │ 20          EORZ, R0    } data to EPROM. If store
   9  │▶F87E        BDRR, R0    } fails three times, error.
   B  │ ED9902      COMA, R1 ind.
   E  │ 1804        BCTR
1940  └─FA73        BDRR, R2
   2  ▶ 1B21        BCTR, UN
   4  │▶0604        LODI, R2
   6  │▶0502        LODI, R1
   8  │ 770B        PPSL, WC + COM + C
   A  │▶0E5900      LODA, I-R2
   D  │ ED7903      COMA, I/R2
1950  │ 1802        BCTR
   2  │ 7502        CPSL, COM   } increment addresses. If
   4  │▶8400        ADDI, R0    } end of range, end program.
   6  │ CE7900      STRA, I/R2
   9  │ F96F        BDRR, R1
   B  │ B502        TPSL, COM
   D  │ 1804        BCTR
   F  │ 5A65        BRNR, R2
1961  └─1B4D        BCTR, UN
   3  ▶ 072A        LODI, R3
   5  ▶CF1FC6       STRA, R3    set screen colour
   8  ▶0C1E8B       LODA, R0  }
   B  │ D0          RRL,  R0  } wait for Start key
   C  │ 9A7A        BCFR      }
   E  └─1F0000      BCTR, UN
```

switched to red and the program ends. In all normal cases, however, the EPROM should be programmed correctly – and this is indicated by the screen switching to green.

Using the unit

In practical terms, the whole procedure for programming a 2716 EPROM can be summed up as follows:

- Insert the auxiliary circuit in the EPROM socket on the plug-in cartridge board and plug the (erased) EPROM into the top; connect the R/W, OPACK and 0/25V lines with flying leads, as described above. Set S4 for the desired address range.
- Load the program given in Table 48, and branch to its start address (1906).
- Operate S1 or S2, to switch from RAM or monitor to the EPROM according to the desired address range.
- The actual programming procedure can now be started by operating the WCAS key. The screen should remain blue; after anything up to two minutes (depending on how much of the EPROM is to be programmed) the screen should switch to green – white or red indicate errors, as described above.
- Reset S1 (or S2) to its original position and operate the Start key to return to monitor. Disconnect the flying leads, remove the auxiliary circuit and insert the EPROM in its intended socket. Job done!

All this is relatively straightforward. However, it is based on a few unspoken assumptions: the program that is to be stored in EPROM must be suitable for this application – a lot of existing programs aren't! – and some kind of plug-in extension board must be available. To start with the first point:

- Many programs use 'short cuts' based on the fact that the program is to be stored in RAM: the data at several addresses is modified in the course of the program. This is no longer possible when the program is stored in EPROM, so the relevant sections will have to be modified. One relatively quick way to locate this sort of thing is to load the program from tape and then run it a few times. It can then be checked against the original (using the 'FIL-' mode); the check routine will terminate at the first address of any modified block.
- It will often prove necessary to move the program to a different address range. This involves a modification of nearly all the absolute addresses. The disassembler routine will simplify this job.
- If the program is to be located from 0000 to 07FF, the interrupt address will become 0003. This is rarely a problem. However, it should be noted that all 'monitor' subroutines in the program are no longer operative!
- If the program is located from 0800 to 0FFF, the monitor RAM scratch becomes inoperative. All monitor routines that use this area – text routines, keyboard scan, etc. – are blocked. Furthermore, the program cannot be started and the interrupt address will be incorrect unless the correct data is stored from 08B9...08BF, as described in chapter 16

under 'start up'.

- When loading data into the EPROM, the program data is stored elsewhere in memory. This is no real problem: after setting up (and testing) the program in RAM, a 'block transfer' can be used to move it up to the 1000...17FF area. Alternatively, as often occurs, it can be left in the 0900...10FF range or moved from there to 1100...18FF – the EPROM programming routine is located from 1900 on for this very reason!

All in all, a certain amount of skill and experience is involved, certainly when storing someone else's program. However, since incorrect software cannot possibly damage an EPROM, there is never any harm in trying!
The second assumption referred to above is that some kind of extension board is available. This point can be cleared up with an absolute minimum of effort: a suitable layout is given in figure 54!

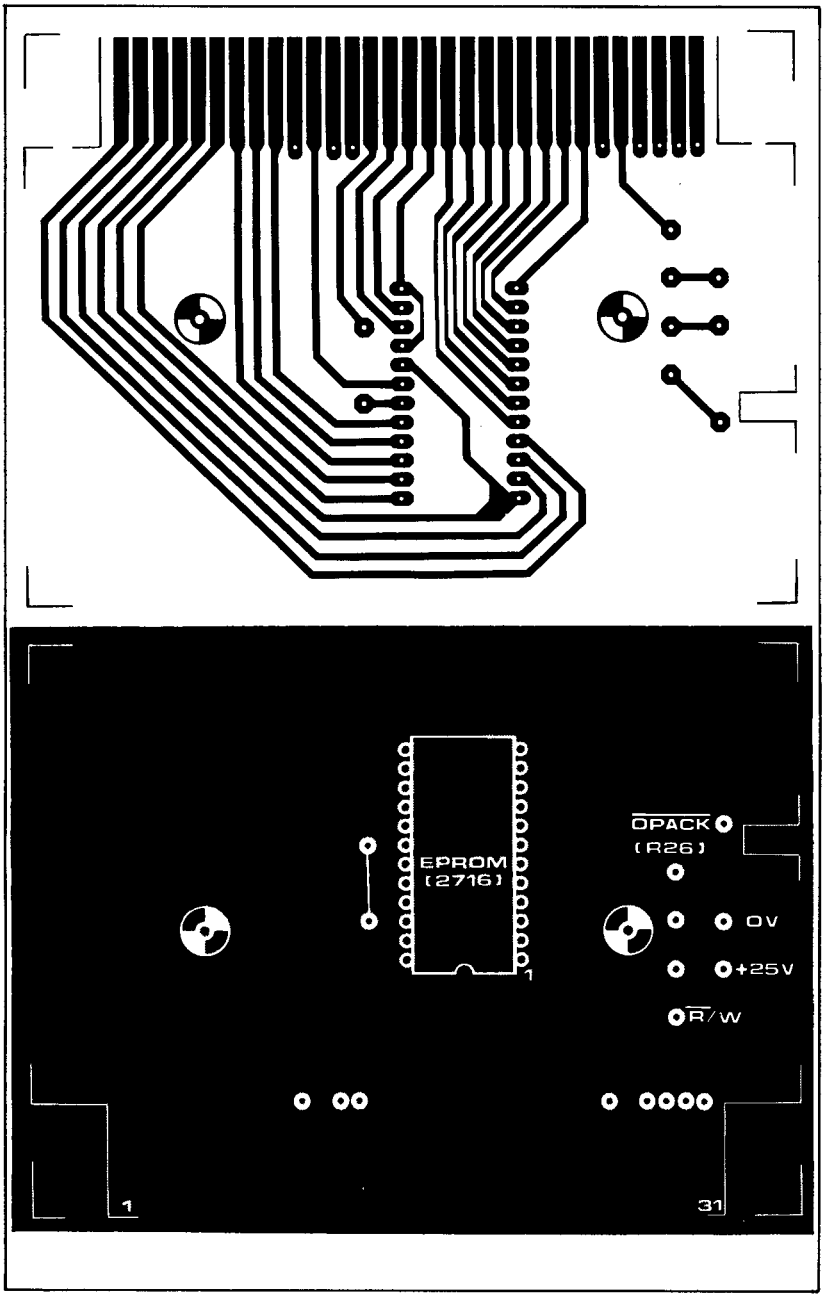
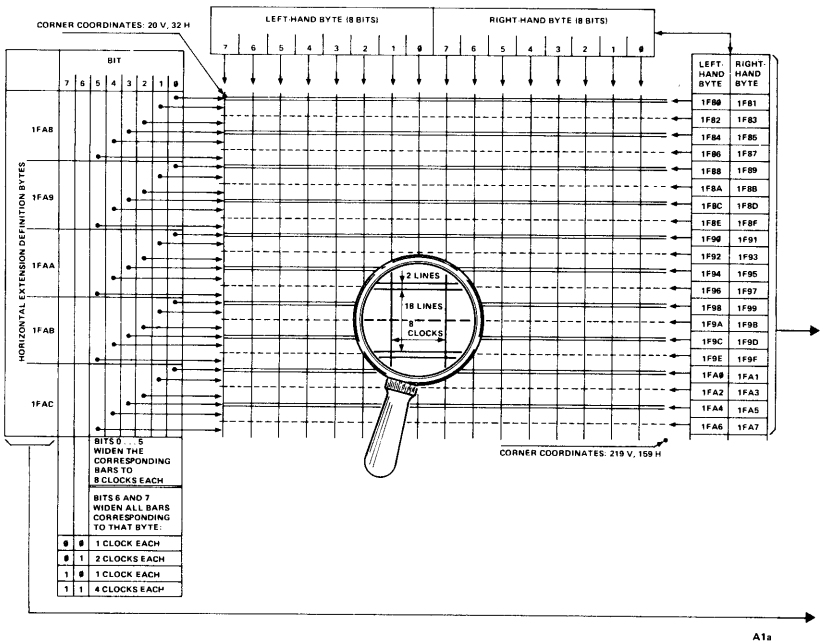


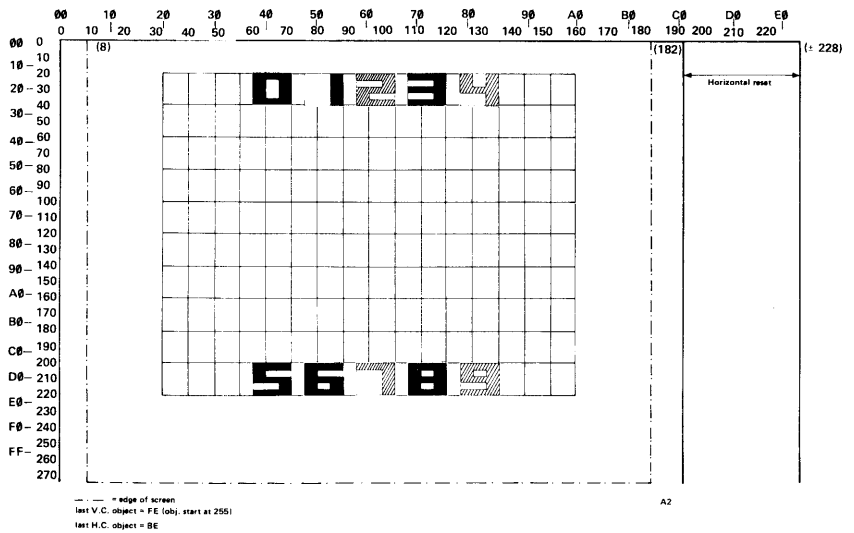
Figure 54. This board can be used for plugging 2716 EPROMS into the INT connector on the extension board.

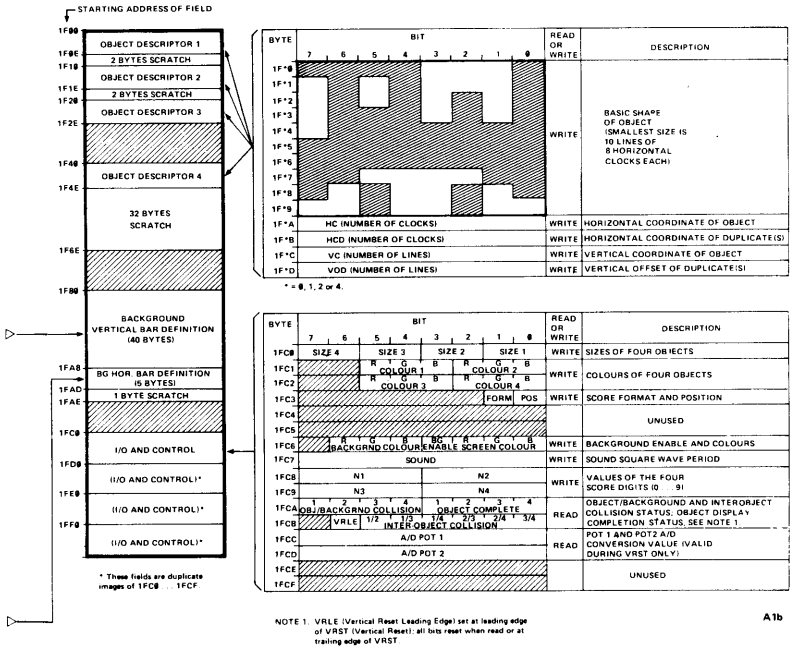
Appendix A:

- **PVI survey**
- **PSG survey**
- **2650 instruction set**

PVI survey


CORNER COORDINATES: 20 V, 32 H
LEFT-HAND BYTE (8 BITS)
RIGHT-HAND BYTE (8 BITS)
BIT
HORIZONTAL EXTENSION DEFINITION BYTES
1FA8
1FA9
1FAA
1FAB
1FAC
2 LINES
18 LINES
8 CLOCKS
CORNER COORDINATES: 219 V, 159 H
BITS 0 ... 5 WIDEN THE CORRESPONDING BARS TO 8 CLOCKS EACH
BITS 6 AND 7 WIDEN ALL BARS CORRESPONDING TO THAT BYTE:
0 0 1 CLOCK EACH
0 1 2 CLOCKS EACH
1 0 1 CLOCK EACH
1 1 4 CLOCKS EACH
A1a


(8)
(182)
(± 228)
Horizontal reset
0 1 2 3
5 6 7 8
= edge of screen
last V.C. object = FE (obj. start at 255)
last H.C. object = BE
A2

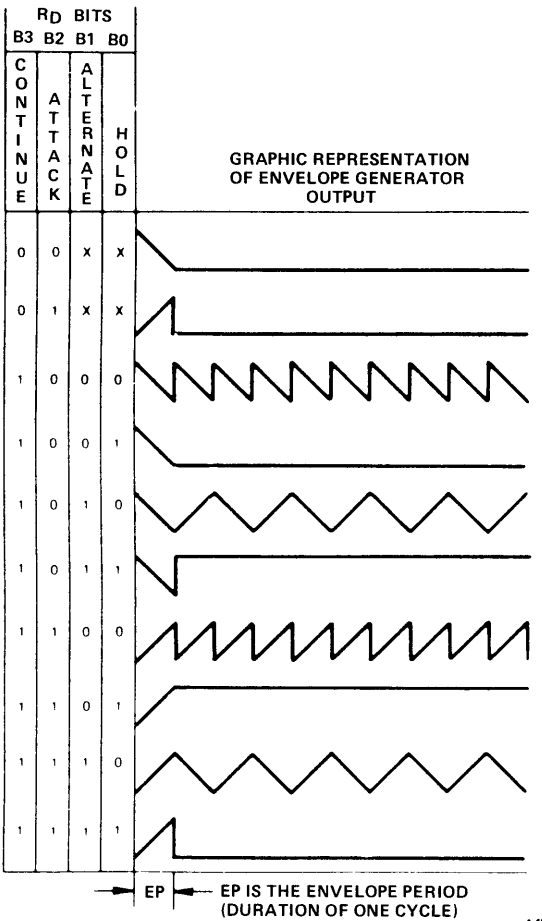


Notes:

- The object colours (address 1FC1, 1FC2) and the colour of the background lines (bits 4...6 at 1FC6) are selected by setting the corresponding bit to Ø. In effect, *setting* the bit *suppresses* the colour. The opposite applies for the screen colour: bits Ø...2 at 1FC6 select the corresponding colour when they are set to logic 1. The colour of the score digits is the opposite of the screen colour: if the screen is blue, the score will be red + green = yellow.
- The score format and position are determined at 1FC3 as follows: bit Ø determines the position (logic Ø for top of screen, logic 1 for bottom) and bit 1 sets the format (logic Ø for two groups of two digits, logic 1 for a single 4-digit number).
- The 'scratch' areas in the PVI address field can be used as normal RAM, for storing program data etc. The object descriptor areas and background definition range can also be read at all times in the normal way. In the I/O and control section, however, reading data causes that location to be reset to ØØ.

REGISTER \ BIT		B7	B6	B5	B4	B3	B2	B1	B0
R0	Channel A Tone Period	8-BIT Fine Tune A							
R1		4-BIT Coarse Tune A							
R2	Channel B Tone Period	8-BIT Fine Tune B							
R3		4-BIT Coarse Tune B							
R4	Channel C Tone Period	8-BIT Fine Tune C							
R5		4-BIT Coarse Tune C							
R6	Noise Period	5-BIT Period Control							
R7	Enable	IN/OUT		Noise			Tone		
		IOB	IOA	C	B	A	C	B	A
R8	Channel A Amplitude	M	L3	L2	L1	L0			
R9	Channel B Amplitude	M	L3	L2	L1	L0			
RA	Channel C Amplitude	M	L3	L2	L1	L0			
RB	Envelope Period	8-BIT Fine Tune E							
RC		8-BIT Coarse Tune E							
RD	Env. Shape/Cycle	CONT	ATT	ALT	HOLD				
RE	I/O Port A Data Store	8-BIT PARALLEL I/O on Port A							
RF	I/O Port B Data Store	8-BIT PARALLEL I/O Port B							

A4a



A4b

OCTAVE	NOTE	2x8-BIT REGISTERS COARSE	FINE	IDEAL FREQUENCY	ACTUAL FREQUENCY
0	A	0F	BE	27.500	27.503
0	A#	0E	DC	29.135	29.137
0	B	0E	07	30.868	30.865
1	C	0D	3D	32.703	32.705
1	C#	0C	7F	34.648	34.647
1	D	0B	CB	36.708	36.713
1	D#	0B	22	38.891	38.890
1	E	0A	82	41.203	41.203
1	F	09	EB	43.654	43.654
1	F#	09	5D	46.249	46.240
1	G	08	D6	48.999	48.999
1	G#	08	57	51.913	51.914
1	A	07	DF	55.000	55.006
1	A#	07	6E	58.270	58.274
1	B	07	03	61.735	61.747
2	C	06	9F	65.406	65.390
2	C#	06	3F	69.296	69.316
2	D	05	E6	73.416	73.402
2	D#	05	91	77.782	77.780
2	E	05	41	82.407	82.406
2	F	04	F6	87.307	87.273
2	F#	04	AE	92.499	92.518
2	G	04	6B	97.999	97.999
2	G#	04	2C	103.826	103.780
2	A	03	F0	110.000	109.957
2	A#	03	B7	116.541	116.547
2	B	03	82	123.471	123.426
3	C	03	4F	130.813	130.858
3	C#	03	20	138.591	138.546
3	D	02	F3	146.832	146.804
3	D#	02	C8	155.563	155.669
3	E	02	A0	164.814	164.936
3	F	02	7B	174.614	174.546
3	F#	02	57	184.997	185.036
3	G	02	35	195.998	196.171
3	G#	02	16	207.652	207.559
3	A	01	F8	220.000	219.914
3	A#	01	DC	233.082	232.850
3	B	01	C1	246.942	246.852
4	C	01	A8	261.626	261.407
4	C#	01	90	277.183	277.092
4	D	01	79	293.665	293.996
4	D#	01	64	311.127	311.339
4	E	01	50	329.628	329.871
4	F	01	3D	349.228	349.642
4	F#	01	2C	369.994	369.456
4	G	01	1B	391.995	391.649
4	G#	01	0B	415.305	415.119
4	A	00	FC	440.000	439.828
4	A#	00	EE	466.164	465.700
4	B	00	E0	493.883	494.807

OCTAVE	NOTE	2x8 BIT-REGISTERS COARSE	FINE	IDEAL FREQUENCY	ACTUAL FREQUENCY
5	C	00	D4	523.251	522.814
5	C#	00	C8	554.365	554.183
5	D	00	BD	587.330	586.437
5	D#	00	B2	622.254	622.678
5	E	00	A8	659.255	659.742
5	F	00	9F	698.456	697.086
5	F#	00	96	739.989	738.911
5	G	00	8D	783.991	786.076
5	G#	00	85	830.609	833.358
5	A	00	7E	880.000	879.656
5	A#	00	77	932.328	931.400
5	B	00	70	987.767	989.613
6	C	00	6A	1046.502	1045.629
6	C#	00	64	1108.731	1108.367
6	D	00	5E	1174.659	1179.113
6	D#	00	59	1244.508	1245.356
6	E	00	54	1318.510	1319.484
6	F	00	4F	1396.913	1402.996
6	F#	00	4B	1479.978	1477.822
6	G	00	47	1567.982	1561.080
6	G#	00	43	1661.219	1654.278
6	A	00	3F	1760.000	1759.312
6	A#	00	3B	1864.655	1878.587
6	B	00	38	1975.533	1979.226
7	C	00	35	2093.005	2091.258
7	C#	00	32	2217.461	2216.733
7	D	00	2F	2349.318	2358.227
7	D#	00	2D	2489.016	2463.037
7	E	00	2A	2637.020	2638.968
7	F	00	28	2793.826	2770.916
7	F#	00	25	2959.955	2995.585
7	G	00	23	3135.963	3166.762
7	G#	00	21	3322.438	3358.687
7	A	00	1F	3520.000	3575.376
7	A#	00	1E	3729.310	3694.555
7	B	00	1C	3951.066	3958.452
8	C	00	1A	4186.009	4262.948
8	C#	00	19	4434.922	4433.466
8	D	00	18	4698.636	4618.194
8	D#	00	16	4978.032	5038.030
8	E	00	15	5274.041	5277.936
8	F	00	14	5587.652	5541.833
8	F#	00	13	5919.911	5833.508
8	G	00	12	6271.927	6157.592
8	G#	00	11	6644.875	6519.803
8	A	00	10	7040.000	6927.291
8	A#	00	0F	7458.620	7389.110
8	B	00	0E	7902.133	7916.904
9	C	00	0D	8372.018	8525.897

EQUAL TEMPERED CHROMATIC SCALE (f_{CLOCK} = 1.773387 MHz)

2650 Instruction set

	DESCRIPTION OF OPERATION	FORMAT	MNEMONIC	OP CODE R or CC 0	1	2	3	BYTES	CYCLES	BIT FORMAT	PSW BITS AFFECTED CC	IDC	C	OVF	SP	II	F	NOTE
LOAD/STORE	Load register zero	2	LOD Z	00	01	02	03	1	2	Z	•							1
	Load immediate	5	LOD I	04	05	06	07	2	2	I	•							1
	Load relative	7	LOD R	08	09	0A	0B	2	3	R	•							1,6
	Load absolute	10	LOD A	0C	0D	0E	0F	3	4	A	•							6
	Store register zero	2	STR Z	—	C1	C2	C3	1	2	Z	•							1
	Store relative	7	STR R	C8	C9	CA	CB	2	3	R								6
	Store absolute	10	STR A	CC	CD	CE	CF	3	4	A								6
ARITHMETIC	Add to register zero w/wo carry	2	ADD Z	80	81	82	83	1	2	Z	•	•	•	•				1
	Add immediate w/wo carry	5	ADD I	84	85	86	87	2	2	I	•	•	•	•				1
	Add relative w wo carry	7	ADD R	88	89	8A	8B	2	3	R	•	•	•	•				1,6
	Add absolute w wo carry	10	ADD A	8C	8D	8E	8F	3	4	A	•	•	•	•				1,6
	Subtract from register zero w wo borrow	2	SUB Z	A0	A1	A2	A3	1	2	Z	•	•	•	•				1
	Subtract immediate w wo borrow	5	SUB I	A4	A5	A6	A7	2	2	I	•	•	•	•				1
	Subtract relative w wo borrow	7	SUB R	A8	A9	AA	AB	2	3	R	•	•	•	•				1,6
	Subtract absolute w wo borrow	10	SUB A	AC	AD	AE	AF	3	4	A	•	•	•	•				1,6
	Decimal adjust register	3	DAR	94	95	96	97	1	3	Z	•							1,10
LOGICAL	AND to register zero	2	AND Z	40	41	42	43	1	2	Z	•							1
	AND immediate	5	AND I	44	45	46	47	2	2	I	•							1
	AND relative	7	AND R	48	49	4A	4B	2	3	R	•							1,6
	AND absolute	10	AND A	4C	4D	4E	4F	3	4	A	•							1,6
	Inclusive-OR to register zero	2	IOR Z	60	61	62	63	1	2	Z	•							1
	Inclusive-OR immediate	5	IOR I	64	65	66	67	2	2	I	•							1
	Inclusive-OR relative	7	IOR R	68	69	6A	6B	2	3	R	•							1,6
	Inclusive-OR absolute	10	IOR A	6C	6D	6E	6F	3	4	A	•							1,6
	Exclusive-OR to register zero	2	EOR Z	20	21	22	23	1	2	Z	•							1
	Exclusive-OR immediate	5	EOR I	24	25	26	27	2	2	I	•							1
	Exclusive-OR relative	7	EOR R	28	29	2A	2B	2	3	R	•							1,6
	Exclusive-OR absolute	10	EOR A	2C	2D	2E	2F	3	4	A	•							1,6
ROTATE/COMPARE	Compare to register zero arithmetic/logical	2	COM Z	E0	E1	E2	E3	1	2	Z	•							2
	Compare immediate arithmetic/logical	5	COM I	E4	E5	E6	E7	2	2	I	•							3
	Compare relative arithmetic/logical	7	COM R	E8	E9	EA	EB	2	3	R	•							3,6
	Compare absolute arithmetic/logical	10	COM A	EC	ED	EE	EF	3	4	A	•							3,6
	Rotate register w/wo carry	3	RRR	50	51	52	53	1	2	Z	•	•	•	•				1
	Rotate register left w/wo carry	3	RRL	D0	D1	D2	D3	1	2	Z	•	•	•	•				1
BRANCH	Branch on condition true relative	7	BCT R	18	19	1A	1B	2	3	R								7,8
	Branch on condition true absolute	8	BCT A	1C	1D	1E	1F	3	3	B								7,8
	Branch on condition false relative	7	BCF R	98	99	9A	—	2	3	R								7
	Branch on condition false absolute	8	BCF A	9C	9D	9E	—	3	3	B								7
	Branch on register non-zero relative	7	BRN R	58	59	5A	5B	2	3	R								7,8
	Branch on register non-zero absolute	8	BRN A	5C	5D	5E	5F	3	3	B								7,8

A6a

	DESCRIPTION OF OPERATION	FORMAT	MNEMONIC	OP CODE R or CC 0	1	2	3	BYTES	CYCLES	BIT FORMAT	PSW BITS AFFECTED CC	IDC	C	OVF	SP	II	F	NOTE
BRANCH	Branch on incrementing register relative	7	BIR R	D8	D9	DA	DB	2	3	R								7,8
	Branch on incrementing register absolute	8	BIR A	DC	DD	DE	DF	3	3	B								7,8
	Branch on decrementing register relative	7	BDR R	F8	F9	FA	FB	2	3	R								7,8
	Branch on decrementing register absolute	8	BDR A	FC	FD	FE	FF	3	3	B								7,8
	Zero branch relative, unconditional	6	ZBRR	—	—	—	9B	2	3	ER								6
	Branch indexed absolute, unconditional	9	BXA	—	—	—	9F	3	3	EB								5,6
SUBROUTINE BRANCH/RETURN	Branch to subroutine on condition true, relative	7	BST R	38	39	3A	3B	2	3	R					•			7,8
	Branch to subroutine on condition true, absolute	8	BST A	3C	3D	3E	3F	3	3	B					•			7,8
	Branch to subroutine on condition false, relative	7	BSF R	B8	B9	BA	—	2	3	R					•			7
	Branch to subroutine on condition false, absolute	8	BSF A	BC	BD	BE	—	3	3	B					•			7
	Branch to subroutine on non-zero register, relative	7	BSN R	78	79	7A	7B	2	3	R					•			7,8
	Branch to subroutine on non-zero register, absolute	8	BSN A	7C	7D	7E	7F	3	3	B					•			7,8
	Zero branch to subroutine relative, unconditional	6	ZBSR	—	—	—	BB	2	3	ER								6
	Branch to subroutine, indexed, absolute unconditional	9	BSXA	—	—	—	BF	3	3	EB								5,6
	Return from subroutine, conditional	3	RET C	14	15	16	17	1	3	Z					•			8
	Return from subroutine and enable interrupt, conditional	3	RET E	34	35	36	37	1	3	Z					•	•		8
INPUT/OUTPUT	Write data	3	WRTD	F0	F1	F2	F3	1	2	Z								
	Read data	3	REDD	70	71	72	73	1	2	Z	•							1
	Write control	3	WRTC	B0	B1	B2	B3	1	2	Z								
	Read control	3	REDC	30	31	32	33	1	2	Z	•							1
	Write extended	5	WRTE	D4	D5	D6	D7	2	3	I								
	Read extended	5	REDE	54	55	56	57	2	3	I	•							1
MISC.	Halt, enter wait state	1	HALT	40	—	—	—	1	1	E								
	No operation	1	NOP	C0	—	—	—	1	1	E								
	Test under mask immediate	5	TMI	F4	F5	F6	F7	2	3	I	•							4
PROGRAM STATUS	Load program status, upper	1	LPS U	—	—	92	—	1	2	E					•	•	•	
	Load program status, lower	1	LPS L	—	—	93	—	1	2	E	•	•	•	•				9
	Store program status, upper	1	SPS U	—	—	12	—	1	2	E	•							1
	Store program status, lower	1	SPS L	—	—	13	—	1	2	E	•							1
	Clear program status, upper, masked	4	CPS U	—	—	74	—	2	3	EI					•	•	•	
	Clear program status, lower, masked	4	CPS L	—	—	75	—	2	3	EI	•	•	•	•				9
	Preset program status, upper, masked	4	PPS U	—	—	76	—	2	3	EI					•	•	•	
	Preset program status, lower, masked	4	PPS L	—	—	77	—	2	3	EI	•	•	•	•				9
	Test program status, upper, masked	4	TPS U	—	—	B4	—	2	3	EI	•							4
	Test program status, lower, masked	4	TPS L	—	—	B5	—	2	3	EI	•							4

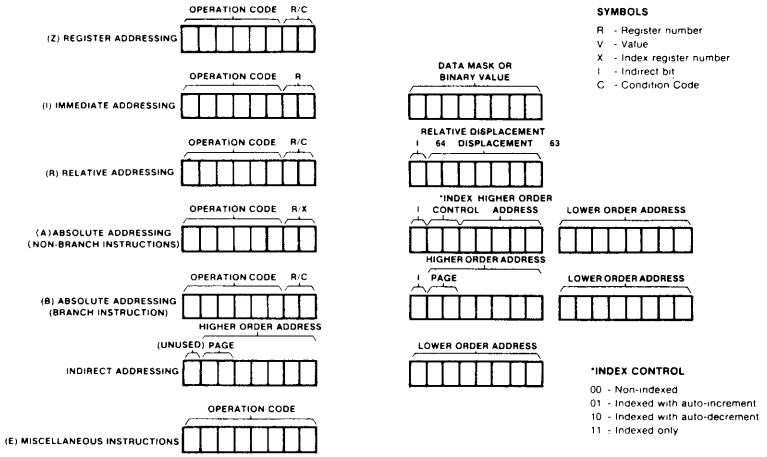
NOTES

1. Condition code (CC1, CC0): 01 if positive; 00 if zero; 10 if negative
2. Condition code (CC1, CC0): 01 if R0 > r; 00 if R0 = r; 10 if R0 < r
3. Condition code (CC1, CC0): 01 if r > V; 00 if r = V; 10 if r < V
4. Condition code (CC1, CC0): 00 if all selected bits are 1s; 10 if not all the selected bits are 1s
5. Index register must be register 3 or 3
6. Requires two additional cycles if indirection is specified
7. Requires two additional cycles if indirection is specified and branch is taken
8. Specify CC = 11 for unconditional branch
9. RS, WC and COM bits in PSW are also affected
10. CC assumes number in register is a binary number

A6b

2650 Instruction set

ADDRESSING MODES



PROGRAM STATUS WORD

PSU

7	6	5	4	3	2	1	0
S	F	II	Not Used	Not Used	SP2	SP1	SP0

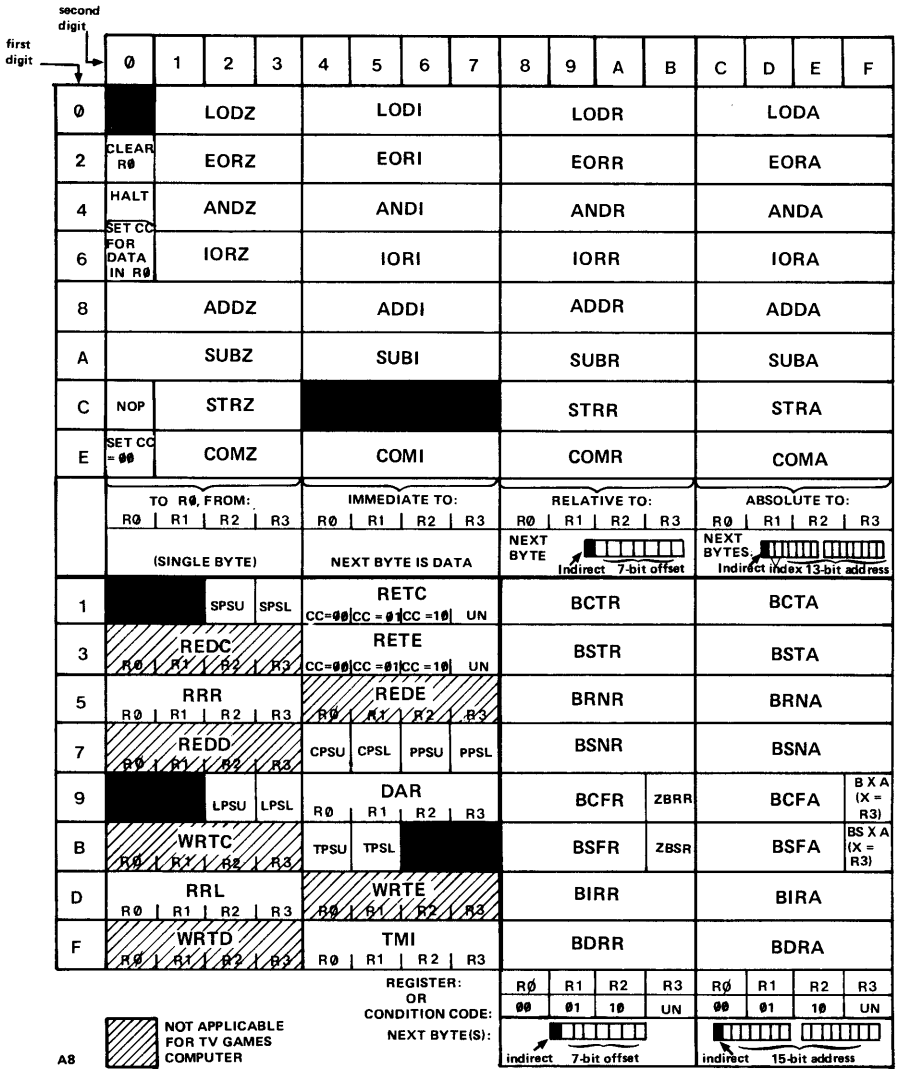
S Sense
F Flag
II Interrupt Inhibit
SP2 Stack Pointer Two
SP1 Stack Pointer One
SP0 Stack Pointer Zero

PSL

7	6	5	4	3	2	1	0
CC1	CC0	IDC	RS	WC	OVF	COM	C

CC1 Condition Code One
CC0 Condition Code Zero
IDC Interdigit Carry
RS Register Bank Select
WC With/Without Carry
OVF Overflow
COM Logical/Arith. Compare
C Carry/Borrow

A7


second digit
first digit
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 LODZ LODI LODR LODA
2 CLEAR R0 EORZ EORI EORR EORA
4 HALT ANDZ ANDI ANDR ANDA
6 SET CC FOR DATA IN R0 IORZ IORI IORR IORA
8 ADDZ ADDI ADDR ADDA
A SUBZ SUBI SUBR SUBA
C NOP STRZ STRR STRA
E SET CC = 00 COMZ COMI COMR COMA
TO R0, FROM: R0 R1 R2 R3
IMMEDIATE TO: R0 R1 R2 R3
RELATIVE TO: R0 R1 R2 R3
ABSOLUTE TO: R0 R1 R2 R3
(SINGLE BYTE)
NEXT BYTE IS DATA
NEXT BYTE
Indirect 7-bit offset
NEXT BYTES
Indirect index 13-bit address
1 SPSU SPSL RETC CC=00 CC=01 CC=10 UN BCTR BCTA
3 REDC R0 R1 R2 R3 RETE CC=00 CC=01 CC=10 UN BSTR BSTA
5 RRR R0 R1 R2 R3 REDE R0 R1 R2 R3 BRNR BRNA
7 REDD R0 R1 R2 R3 CPSU CPSL PPSU PPSL BSNR BSNA
9 LPSU LPSL DAR R0 R1 R2 R3 BCFR ZBRR BCFA BXA (X = R3)
B WRTC R0 R1 R2 R3 TPSU TPSL BSFR ZBSR BSFA BSXA (X = R3)
D RRL R0 R1 R2 R3 WRTE R0 R1 R2 R3 BIRR BIRA
F WRTD R0 R1 R2 R3 TMI R0 R1 R2 R3 BDRR BDRA
REGISTER: R0 R1 R2 R3 R0 R1 R2 R3
OR CONDITION CODE: 00 01 10 UN 00 01 10 UN
NEXT BYTE(S): indirect 7-bit offset indirect 15-bit address
NOT APPLICABLE FOR TV GAMES COMPUTER
A8

Appendix B:

- **Keyboard scan routine**
- **Text routine**
- **Interrupt routines**
- **Joystick calibration routines**
- **Random numbers routines**

Keyboard scan routine

```
ØFDØ    2Ø          EORZ, RØ
ØFD1    CCØ89F      STRA, RØ      } presets for keyscan
ØFD4    7712        PPSL, RS, COM
ØFD6    75Ø9        CPSL, WC, C
ØFD8    ►ØCIFCB     LODA, RØ
ØFDB    F44Ø        TMI, RØ       } wait for VRLE
ØFDD    └9879       BCFR
ØFDF    3FØ183      BSTA, UN      } Gosub 'Keyscan
ØFE2    F54Ø        TMI, R1       } repeat if
ØFE4    └1872       BCTR          } 1st scan
ØFE6    Ø1          LODZ, R1
ØFE7    ┌1A05       BCTR
ØFE9    Ø43Ø        LODI, RØ      } load '3Ø'
ØFEB    751Ø        CPSL, RS      } if no key
ØFED    17          RETC, UN      } else load
ØFEE    └►451F      ANDI, R1      } translated code
ØFFØ    ØD6122      LODA, I/R1    } and return
ØFF3    751Ø        CPSL, RS
ØFF5    17          RETC, UN
ØFF6    ►3B58       BSTR, UN      } (keyscan
ØFF8    F43Ø        TMI, RØ       } (no key)
ØFFA    └987A       BCFR
ØFFC    17          RETC, UN      } return
```

keyboard scan and decode

wait for key release, then return

Registers used: RØ, R1', R2', R3';
Subroutine levels used: 2 for 'keyboard scan',
3 for 'wait for key release'.

Data in RØ at ØFE6:

system keys	Left-hand keyboard			Right-hand keyboard		
UC ØF	RCAS Ø3	WCAS Ø7	C ØB	D 13	E 17	F 1B
STRT ØE	BP Ø2	REG Ø6	8 ØA	9 12	A 16	B 1A
LC ØD	PC Ø1	MEM Ø5	4 Ø9	5 11	6 15	7 19
RESET * ØC	– ØØ	+ Ø4	Ø Ø8	1 1Ø	2 14	3 18

*note that this code is only obtained if this key is wired as part of the normal keyboard – not if it is wired direct to the reset input, as in the suggested keyboard layout.

Data in RØ at ØFED/ØFF5:

system keys	Left-hand keyboard			Right-hand keyboard		
UC 8Ø	RCAS 9Ø	WCAS 93	C ØC	D ØD	E ØE	F ØF
STRT 8A	BP 84	REG 87	8 Ø8	9 Ø9	A ØA	B ØB
LC 8Ø	PC 8D	MEM 81	4 Ø4	5 Ø5	6 Ø6	7 Ø7
RESET 8Ø	– CØ	+ E0	Ø ØØ	1 Ø1	2 Ø2	3 Ø3

3Ø = no key operated

Monitor character set:

character	code	character	code	character	code	character	code
0	Ø0	A	ØA	P	14	?	5F
1	Ø1	b	ØB	r	15	. .	8A
2	Ø2	C	ØC	=	16	n (1)	AA
3	Ø3	d	ØD	space	17	I	BB
4	Ø4	E	ØE	+	18	T	BC
5	Ø5	F	ØF	–	19	I	DF
6	Ø6	G	1Ø	:	1A	: (2)	E6
7	Ø7	L	11	x	1B	•	F7
8	Ø8	I	12			! (3)	A2
9	Ø9	n	13				

Notes:
(1) this n is slightly larger than the 'official' version (code 13), and looks better between capitals.
(2) similarly, this colon is larger than that obtained by code 1A, which can be useful.
(3) the exclamation mark is too small, actually, but no better version exists . . .
(4) the Ø (code Ø0) can be used as the letter O; similarly, a 5 makes a good S and a 2 will pass for a Z.

Demonstration routine:

```
0900          7620      PPSU, II
0902          3F0161    BSTA, UN      (clear/initiate PVI)
0905          072A      LODI, R3
0907          0507      LODI, R1
0909          3F02D9    BSTA, UN      (load 8 spaces)
090C          0F4930    LODA, I-R3    (messline data)
090F          CD4890    STRA, I-R1
0912          5978      BRNR, R1
0914          7710      PPSL, RS
0916          3F020E    BSTA, UN      (load Mline)
0919          7510      CPSL, RS
091B          5B0A      BRNR, R3
091D          0C1E89    LODA, R0
0920          F410      TMI, R0       wait for '+' key release
0922          1879      BCTR
0924          1F095A    BCTA, UN
0927          7710      PPSL, RS
0929          3F02CF    BSTA, UN      (scroll)
092C          7510      CPSL, RS
092E          1B57      BCTR, UN

0930          5F A2 17 8A 17 E6 F7 sixth line
0937          02 16 17 18 19 1A 1B fifth line
093E          AA 13 00 14 15 05 BC fourth line      DATA
0945          0E 0F 10 12 DF 11 BB third line
094C          07 08 09 0A 0B 0C 0D second line
0953          00 01 02 03 04 05 06 first line

095A          0C1FCB    LODA, R0
095D          F440      TMI, R0       wait for VRLE
095F          9879      BCFR
0961          0C1E88    LODA, R0
0964          F420      TMI, R0       return to monitor if 'PC'
0966          1C0000    BCTA
0969          7702      PPSL, COM
096B          3F0055    BCTA, UN      display 6 lines
096E          1B6A      BCTR, UN
```

Start address: 0900

Note: A complete alphabet can be obtained with the extended routine described in chapter 17.

Interrupt routines

A simple routine, using the upper register bank for interrupts:

```
0900    1Fxxxx    Jump to main program
0903    C808      STRR,R0  ) Preserve R0 and
   5    13        SPSL     ) PSL, select upper
   6    C809      STRR,R0  ) register bank.
   8    7710      PPSL, RS )
   A  ┌─3B07      BSTR, UN
   C  │ 0400      LODI, R0 ) Restore R0 and
   E  │ 75FF      CPSL     ) PSL – the latter
0910  │ 7700      PPSL     ) selects the lower
   2  │ 37        RETE,UN  ) register bank.
0913  └─►....      Interrupt routine starts here. It can be
                  terminated at any point with 14, 15, 16 or 17 (RETC).
```

Note: this system requires a minimum of two subroutine levels for the interrupt routine.

The main program typically starts as follows:

```
        7620      PPSU,II
        0402      LODI,R0  ) select lower
        93        LPSU     ) register bank
        ....               ) initialisation
        ....               ) procedure
        7420      CPSU,II
        ....               ) start of main
        ....               ) program loop
```

Both register banks available for main program and interrupt routines:

```
0900    1Fxxxx    Jump to main program
0903    C82C      STRR,R0
   5    13        SPSL
   6    C82D      STRR,R0
   8    12        SPSU
   9    C823      STRR,R0
   B    7510      CPSL, RS
   D    C911      STRR'R1
   F    CA11      STRR,R2
0911    CB11      STRR,R3
   3    7710      PPSL, RS
   5    C911      STRR,R1
   7    CA11      STRR,R2
   9    CB11      STRR,R3
   B  ┌─3B1A      BSTR, UN
   D  │ 7510      CPSL, RS
   F  │ 0500      LODI, R1
0921  │ 0600      LODI, R2
   3  │ 0700      LODI, R3
   5  │ 7710      PPSL, RS
   7  │ 0500      LODI, R1
   9  │ 0600      LODI, R2
   B  │ 0700      LODI, R3
   D  │ 0400      LODI, R0
   F  │ 92        LPSU
0930  │ 0400      LODI, R0
   2  │ 75FF      CPSL
   4  │ 7700      PPSL
   6  │ 37        RETE, UN
0937  └─►....      Interrupt routine starts here.
```

Both register banks protected; uses monitor routines:

```
0900    1Fxxxx    Jump to main program
0903    CC08AC    STRA, R0
   6    13        SPSL
   7    CC08B3    STRA, R0
   A    12        SPSU
   B    CC08B4    STRA, R0
   E    7510      CPSL, RS
0910    CD08AD    STRA, R1
   3    CE08AE    STRA, R2
   6    CF08AF    STRA, R3
   9    7710      PPSL, RS
   B    CD08B0    STRA, R1
   E    CE08B1    STRA, R2
0921    CF08B2    STRA, R3
   4  ┌─3B09      BSTR, UN
   6  │ 042E      LODI, R0
   8  │ CC08BF    STRA, R0
   B  │ 1F0562    BCTA, UN
   E  │ 37        RETE, UN
092F  └─►....     Interrupt routine starts here.
```

Both register banks protected; monitor routines used; must be located from 08C0 on:

```
08C0    C86A      STRR, R0
   2    13        SPSL
   3    C86E      STRR, R0
   5    12        SPSU
   6    C86C      STRR, R0
   8    7510      CPSL, RS
   A    C961      STRR, R1
   C    CA60      STRR, R2
   E    CB5F      STRR, R3
08D0    7710      PPSL, RS
   2    C95C      STRR, R1
   4    CA5B      STRR, R2
   6    CB5A      STRR, R3
   8    3Fxxxx    BSTA, UN to interrupt routine
   B    0437      LODI, R0
   D    C85E      STRR, R0
   F    1F056C    BCTA, UN
        ....
        ....
0900    1Fxxxx    Jump to main program
0903    1F08C0    Jump to interrupt routine
```

Joystick calibration

'Switchpoint' calibration, both joysticks:

```
08C0  ┌►0C1FCB   LODA,R0  }
   3  │  D0      RRL,  R0  } subroutine
   4  └─9A7A     BCFR      } wait for VRLE
   6     17      RETC, UN  }
08C7     7660    PPSU, II + flag
   9     7518    CPSL, RS + WC
   B     07FF    LODI, R3
   D     3B71    BSTR, UN
   F     3B6F    BSTR, UN
08D1  ┌►3B6D     BSTR, UN
   3  │  0602    LODI, R2
   5  │►0E5FCC   LODA,I-R2
   8  │  C1      STRZ, R1
   9  │  51      RRR,  R1
   A  │  51      RRR,  R1
   B  │  453F    ANDI, R1
   D  │  A1      SUBZ, R1
   E  │  CF28EF  STRA, I + R3
08E1  │  81      ADDZ, R1
   2  │  81      ADDZ, R1
   3  │  CF28EF  STRA, I + R3
   6  └─5A6D     BRNR, R2
   8     B440    TPSU, flag
   A     16      RETC
   B     7440    CPSU, flag
   D     1B62    BCTR, UN

08EF    00   00      00   00     vertical data
        up   down    up   down
             Right        Left
08F3    00   00      00   00     horizontal data
        up   down    up   down
             Right        Left
```

Note: this joystick calibration routine can be located at any other point in memory, provided the data at addresses 08DE and 08E3 are modified to correspond to the desired 'switchpoint data' locations.

The joystick calibration routine places 'switchpoint data' in addresses 08EF...08F6. To use this data in a program, the flag must be set and reset on alternate frames (for both horizontal and vertical joystick scan), and the data found at frame-end must be compared with the calculated values. The layout for a typical 'joystick scan' routine is therefore as follows:

```
3F08C0    BSTA, UN    Wait for VRLE
12        SPSU        } set/reset flag on
2440      EORI, R0    } alternate frames
92        LPSU        }
0E1FCC    LODA, R2    left-hand joystick data
0F1FCD    LODA, R3    right-hand joystick data
7702      PPSL, COM
B440      TPSU, flag  } branch for horizontal routines
18..      BCTR        } if flag now set
EE08F1    COMA, R2
9A..      BCFR
...                   left-hand joystick up routine
EE08F2    COMA, R2
99..      BCFR
...                   left-hand joystick down routine
EF08EF    COMA, R3
9A..      BCFR
...                   right-hand joystick up routine
etc.
```

Joystick calibration

Demonstration of joystick calibration and scan routines:

```
0900    3F08C7   BSTA, UN                calibration routine
   3    0444     LODI, R0          ┐
   5    CC1F0A   STRA, R0          │
   8    0455     LODI, R0          │
   A    CC1F1A   STRA, R0          │ initial
   D    0477     LODI, R0          │ positions
   F    CC1F0C   STRA, R0          │
0912    CC1F1C   STRA, R0          ┘
   5    3F08C0   BSTA, UN                wait for VRLE
   8    12       SPSU              ┐
   9    2440     EORI, R0          │ set/reset flag
   B    92       LPSU              ┘ on alternate frames
   C    0E1FCC   LODA,R2                  left-hand joystick data
   F    0F1FCD   LODA,R3                  right-hand joystick data
0922    7702     PPSL, COM
   4    B440     TPSU, flag        ┐ branch for horizontal
   6    ┌1829    BCTR              ┘ routines if flag now set
   8    │0C1F0C  LODA,R0                   vertical position object 1
   B    │0D1F1C  LODA,R1                   vertical position object 2
   E    │EE08F1  COMA,R2            ┐
0931    ┌9A02    BCFR               │
   3    │A401    SUBI, R0           │
   5    └►EE08F2 COMA,R2            │
   8    ┌9902    BCFR               │
   A    │8401    ADDI, R0           │
   C    └►EF08EF COMA,R3            │
   F    ┌9A02    BCFR               │ vertical movement
0941    │A501    SUBI, R1           │
   3    └►EF08F0 COMA,R3            │
   6    ┌9902    BCFR               │
   8    │8501    ADDI, R1           │
   A    └►C8DD   STRR, R0 Ind (1F0C)│
   C     C9DE    STRR, R1 Ind (1F1C)┘
   E     1F0915  BCTA, UN
0951    ►0C1F0A  LODA, R0                 horizontal position object 1
   4     0D1F1A  LODA, R1                 horizontal position object 2
   7     EE08F5  COMA,R2           ┐
   A    ┌9A02    BCFR               │
   C    │A401    SUBI, R0           │
   E    └►EE08F6 COMA,R2            │
0961    ┌9902    BCFR               │
   3    │8401    ADDI, R0           │
   5    └►EF08F3 COMA,R3            │
   8    ┌9A02    BCFR               │ horizontal movement
   A    │A501    SUBI, R1           │
   C    └►EF08F4 COMA,R3            │
   F    ┌9902    BCFR               │
0971    │8501    ADDI, R1           │
   3    └►C8DD   STRR, R0 Ind (1F0A)│
   5     C9DE    STRR, R1 Ind (1F1A)│
   7     1F0915  BCTA, UN           ┘
```

Simplified joystick scan.

This joystick calibration routine is a simplified version of the general procedure given earlier. It calibrates the left-hand joystick only.

```
08C0 ┌►0C1FCB   LODA, R0  ┐
   3 │  D0       RRL,  R0  │ Subroutine:
   4 └─9A7A     BCFR      │ wait for VRLE
   6    17      RETC, UN  ┘
08C7    7660    PPSU, II + flag
   9    7518    CPSL, RS + WC
   B    07FF    LODI, R3
   D    3B71    BSTR, UN
   F    3B6F    BSTR, UN
08D1 ┌►3B6D     BSTR, UN
   3 │  0C1FCC   LODA, R0
   6 │  C1       STRZ, R1
   7 │  51       RRR,  R1
   8 │  51       RRR,  R1
   9 │  453F     ANDI, R1
   B │  A1       SUBZ, R1
   C │  CF28EB   STRA, I + R3
   F │  81       ADDZ, R1
08E0 │  81       ADDZ, R1
   1 │  CF28EB   STRA, I + R3
   4 │  B440     TPSU, flag
   6 │  16       RETC
   7 │  7440     CPSU, flag
   9 └─1B66     BCTR, UN

08EB    00   00   00   00      Left-hand joystick data
        up   down left right
             ~~~~ ~~~~~
             vert horiz
```

Joystick calibration

Simplified joystick scan

This joystick calibration routine is also a variation on the general routine given earlier. It calibrates the left-hand joystick, and stores the results in the main program. The main restriction is that the upper address byte (indicated by '–' at addresses Ø8DA and Ø8DF) must be the same for all four data values. The lower address byte is indicated by 'xx' and 'yy' for the vertical upper and lower switchpoints, and by 'x́x́' and 'ý ý' for horizontal left and right.

```
Ø8CØ  ┌►ØC1FCB   LODA,RØ  ┐
   3  │  DØ       RRL,  RØ  │ Subroutine:
   4  └─9A7A      BCFR      │ wait for VRLE
   6     17       RETC, UN  ┘
Ø8C7     766Ø    PPSU, II + flag
   9     7518    CPSL, RS + WC
   B     3B73    BSTR, UN
   D     3B71    BSTR, UN
   F  ┌►3B6F     BSTR, UN
Ø8D1  │  ØC1FCC  LODA,RØ
   4  │  C1      STRZ, R1
   5  │  51      RRR,  R1
   6  │  51      RRR,  R1
   7  │  453F    ANDI, R1
   9  │  A1      SUBZ, R1
   A  │  CC — — xx  STRA, RØ
   D  │  81      ADDZ,R1
   E  │  81      ADDZ,R1
   F  │  CC — — yy  STRA, RØ
Ø8E2  │  B44Ø    TPSU, flag
   4  │  16      RETC
   5  │  744Ø    CPSU, flag
   7  │  Ø4x́x́   LODI, RØ
   9  │  C871    STRR, RØ
   B  │  Ø4ý ý   LODI, RØ
   D  │  C872    STRR, RØ
   F  └─1B5E     BCTR, UN
```

Joystick calibration for analog use:

```
Ø8CØ    77Ø2      PPSL, COM
   2    7518      CPSL, RS/WC
   4    2Ø        EORZ,RØ
   5    C2        STRZ, R2
   6    ØB9F      LODR,R3 Ind (1FCC)
   8  ┌►E76Ø      COMI, R3
   A  │ 19Ø3      BCTR
   C  │ D3        RRL,  R3
   D  └ D879      BIRR, RØ
   F    ►C819     STRR, RØ (Ø8EA)
Ø8D1    47FE      ANDI, R3
   3    53        RRR,  R3
   4    Ø3        LODZ, R3
   5  ┌►668Ø      IORI, R2
   7  ├►D2        RRL,  R2
   8  │ CA1A      STRR, R2 (Ø8F4)
   A  │ 16        RETC
   B  │ 47FE      ANDI, R3
   D  │ 53        RRR,  R3
   E  │ 83        ADDZ,R3
   F  │ E46Ø      COMI, RØ
Ø8E1  └ 9972      BCFR
   3    A3        SUBZ, R3
   4    1B71      BCTR, UN

Ø8E6    ØC1FCC    LODA,RØ
   9    Ø5ØØ      LODI, R1
   B    44FE      ANDI, RØ
   D  ┌ 18Ø3      BCTR
   F  ├►DØ        RRL,  RØ
Ø8FØ  └ F97D      BDRR,R1
   2    ►5Ø       RRR,  RØ
   3    Ø5ØØ      LODI, R1
   5    C2        STRZ, R2
   6  ┌►46FE      ANDI, R2
   8  │ 14        RETC
   9  │ 52        RRR,  R2
   A  │ D1        RRL,  R1
   B  │ 9A79      BCFR
   D  │ 82        ADDZ,R2
   E  └ 1B76      BCTR, UN
```

During the initialisation routine at the start of the program, the calibration routine from Ø8CØ must be run. When a joystick scan is required during the program, the calculation routine from Ø8E6 on is used, at frame-end; this reads the joystick value and converts it into the desired range. The conversion factor is determined during the initial calibration.

Note: When calibrating joysticks for vertical operation (flag set), the instructions at addresses Ø8C8 and Ø8DF can be modified to E77F and E47F respectively. Larger values than 7F should not be used for these comparisons!

Random numbers

Random, non-zero number in 1FAD:

```
..00     0C1FAD  LODA,R0
   3    ┌9802    BCFR
   5    │ 0401   LODI, R0
   7    └►C1     STRZ, R1
   8      C2     STRZ, R2
   9      0703   LODI, R3
   B      52     RRR, R2
   C      52     RRR, R2
   D      52     RRR, R2
   E    ┌►52     RRR, R2
   F    │ 02     LODZ,R2
..10    │ 4401   ANDI, R0
   2    │ 21     EORZ,R1
   3    │ C1     STRZ, R1
   4    └─FB78   BDRR,R3
   6      51     RRR, R1
   7      C9E8   STRR, R1  Ind (1FAD)
   9      17     RETC, UN
```

Random number, from 00 to FF, in 1FAD:

```
..00     0C1FAD  LODA,R0
   3      C1     STRZ, R1
   4      C2     STRZ, R2
   5      0703   LODI, R3
   7      52     RRR, R2
   8      52     RRR, R2
   9      52     RRR, R2
   A    ┌►52     RRR, R2
   B    │ 02     LODZ,R2
   C    │ 4401   ANDI, R0
   E    │ 21     EORZ,R1
   F    │ C1     STRZ, R1
..10    └─FB78   BDRR,R3
   2    ┌─9802   BCFR
   4    │ 0501   LODI, R1
   6    └►E401   COMI, R0
   8    ┌─9802   BCFR
   A    │ 0500   LODI, R1
   C    └►51     RRR, R1
   D      C9E2   STRR, R1  Ind (1FAD)
   F      17     RETC, UN
```

Appendix C:

- **The monitor program**

Useful monitor routines

Routine description	Presets, initial data; comments	Start address	Registers used R0	R1	R2	R3
keyboard scan	clear 089F; see chapter 11	0181	x	x	x	
translate key codes	451F, 0D6122; see figure 29	—	x	x		
clear duplicates	VOD 1 . . . 4 is 'FE'	009E	x			
load VOD duplicates	data in R0	00A0	x			
clear objects	1F00 . . . 1F4D is '00'	016E	x		x	
load objects	data in R0 to 1F00 . . . 1F4D	016F	x		x	
split register	R1 = XY; then R0 = 0X, R1 = 0Y	035E	x	x		
increment double byte	data in 08A4, 08A5	039F	x	x		
delay	delay is (R3 data) x 200 µs	06F7		x		x
assemble data	clear 08A7; enter successive digits (00 . . . 0F) in 08A1; result shifts left in 08A2 (= R2) and 08A3 (= R1)	031D	x	x	x	x
Text routines:						
initiate PVI		0161	x	x	x	
display six lines	set COM bit; wait for VRLE	0055	x	x	x	
display last line only	set COM bit, VC objects, R1 = 78, R2 = 00	007A	x	x	x	
translate MLINE to display	see Table 17 for codes	020E	x	x	x	x
load 8 spaces to MLINE		02D9	x		x	
load spaces to MLINE	first position in R2, number of spaces in R1	0379	x	x	x	
scroll, 8 spaces to MLINE		02CF	x	x	x	
output single byte to MLINE	data in R1, position in R3	0354	x	x		x
output single byte to screen	data in R1, position in R3	0480	x	x		x
output double byte to MLINE	data in R2/R1, position in R3	0350	x	x	x	x
output double byte to MLINE	data in R2/R1, last positions	034E	x	x	x	x
output double byte to screen	data in R1/R2; last positions	0529	x	x	x	x
output three bytes to screen	data in 08A4, 08A5 to first positions; 08A3 to right	0470	x	x	x	x
address + data to MLINE	address in 08A4, 08A5	042B	x	x	x	x
output message to MLINE	R2 is message code (note 1)	02EA	x		x	x
scroll, message to MLINE	R2 is message code (note 1)	02E3	x	x	x	x
scroll, message to screen	R2 is message code (note 1)	0602	x	x	x	x
assemble data on screen	see 'assemble data', above	052F	x	x	x	x

Note 1: message code data:
04: 'PC='
08: 'AD='
0C: 'R='
10: 'BP1='
14: 'BP2='
18: 'BEG='
IC: 'END='
20: 'SAD='
24: 'FIL='
28: 'IIII'

Use of monitor scratch (RAM area 0800 . . . 08BF)

0800 . . . 0817	shape data, first text line
0818 . . . 082F	shape data, second text line
0830 . . . 0847	shape data, third text line
0848 . . . 085F	shape data, fourth text line
0860 . . . 0877	shape data, fifth text line
0878 . . . 088F	shape data, sixth text line
0890 . . . 0897	'monitor line': eight character codes for display
0898	BP function indicator/RCAS silence counter
0899	+/– enter key memory
089A	monitor function index
089B, 089C	scratch for keyboard routine
089D	right keyboard status
089E	left keyboard status
089F	combined keyboard status
08A0 . . . 08AB	function scratch, used by memory and data routines
08AC . . . 08B4	save registers, PSU and PSL area
08B5, 08B6	breakpoint address 1
08B7, 08B8	breakpoint address 2
08B9 . . . 08BF	startup program (start address at 08BE, 08BF)

Abbreviations used in the monitor listing

Condition codes:

P	positive result
Z	zero result
N	negative result
LT	less than
EQ	equals
GT	greater than
UN	unconditional
A1	all selected bits are one
N1	not all selected bits are one

Note that the following survey of the monitor program is given as information to TV Games computer owners only. It is reproduced here by courtesy of NV Philips Gloeilampenfabrieken; the contents are not to be reproduced, in whole or in part, nor disclosed in any form without their written consent. The monitor routine is available in ROM, via normal retail outlets. No third party has been authorised to copy and distribute this program in EPROM, at the time of printing.

The monitor program

```
LOC  OBJECT   ADDR  SOURCE  LINE

0000 1F000A   000A  RESET   BCTA,UN START

                              *INTERRUPT ADDRESS H'0003'
0003 1F88B9   08B9  INTRR   BCTA,UN INTADR VIA RAM ADDRESS

                              *VECTORS FOR BREAKPOINT ENTRY
0006 0594           BVEC1   BREAK 1
0008 05B2           BVEC2   BREAK 2

                              *AT RESET THE SYSTEM ENTERS THE MONITOR
                              *INITIATION IS DONE AFTER POWER UP OR IF
                              *RESET + INITIATE KEY

000A 7620           START   PPSU II
000C 0C08B9   08B9          LODA,R0 INTADR    WAS USER ACTIVE?
000F E409                   COMI,R0 H'09'
0011 3C05CD   05CD          BSTA,EQ BRKSBR    THEN SAVE STATUS AND CLR BREAKPOINTS
0014 20                     EORZ,R0
0015 CC089A   089A          STRA,R0 MFUNC     SET NO FUNCTION
0018 0518                   LODI,R1 24                  LAST LINE OF MON. DISPLAY MEM
001A 6D4878   0878          IORA,R0 MONOB+(NLINES-1) *24,R1,-  IS TESTED FOR POWER UP
001D 597B     001A          BRNR,R1 $-3
001F 4411                   ANDI,R0 H'11'     THESE POSITIONS ARE ZERO IF RUNNING
0021 1815     0038          BCTR,Z START1
0023 7660           INIT0   PPSU FLAG+II      INITIATE SYSTEM STATUS
0025 20                     EORZ R0
0026 05C0                   LODI,R1 >(PC-MONOB+2)
0028 CD5F00   1F00  INIT1   STRA,R0 RAM,R1,-  INITIATE PVI
002B CD6800   0800          STRA,R0 MONOB,R1  INITIATE SCRATCH
002E 5978     0028          BRNR,R1, INIT1
0030 0628                   LODI,R2 40        MESSAGE "IIII"
0032 3F0602   0602          BSTA,UN WCAS3
0035 3F0161   0161          BSTA,UN SAVE1     INITIATE FOR MONITOR DISPLAY
0038 044B           START1  LODI,R0 > MONINT  SET MONITOR INT ADDRESS
003A CC08BA   08BA          STRA,R0 INTADR+1
003D 0400                   LODI,R0 < MONINT
003F CC08B9   08B9          STRA,R0 INTADR
0042 CC089F   089F          STRA,R0 MKBST     ASK KEY ENTRY
0045 7702           MWAIT0  PPSL COM
0047 7427           MWAIT1  CPSU II+SP        CLEAR II
0049 1B7C     0047          BCTR,UN MWAIT1    WAITING FOR INTERRUPTS
```

```
LOC  OBJECT   ADDR  SOURCE  LINE

                            *ENTRY FOR MONITOR INTERRUPTS VIA RAM ADDRESS

004B B480           MONINT  TPSU SENS          ACCEPT ONLY VERTICAL MON. INTERRUPTS
004D 16                     RETC,N1

                            *MONITOR SCREEN REFRESH

004E 3F0181   0181    FREFR0  BSTA,UN KBSCAN
0051 01                     LODZ R1             MKBST
0052 1E00BD   00BD          BCTA,N MKEY         IF KEY ENTRY
0055 0504                   LODI,R1 4           X,XD  Y,YD
0057 0D40AD   00AD    FREF1   LODA,R0 MPOBJ1,R1,-     SET OBJECT POSITIONS
005A CD7F0A   1F0A          STRA,R0 OBJ1+10,R1
005D 0D60B1   00B1          LODA,R0 MPOBJ2,R1
0060 CD7F1A   1F1A          STRA,R0 OBJ2+10,R1
0063 0D60B5   00B5          LODA,R0 MPOBJ3,R1
0066 CD7F2A   1F2A          STRA,R0 OBJ3+10,R1
0069 0D60B9   00B9          LODA,R0 MPOBJ4,R1
006C CD7F4A   1F4A          STRA,R0 OBJ4+10,R1
006F 5966     0057          BRNR,R1 FREF1
                            *(R1) = 0,          INDEX FOR STORE OBJECTS
0071 B480           FILELN  TPSU SENS           WAIT UNTIL END OF VERTICAL RESET
0073 187C     0071          BCTR,A1 $-2
0075 045C                   LODI,R0 MTOPDL      WAIT FOR TOP OF SCREEN
0077 F87E     0077  FREFR1  BDRR,R0 $
0079 C2                     STRZ R2             INDEX FOR STORE IN OBJECTS
007A 0D6800   0800  FREFR2  LODA,R0 MONOB,R1
007D CE7F00   1F00          STRA,R0 OBJ1,R2
0080 0D6806   0806          LODA,R0 MONOB+6,R1
0083 CE7F10   1F10          STRA,R0 OBJ2,R2
0086 0D680C   080C          LODA,R0 MONOB+12,R1
0089 CE7F20   1F20          STRA,R0 OBJ3,R2
008C 0D2811   0811          LODA,R0 MONOB+18-1,R1,+
008F CE3F3F   1F3F          STRA,R0 OBJ4-1,R2,+
0092 E606                   COMI,R2 6           OBJECTS OF ROW COMPLETED?
0094 9864     007A          BCFR,EQ FREFR2      IF NOT NEXT BYTE
0096 04A0                   LODI,R0 OBJDL       DELAY BETWEEN OBJECT REFRESHES
0098 8512                   ADDI,R1 18          PREPARE INDEX FOR NEXT REFRESH
009A E58F                   COMI,R1 24*NLINES-1       END OF SCREEN?
009C 1A59     0077          BCTR,LT FREFR1      IF NOT END
009E 04FE                   LODI,R0 H'FE'
00A0 CC1F0D   1F0D          STRA,R0 YD1         OBJ OFFSETS ARE SET TO H'FE' TO DISABLE
00A3 CC1F1D   1F1D          STRA,R0 YD2
00A6 CC1F2D   1F2D          STRA,R0 YD3
00A9 CC1F4D   1F4D          STRA,R0 YD4
00AC 17                     RETC,UN
                            *PVI PARAMETERS DURING MONITOR DISPLAY
00AD 1F1F10FF       MPOBJ1  DATA H'1F,1F', LINPOS, H'FF'        MONITOR PARAMETERS
00B1 3F3F10FF       MPOBJ2  DATA H'3F,3F', LINPOS, H'FF'        FOR THE OBJECTS
00B5 5F5F10FF       MPOBJ3  DATA H'5F,5F', LINPOS, H'FF'
00B9 7F7F10FF       MPOBJ4  DATA H'7F,7F', LINPOS, H'FF'
```

```
LOC  OBJECT   ADDR  SOURCE  LINE
                            *VERTICAL RESET WITH KEY ENTRY
00BD 0F089A   089A  MKEY    LODA,R3 MFUNC     FUNCTION CODE
00C0 451F                   ANDI,R1 H'1F'     KEY CODE ONLY
00C2 0D6122   0122          LODA,R0 KBFTBL,R1 TRANSLATE KEY CODE
00C5 1A17     00DE          BCTR,N MKEY3      IF FUNCTION KEY
00C7 CC08A1   08A1          STRA,R0 FSCRM+1   NEW HEX CHAR
00CA 3F016E   016E  MKEY1   BSTA,UN CLROBJ    NO DISPLAY
00CD 0E08A0   08A0  MKEY2   LODA,R2 FSCRM     FUNCTION STATUS
00D0 60                     IORZ R0           TO SET CC
00D1 BF010C   010C          BSXA FNOP,R3      FUNCTION EXECUTION
00D4 047F           MKEYR   LODI,R0 H'7F'     ENABLE NEW KEYBOARD ENTRY
00D6 4C089F   089F          ANDA,R0 MKBST
00D9 C8FC     00D7          STRR,R0 *$-2
00DB 1F0045   0045          BCTA,UN MWAIT0    WAIT LOOP

00DE F440           MKEY3   TMI,R0 H'40'      TEST FOR +/-
00E0 181B     00FD          BCTR,A1 MKEY6     IF + OR -
00E2 441F                   ANDI,R0 H'1F'     FUNCTION INDEX ONLY
00E4 C8D8     00BE          STRR,R0 *MKEY+1   MFUNC
00E6 05FF                   LODI,R1 H'FF'     IF 1ST ENTRY
00E8 E3                     COMZ R3           RE-ENTRY?
00E9 9804     00EF          BCFR,EQ MKEY5     IF 1ST ENTRY
00EB 0983     00F0          LODR,R1 *MKEY5+1  FSEQ FUNCTION SEQUENCE INDICATOR
00ED 257F                   EORI,R1 H'7F'     TO SWITCH BK1/2
00EF CD0898   0898  MKEY5   STRA,R1 FSEQ
00F2 C3                     STRZ R3           NEW COMMAND INDEX
00F3 20                     EORZ R0
00F4 0609                   LODI,R2 9         CLEAR FUNCTION SCRATCH
00F6 CE48A0   08A0          STRA,R0 FSCRM,R2,-
00F9 5A7B     00F6          BRNR,R2 $-3
00FB 1B4D     00CA          BCTR,UN MKEY1

00FD C1             MKEY6   STRZ R1           KEEP ENTER CODE
00FE CC0899   0899          STRA,R0 ENTM
0101 E70D                   COMI,R3 FPC-FNOP
0103 3E016E   016E          BSTA,LT CLROBJ    IF NOT RCAS OR WCAS
0106 01                     LODZ R1           ENTER CODE
0107 0500                   LODI,R1 0         NO RE-ENTRY
0109 1F00CD   00CD          BCTA,UN MKEY2

                            *FUNCTION BRANCH TABLE

010C 17             FNOP    RETC,UN
010D 1F040C   040C  FMEM    BCTA,UN MEM0
0110 1F04A9   04A9  FBK     BCTA,UN BKSC
0113 1F03B0   03B0  FREG    BCTA,UN REG0
0116 1F0023   0023  FINIT   BCTA,UN INIT0
0119 1F050E   050E  FPC     BCTA,UN GO0
011C 1F0758   0758  FRC     BCTA,UN RCAS0
011F 1F05E8   05E8  FWC     BCTA,UN WCAS0
```

```
LOC  OBJECT   ADDR  SOURCE   LINE

                          *KEY CODE FUNCTION TABLE

0122 C0               KBFTBL  DATA H'C0'          "-" KEY
0123 8D                       DATA > (FPC-FNOP+H'80')
0124 84                       DATA > (FBK-FNOP+H'80')
0125 90                       DATA > (FRC-FNOP+H'80')
0126 E0                       DATA H'E0'          "+" KEY
0127 81                       DATA > (FMEM-FNOP+H'80')
0128 87                       DATA > (FREG-FNOP+H'80')
0129 93                       DATA > (FWC-FNOP+H'80')
012A 0004080C                 DATA H'00,04,08,0C'
012E 8080                     DATA H'80,80'       NOT USED: KEY IGNORED
0130 8A                       DATA > (FINIT-FNOP+H'80')
0131 80                       DATA H'80'          NOT USED: KEY IGNORED
0132 0105090D                 DATA H'01,05,09,0D'
0136 02060A0E                 DATA H'02,06,0A,0E'
013A 03070B0F                 DATA H'03,07,0B,0F'

                              *STATUS SAVE ROUTINE

013E CC08AC   08AC    SAVE    STRA,R0 REGM        SAVE 2650 REGISTERS
0141 13                       SPSL
0142 CC08B3   08B3            STRA,R0 REGM+7
0145 12                       SPSU
0146 7620                     PPSU II             DISABLE INTERRUPTS
0148 CC08B4   08B4            STRA,R0 REGM+8
014B 7510                     CPSL RS             BANK 0
014D CD08AD   08AD            STRA,R1 REGM+1
0150 CE08AE   08AE            STRA,R2 REGM+2
0153 CF08AF   08AF            STRA,R3 REGM+3
0156 7710                     PPSL RS             BANK 1
0158 CD08B0   08B0            STRA,R1 REGM+4
015B CE08B1   08B1            STRA,R2 REGM+5
015E CF08B2   08B2            STRA,R3 REGM+6
0161 050A             SAVE1   LODI,R1 10          PVI PARAMETERS
0163 0D4177   0177            LODA,R0 PVIPAR,R1,-
0166 CD7FC0   1FC0            STRA,R0 SIZE, R1
0169 5978     0163            BRNR,R1 $-6
016B CD1E80   1E80            STRA,R1 H'1E80'

                              *CLEAR PVI OBJECTS

016E 20               CLROBJ  EORZ R0
016F 0650                     LODI,R2 H'50'
0171 CE5F00   1F00            STRA,R0 OBJ1,R2,-
0174 5A7B     0171            BRNR,R2 $-3
0176 17                       RETC,UN

                              *PVI CONTROL PARAMETERS DURING MONITOR MODE

                              SIZE,OBCOL1/2,OBCOL3/4,FORM/POS,XX,XX,BGCTL,SND,SCOR1 2, SCOR3 4
0177 AA090900         PVIPAR  DATA H'AA,09,09,00,00,00,19,00,AA,AA'
```

```
LOC  OBJECT    ADDR  SOURCE   LINE

                           *KEYBOARD SCAN ROUTINE

                           *TO GET A KEYBOARD SCAN THE UPPER BIT OF THE
                           *KEYBOARD STATUS WORD MUST BE RESET

                           *THE KEYBOARD RETURNS WITH A NEW KEY ADDRESS IN THE
                           *LOWER FOUR (RKBST LKBST) OR FIVE (MKBST) BITS OF THE
                           *STATUS WORD. THE STATUS IS VALID IF THE SIGN BIT IS SET.

                           *REGISTERS USED:  R0 R1 R2 R3

                           *ENTRY:           KBSCAN
                           *INPUT LOC:       NONE
                           *OUTPUT LOC:      MKBST RKBST LKBST

0181 7519                  KBSCAN   CPSL C+WC+RS
0183 20                             EORZ R0
0184 C893      0199                 STRR,R0 IA01+1 MSCR
0186 C89C      01A4                 STRR,R0 IA04+1 MSCR+1   CLEAR 2 SCRATCH LOCATIONS
0188 0607                           LODI,R2 7               KB COLUMN INDEX
018A 0503                  KBSC0    LODI,R1 3               KB ROW INDEX
018C 0E5E88    1E88                 LODA,R0 KEYBRD,R2,-     READ KB COLUMN
018F 44F0                           ANDI,R0 H'F0'           REMOVE NONSIGNIFICANT BITS
0191 9830      01C3                 BCFR,Z KBSC2            IF KEY IN COL ACTIVE
0193 5A75      018A      KBSC12     BRNR,R2 KBSC0           END OF KB?
0195 0C089D    089D      IA00       LODA,R0 RKBST           RIGHT KB OLD STATUS
0198 0D089B    089B      IA01       LODA,R1 MSCR            RIGHT KB NEW
019B 3F01EE    01EE      IA02       BSTA,UN KBSBR           DETERMINE CODE
019E C9F6      0196                 STRR,R1 IA00+1 RKBST    UPDATE RIGHT KB STATUS
01A0 0C089E    089E      IA03       LODA,R0 LKBST           L KB OLD STATUS
01A3 0D089C    089C      IA04       LODA,R1 MSCR+1          L KB NEW
01A6 3BF4      019C                 BSTR,UN IA02+1 KBSBR
01A8 C9F7      01A1                 STRR,R1 IA03+1 LKBST
01AA 09ED      0199                 LODR,R1 IA01+1 MSCR     NEW R KB CODE
01AC 1808      01B6                 BCTR,Z KBSC5
01AE 08F4      01A4                 LODR,R0 IA04+1 MSCR+1   NEW L KB CODE
01B0 1806      01B8                 BCTR,Z KBSC6
01B2 05FF                           LODI,R1 H'FF'           DOUBLE KEY CODE
01B4 1B02      01B8                 BCTR,UN KBSC6
01B6 09EC      01A4      KBSC5      LODR,R1 IA04+1 MSCR+1   NEW L KB
01B8 0C089F    089F      KBSC6      LODA,R0 MKBST           OLD MON KB STATUS
01BB 3BDF      019C                 BSTR,UN IA02+1 KBSBR
01BD C9FA      01B9      KBSC7      STRR,R1 KBSC6+1 MKBST   UPDATE MON KB STATUS
01BF 17                             RETC,UN
```

```
LOC  OBJECT  ADDR  SOURCE  LINE

                           *KEYBOARD SCAN ROUTINE CONTINUED

01C0 F900    01C2  KBSC1   BDRR,R1 $+2        DECR ROW INDEX
01C2 D0                    RRL,R0             TEST NEXT KEY BIT
01C3 9A7B    01C0  KBSC2   BCFR,N KBSC1       IF NOT THIS BIT
01C5 447F                  ANDI,R0 H'7F'      THESE BITS MUST BE ZERO
01C7 9816    01DF          BCFR,Z KBSC10      OTHERWISE DOUBLE KEY ENTRY
01C9 02                    LODZ R2            FORM COMPLETE KEY ADDRESS
01CA D0                    RRL,R0
01CB D0                    RRL,R0
01CC 81                    ADDZ R1            PLUS ROW ADDRESS
01CD 6460                  IORI,R0 H'60'      SET RELEASE AND COUNT BIT
01CF E603          KBSC3   COMI,R2 3          RIGHT OR LEFT?
01D1 1910    01E3          BCTR,GT KBSC4      IF GT: RIGHT
01D3 0985    01DA          LODR,R1 KBSC8+1 MSCR+1 NEW L KB
01D5 1802    01D9          BCTR,Z KBSC8       MUST BE ZERO
01D7 04FF                  LODI,R0 H'FF'      OTHERWISE DOUBLE KEY
01D9 CC089C  089C  KBSC8   STRA,R0 MSCR+1     STORE IN SCRATCH
01DC 1F0193  0193  KBSC11  BCTA,UN KBSC12     TEST NEXT COLUMN
01DF 04FF          KBSC10  LODI,R0 H'FF'      DOUBLE KEY ERROR CODE
01E1 1B6C    01CF          BCTR,UN KBSC3
01E3 0985    01EA  KBSC4   LODR,R1 KBSC9+1 MSCR
01E5 1802    01E9          BCTR,Z KBSC9       MUST STILL BE ZERO
01E7 04FF                  LODI,R0 H'FF'      OTHERWISE DOUBLE KEY
01E9 CC089B  089B  KBSC9   STRA,R0 MSCR       STORE IN SCRATCH
01EC 1BEF    01DD          BCTR,UN KBSC11+1   TEST NEXT COLUMN
01EE 5904    01F4  KBSBR   BRNR,R1 KBSBR0     IF NEW CODE
01F0 449F                  ANDI,R0 H'9F'      CLR RELEASE & COUNT BITS
01F2 C1                    STRZ R1            KEEP OLD CODE
01F3 17                    RETC,UN
01F4 E5FF          KBSBR0  COMI,R1 -1         DOUBLE KEY CODE?
01F6 1803    01FB          BCTR,EQ KBSBR4
01F8 60                    LODZ R0
01F9 9A04    01FF          BCFR,N KBSBR1      IF NEW CODE WANTED
01FB 6420          KBSBR4  IORI,R0 H'20'      SET RELEASE BIT
01FD C1            KBSBR2  STRZ R1            CODE TO BE KEPT
01FE 17                    RETC,UN
01FF F420          KBSBR1  TMI,R0 H'20'       TEST RELEASE BIT
0201 16                    RETC,N1            IF ZERO ACCEPT NEW CODE
0202 F440                  TMI,R0 H'40'       TEST SCAN COUNT BIT
0204 9877    01FD          BCFR,A1 KBSBR2     IF NOT 2ND SCAN: OLD CODE
0206 E1                    COMZ R1            2ND SCAN MUST GIVE SAME
0207 9872    01FB          BCFR,EQ KBSBR4     RESULT AS 1ST SCAN
0209 6580                  IORI,R1 H'80'      SET KEY READY INDICATOR
020B 45BF                  ANDI,R1 H'BF'      RESET SCAN COUNT BIT
020D 17                    RETC,UN
```

```
LOC  OBJECT   ADDR  SOURCE  LINE

                            *LINE IMAGE ROUTINES


                            *MON. ROUTINE TO CONVERT AN 8 CHAR LINE TO THE
                            *SCREEN IMAGE MEMORY FIELD MONOB

020E 0508           LINE    LODI,R1 8           NUMBER OF CHAR. PER LINE
0210 7508                   CPSL WC
0212 F501           LINE2   TMI,R1 H'01'        TEST POSITION
0214 1808     021E          BCTR,A1 LINE07      IF NOT NEW PAIR
0216 0606                   LODI,R2 6
0218 20                     EORZ R0
0219 CE4878   0878          STRA,R0 MONOB+24 *(NLINES-1),R2,-
021C 5A7B     0219          BRNR,R2 $-3
021E 0D4890   0890  LINE07  LODA,R0 MLINE,R1,-     GET CHAR
0221 CD089B   089B          STRA,R1 MSCR        UPDATE CHAR. INDEX
0224 0600                   LODI,R2 H'00'
0226 F401                   TMI,R0 H'01'        TEST CHARACTER CODE
0228 980A     0234          BCFR,A1 LINE6       IF EVEN
022A 8602                   ADDI,R2 H'02'       CODE IS ODD: LOWER IMAGE
022C F501                   TMI,R1 H'01'        POS. IN LINE ODD?
022E 1802     0232          BCTR,A1 $+4         IF NOT ROTATE
0230 8601                   ADDI,R2 H'01'       ROTATE
0232 1B06     023A          BCTR,UN LINE7
0234 F501           LINE6   TMI,R1 H'01'        POS. IN LINE EVEN?
0236 9802     023A          BCFR,A1 $+4         POS. EVEN: UPPER BITS - NO ROTATE
0238 8601                   ADDI,R2 H'01'
023A 3B0F     024B  LINE7   BSTR,UN CHARPS
023C 0D427B   027B  LINE8   LODA,R0 IMGTBL,R1,-
023F E602                   COMI,R2 H'02'       UPPER OR LOWER IMAGE?
0241 1A12     0255          BCTR,LT LINE05      IF UPPER
0243 440E                   ANDI,R0 H'0E'       LOWER PART OF IMAGE
0245 E602                   COMI,R2 H'02'
0247 1816     025F          BCTR,EQ LINE06      IF 02: NO ROTATE
0249 1B10     025B          BCTR,UN LINE09

024B 0706           CHARPS  LODI,R3 6           INDEX
024D 44FE                   ANDI,R0 H'FE'
024F C1                     STRZ R1             FORM INDEX: (R0) H'FE'+6
0250 D0                     RRL,R0
0251 81                     ADDZ R1
0252 83                     ADDZ R3
0253 C1                     STRZ R1
0254 17                     RETC,UN

0255 44E0           LINE05  ANDI,R0 H'E0'       UPPER PART OF IMAGE
0257 E600                   COMI,R2 00
0259 1804     025F          BCTR,EQ LINE06      NO ROTATE
```

```
LOC  OBJECT   ADDR  SOURCE  LINE

                            *CHARACTER AND LINE ROUTINES, CONTINUED

025B D0                     RRL,R0
025C D0                     RRL,R0
025D D0                     RRL,R0
025E D0                     RRL,R0
025F 6F4878   0878  LINE06  IORA,R0 MONOB+24  *(NLINES-1),R3,-
0262 CF6878   0878          STRA,R0 MONOB+24  *(NLINES-1),R3
0265 5B55     023C          BRNR,R3 LINE8     END OF MERGE?
0267 0C089B   089B          LODA,R0 MSCR      INDEX OF CHAR IN MLINE
026A 14                     RETC,Z            IF ZERO LAST CHAR ALREADY IN POSITION
026B 3B5E     024B          BSTR,UN CHARPS    FORM INDEX FOR COPY
026D 0F4878   0878          LODA,R0 MONOB+24  *(NLINES-1),R3,-
0270 CD4878   0878          STRA,R0 MONOB+24  *(NLINES-1),R1,-
0273 5B78     026D          BRNR,R3 $-6
0275 0D089B   089B          LODA,R1 MSCR      INDEX OF NEXT CHAR
0278 1F0212   0212          BCTA,UN LINE2

                            *IMAGE TABEL FOR CHARACTER DISPLAY

027B E2A6A2A2       IMGTBL  DATA H'E2,A6,A2,A2,A2,E2'   0,1   00,01
0281 EE22EE82               DATA H'EE,22,EE,82,82,EE'   2,3   02,03
0287 AEA8AEE2               DATA H'AE,A8,AE,E2,22,2E'   4,5   04,05
028D EE8282E2               DATA H'EE,82,82,E2,A2,E2'   6,7   06,07
0293 EEAAEEA2               DATA H'EE,AA,EE,A2,A2,EE'   8,9   08,09
0299 E8A8EEAA               DATA H'E8,A8,EE,AA,AA,AE'   A,B   0A,0B
029F E2828E8A               DATA H'E2,82,8E,8A,8A,EE'   C,D   0C,0D
02A5 EE88EE88               DATA H'EE,88,EE,88,88,E8'   E,F   0E,0F

02AB E8888888               DATA H'E8,88,88,88,A8,EE'   G,L   10,11
02B1 E0404E4A               DATA H'E0,40,4E,4A,4A,EA'   I,N   12,13
02B7 E0A0EE88               DATA H'E0,A0,EE,88,88,88'   P,R   14,15
02BD 000000E0               DATA H'00,00,00,E0,00,E0'   =,    16,17 SPACE
02C3 4040EE40               DATA H'40,40,EE,40,40,00'   +,-   18,19
02C9 004A044A               DATA H'00,4A,04,4A,00,00'   :,X   1A,1B
```

```
LOC  OBJECT    ADDR  SOURCE   LINE

                             *SCREEN ROUTINES

02CF 0588            SCROLL   LODI,R1 256-24 *(NLINES-1) BYTES TO BE KEPT
02D1 0D6790  0790    SCROL1   LODA,R0 MONOB-256+24 *NLINES,R1
02D4 CD6778  0778             STRA,R0 MONOB-256+24 *(NLINES-1),R1
02D7 D978    02D1             BIRR,R1 SCROL1
02D9 0608            SCROL3   LODI,R2 8          SPACES TO MLINE
02DB 0417            SCROL4   LODI,R0 H'17'
02DD CE4890  0890             STRA,R0 MLINE,R2,-
02E0 5A7B    02DD             BRNR,R2 $-3
02E2 17                       RETC,UN

                             *OUTPUT OF MESSAGE TO A NEW LINE

02E3 CE089B  089B    NEWMES   STRA,R2 MSCR       TEMP STORE
02E6 3B67    02CF             BSTR,UN SCROLL
02E8 0AFA    02E4             LODR,R2 NEWMES+1 MSCR

                             *MESSAGE OF 4 CHARACTERS TO BEGIN OF LINE ROUTINE

                             *USE OF REGISTERS:  R0 XX
                                                 R3 INDEX IN MLINE
                                                 R2 INDEX IN MESSAGE

02EA 0704            MESSAG   LODI,R3 4
02EC 0E42F5  02F5             LODA,R0 MESSL,R2,-
02EF CF4890  0890             STRA,R0 MLINE,R3,-
02F2 5B78    02EC             BRNR,R3 $-6
02F4 17                       RETC,UN                          R2:
02F5 140C1716        MESSL    DATA H'14,0C,17,16'    "PC  ="   04
02F9 0A0D1716                 DATA H'0A,0D,17,16'    "AD  ="   08
02FD 15171716                 DATA H'15,17,17,16'    "R   ="   0C
0301 0B140116                 DATA H'0B,14,01,16'    "BP1 ="   10
0305 0B140216                 DATA H'0B,14,02,16'    "BP2 ="   14
0309 0B0E1016                 DATA H'0B,0E,10,16'    "BEG="    18
030D 0E130D16                 DATA H'0E,13,0D,16'    "END="    1C
0311 050A0D16                 DATA H'05,0A,0D,16'    "SAD="    20
0315 0F121116                 DATA H'0F,12,11,16'    "FIL ="   24
0319 12121212                 DATA H'12,12,12,12'    "IIII"    28
```

```
LOC  OBJECT   ADDR  SOURCE   LINE

                            *ROUTINES FOR INPUT AND DISPLAY OF CONSTANTS

                            *ASSEMBLE VALUE
                            *(FSCRM+1) IS SHIFTED INTO (FSCRM+2,FSCRM+3)
                            *AT EXIT (R2,R1) ALSO IS THE NEW VALUE
031D 0C08A7   08A7  ASMDAT  LODA,R0 FSCRM+7 1ST DIGIT?
0320 381E     0340          BSTR,Z RESDAT
0322 0401                   LODI,R0 1          SET FSCRM+7 TO INPUT FOLLOWING DIGITS
0324 C8F8     031E          STRR,R0 ASMDAT+1
0326 3B1E     0346          BSTR,UN LOADAT     LOAD DATA IN R0,R1,R2
                            *THIS ROUTINE ADDS ONE HEX CONSTANT IN FSCRM+1 TO
                            *A DOUBLE BYTE HEX VALUE IN R2 AND R1
                            *USE OF REGISTERS:  R0 NEW HEX
                                                R1 LOWER HEX
                                                R2 UPPER HEX
                                                R3 INDEX
0328 770A           ASSMC   PPSL WC+COM
032A 7501                   CPSL C
032C D0                     RRL,R0
032D D0                     RRL,R0
032E D0                     RRL,R0
032F D0                     RRL,R0
0330 0704                   LODI,R3 4          4 SHIFT
0332 D0             ASSMC1  RRL,R0
0333 D1                     RRL,R1
0334 D2                     RRL,R2
0335 FB7B     0332          BDRR,R3 ASSMC1     ASSEMBLE LOOP
0337 7508                   CPSL WC

0339 CD08A3   08A3  STRDAT  STRA,R1 FSCRM+3    LOWER BYTE
033C CE08A2   08A2          STRA,R2 FSCRM+2    UPPER BYTE
033F 17                     RETC,UN

                            *RESET DATA ASSEMBLY LOCATIONS

0340 20             RESDAT  EORZ R0
0341 C8F7     033A          STRR,R0 STRDAT+1 FSCRM+3
0343 C8F8     033D          STRR,R0 STRDAT+4 FSCRM+2
0345 17                     RETC,UN

0346 0C08A1   08A1  LOADAT  LODA,R0 FSCRM+1    NEW DATA
0349 09EF     033A          LODR,R1 STRDAT+1 FSCRM+3  LOWER BYTE OLD DATA
034B 0AF0     033D          LODR,R2 STRDAT+4 FSCRM+2  UPPER BYTE OLD DATA
034D 17                     RETC,UN


                            *OUTPUT OF A DOUBLE BYTE TO MLINE
                            *USE OF REGISTERS:  R0 XX
                                                R1 LOWER BYTE
                                                R2 UPPER BYTE
                                                R3 POSITION OF DATA IN MLINE
034E 0708           OUTADD  LODI,R3 8
0350 3B02     0354  OUTADR  BSTR,UN OUTBYT     OUTPUT LOWER BYTE
0352 02                     LODZ R2            UPPER BYTE
0353 C1                     STRZ R1
```

```
LOC  OBJECT   ADDR  SOURCE   LINE

                            *OUTPUT OF A SINGLE BYTE TO MLINE
                            *USE OF REGISTERS:  R0 SCR BYTE
                                                R1 DATA BYTE
                                                R3 MLINE INDEX

0354 3B08     035E  OUTBYT   BSTR,UN NIBL       SEPARATE DIGITS
0356 CF488F   088F          STRA,R0 MLINE-1,R3,-    UPPER
0359 01                     LODZ R1
035A CF4891   0891          STRA,R0 MLINE+1,R3,-    LOWER
035D 17                     RETC,UN
035E 01             NIBL    LODZ R1
035F 50                     RRR,R0
0360 50                     RRR,R0
0361 50                     RRR,R0
0362 50                     RRR,R0
0363 440F                   ANDI,R0 H'0F'       UPPER CHAR.
0365 450F                   ANDI,R1 H'0F'       LOWER CHAR.
0367 17                     RETC,UN

                            *OUTPUT OF ADDRESS BEING ENTERED

0368 0708           OUTNAD  LODI,R3 8           POSITION IN MLINE
036A 3B5A     0346          BSTR,UN LOADAT      NEW ADDRESS

                            *OUTPUT OF ADDRESS (R1,R2) WITH LEADING SPACES (FSCRM+6)

036C 3B62     0350  OUTADS  BSTR,UN OUTADR
036E 0C08A6   08A6          LODA,R0 FSCRM+6
0371 14                     RETC,Z
0372 C1                     STRZ R1
0373 F800     0375          BDRR,R0 $+2
0375 C8F8     036F          STRR,R0 $-6 FSCRM+6

                            *OUTPUT OF LEADING SPACES

0377 03             SPACES  LODZ R3             POSITION OF SPACES IN MLINE
0378 C2                     STRZ R2
0379 0417                   LODI,R0 H'17'       SPACE
037B CE288F   088F          STRA,R0 MLINE-1,R2,+
037E F97B     037B          BDRR,R1 $-3         (R1): NO OF SPACES
0380 17                     RETC,UN

                            *STORE DATA IN MEMORY

0381 0C08A3   08A3  STDATA  LODA,R0 FSCRM+3
0384 0D08A4   08A4          LODA,R1 FSCRM+4
0387 0E08A5   08A5          LODA,R2 FSCRM+5
038A E508                   COMI,R1 H'08'       NO PATCH IN MON SCRATCH MEMORY
038C 9804     0392          BCFR,EQ STDAT1
038E E6C0                   COMI,R2 > (PC+2)
0390 1A09     039B          BCTR,LT STDAT2      IF IN SCRATCH SKIP STORE
0392 CC88A4   08A4  STDAT1  STRA,R0 FSCRM+4
0395 E518                   COMI,R1 H'18'       IN MEM RANGE?
0397 1A02     039B          BCTR,LT STDAT2      IF SO CHECK FOR MEM ERROR
0399 E0                     COMZ R0             TO SET CC EQ
039A 17                     RETC,UN
039B EC88A4   08A4  STDAT2  COMA,R0 FSCRM+4     TEST IF STORE WAS SUCCESSFULL
039E 17                     RETC,UN

                            *INCREMENT ADDRESS (FSCRM+4,FSCRM+5)

039F 770B           INCADR  PPSL,C+WC+COM
03A1 0502                   LODI,R1 2           DOUBLE BYTE
03A3 0D48A4   08A4          LODA,R0 FSCRM+4,R1,-
03A6 8400                   ADDI,R0 0           PLUS CARRY
03A8 CD68A4   08A4          STRA,R0 FSCRM+4,R1
03AB 5976     03A3          BRNR,R1 $-8
03AD 7509                   CPSL C+WC
03AF 17                     RETC,UN
```

```
LOC  OBJECT   ADDR  SOURCE  LINE
                            *REGISTER DISPLAY AND ALTER
03B0 1A2E     03E0  REG01   BCTR,N REG3        IF ENTER
03B2 01                     LODZ,R1
03B3 1A03     03B8          BCTR,N REG1        IF RE-ENTER
03B5 02                     LODZ,R2
03B6 981E     03D6          BCFR,Z REG2        IF DATA INPUT

03B8 060C           REG1    LODI, R2 12        "R = "
03BA 3F02E3   02E3          BSTA,UN NEWMES
03BD 0401                   LODI,R0 1          ENABLE INPUT OF DATA
03BF CC08A0   08A0          STRA,R0 FSCRM
03C2 0A9E     03E2          LODR,R2 * REG3+2   FSCRM+4 REG. NO
03C4 CE0891   0891          STRA,R2 MLINE+1    TO DISPLAY LINE
03C7 0E68AC   08AC          LODA,R0 REGM, R2   OLD DATA
03CA C1                     STRZ,R1            FOR DATA DISPLAY
03CB C898     03E5          STRR,R0 * REG3+5   FSCRM+3 FOR RESTORE
03CD 20                     EORZ,R0
03CE CC08A7   08A7          STRA,R0 FSCRM+7    TO HAVE LEADING ZEROS
03D1 0706           REG7    LODI,R3 6          POS. OF DATA
03D3 1F0480   0480          BCTA,UN MEMBYT

03D6 3F031D   031D  REG2    BSTA,UN ASMDAT
03D9 0417                   LODI,R0 H'17'      SPACE
03DB CC0893   0893  IAREG   STRA,R0 MLINE+3   TO INDICATE NOT ENTERED
03DE 1B71     03D1          BCTR,UN REG7

03E0 C3             REG3    STRZ,R3            ENTER CODE
03E1 0E08A4   08A4          LODA,R2 FSCRM+4    REG NO
03E4 0C08A3   08A3          LODA,R0 FSCRM+3    DATA
03E7 CE28AB   08AB          STRA,R0 REGM-1, R2+STORE REG DATA
03EA F720                   TMI,R3 H'20'       TEST FOR + OR - ENTER
03EC 9812     0400          BCFR,A1 REG5       IF - WAS ENTERED
03EE E609                   COMI,R2 9          IF + WAS ENTERED
03F0 1A02     03F4          BCTR,LT $+4        TEST FOR WRAP AROUND
03F2 0600                   LODI,R2 0          WRAP AROUND
03F4 CAEC     03E2  REG4    STRR,R2 * REG3+2   FSCRM+4 REG NO
03F6 0416                   LODI,R0 H'16'      "="
03F8 C8E2     03DC          STRR,R0 *IAREG+1   MLINE+3
03FA 3F020E                 BSTA,UN LINE
03FD 1F03B8   03B8          BCTA,UN REG1

0400 770B           REG5    PPSL C+WC+COM      NO BORROW
0402 A602                   SUBI,R2 2          DECR REG NO
0404 9A02     0408          BCFR,N $+4         IF NOT UNDERFLOW
0406 0608                   LODI,R2 8          WRAP AROUND
0408 7509                   CPSL C+WC
040A 1B68     03F4          BCTR,UN REG4
```

```
LOC  OBJECT   ADDR  SOURCE  LINE
                            *MEMORY DISPLAY AND ALTER
040C 1A0E     041C  MEM0    BCTR,N MEM2       IF ENTER-NEXT
040E 01                     LODZ,R1           COMMAND RE-ENTRY?
040F 1A04     0415          BCTR,N MEM1
0411 02                     LODZ,R2           FSCRM: 1ST ENTRY?
0412 9C0465   0465          BCFR,Z MEM8       IF NOT
0415 0608             MEM1  LODI,R2 8         MESSAGE "AD="
0417 0401                   LODI,R0 1         ENABLE INPUT OF ADDRESS
0419 1F05F7   05F7          BCTA,UN WCAS2
                            *ENTER - NEXT
041C 0C08A0   08A0    MEM2  LODA,R0 FSCRM     ADDRESS OR DATA ENTRY?
041F 991D     043E          BCFR,P MEM4       IF DATA
0421 0502                   LODI,R2 2         COPY MEM ADDRESS
0423 0D48A2   08A2          LODA,R0 FSCRM+2,R1 -
0426 CD68A4   08A4          STRA,R0 FSCRM+4,R1
0429 5978     0423          BRNR,R1 $-6
042B 3F02D9   02D9          BSTA,UN SCROL3    SPACES TO MLINE
042E 04FD             MEM3  LODI,R0 -3        TO ENABLE DATA ENTRY
0430 C8B9     0468          STRR,R0 * MEM6+1  FSCRM
0432 0C88A4   08A4          LODA,R0 * FSCRM+4
0435 CD08A3   08A3          STRA,R0 FSCRM+3   OLD DATA
0438 20                     EORZ,R0           CLR FSCRM+7 TO RESET INPUT LOC.
0439 CC08A7   08A7          STRA,R0 FSCRM+7
043C 1B2C     046A          BCTR,UN MEM6

043E 3F0381   0381    MEM4  BSTA,UN STDATA    ACCEPT DATA
0441 980A     044D          BCFR,EQ MEM7      IF STORE FAILS
0443 0C0899   0899          LODA,R0 ENTM      ENTER KEY CODE
0446 E4C0                   COMI,R0 H'C0'     - ENTER?
0448 1808     0452          BCTR,EQ MEM9      DECR. MEM ADDRESS
044A 3F039F   039F          BSTA,UN INCADR    INCR. MEM ADDRESS
044D 3F02CF   02CF    MEM7  BSTA,UN SCROLL
0450 1B5C     042E          BCTR,UN MEM3

0452 040A             MEM9  LODI,R0 WC+COM    WC, SET BORROW
0454 93                     LPSL
0455 0502                   LODI,R1 2
0457 0D48A4   08A4          LODA,R0 FSCRM+4,  R1 -
045A A400                   SUBI,R0 0
045C CD68A4   08A4          STRA,R0 FSCRM+4,  R1
045F 5976     0457          BRNR,R1 $-8
0461 7509                   CPSL  C+WC
0463 1B68     044D          BCTR,UN MEM7

0465 1A1F     0486    MEM8  BCTR,N MEMFS7     IF DATA
0467 3F031D   031D    MEM5  BSTA,UN ASMDAT    ASSEMBLE DATA
046A 0C08A0   08A0    MEM6  LODA,R0 FSCRM     ADDRESS OR DATA?
046D 1D0532   0532          BCTA,P G02        IF ADDRESS
0470 0704                   LODI,R3 4         POSITION OF ADDRESS
0472 0D08A5   08A5          LODA,R1 FSCRM+5
0475 0E08A4   08A4          LODA,R2 FSCRM+4
0478 3F0350   0350          BSTA,UN OUTADR    OUTPUT OF ADDRESS
047B 0708                   LODI,R3 8         POSITION OF MEM BYTE
047D 0D08A3   08A3    MEMFS3 LODA,R1 FSCRM+3  FOR OUTPUT
0480 3F0354   0354    MEMBYT BSTA,UN OUTBYT
0483 1F020E   020E    MEMLIN BCTA,UN LINE

0486 0C08A7   08A7    MEMFS7 LODA,R0 FSCRM+7  1ST OR 2ND DIGIT?
0489 9817     04A2          BCFR,Z FMEM2      IF 2ND
048B E6FD                   COMI,R2-3         1ST DATA INPUT?
048D 1806     0495          BCTR,EQ FMEM1
048F 3F02CF   02CF          BSTA,UN SCROLL    NOT IF FIRST
0492 3F039F   039F          BSTA,UN INCADR    NEW DIGIT FOR NEXT BYTE
0495 04FF             FMEM1 LODI,R0-1         ENABLE SCROLL
0497 C8D2     0468          STRR,R0 MEM6+1 FSCRM      IN NEXT BYTE
0499 3B4C     0467    FMEM3 BSTR,UN MEM5
049B 3F0381   0381          BSTA,UN STDATA
049E 9C0415   0415          BCFA,EQ MEM1      IF STORE FAILS
04A1 17                     RETC,UN

04A2 3B75     0499    FMEM2 BSTR,UN FMEM3
04A4 20                     EORZ R0           TO ENABLE INPUT OF 1ST DIGIT
04A5 C8E0     0487          STRR,R0 MEMFS7+1  FSCRM+7
04A7 1BDB     0484          BCTR,UN MEMLIN+1  LINE
```

```
LOC  OBJECT    ADDR  SOURCE  LINE

                           *BREAKPOINT SET AND CLEAR

04A9 0D0898    0898  BKSC    LODA,R1 FSEQ      TEST FOR BK1 OR BK2
04AC F520                    TMI,R1 H'20'
04AE 9C04F5    04F5          BCFA,A1 FBK20
04B1 60                      IORZ R0           TEST FOR ENTER
04B2 1A27      04DB          BCTR,N FBK4       IF ENTER/CLR
04B4 02                      LODZ R2 FSCRM
04B5 191A      04D1          BCTR,P FBK3       IF NOT 1ST
04B7 0602                    LODI,R2 2         1ST ENTRY
04B9 0E48B5    08B5          LODA,R0 BK1,R2,-  COPY CURRENT BK1 VALUE
04BC CE68A2    08A2          STRA,R0 FSCRM+2,R2
04BF 5A78      04B9          BRNR,R2 $-6
04C1 0610                    LODI,R2 16        "BP1="
04C3 3F02E3    02E3  FBK1    BSTA,UN NEWMES
04C6 0401                    LODI,R0 1         ENABLE INPUT OF NEW BK ADDRESS
04C8 CC08A0    08A0          STRA,R0 FSCRM
04CB 3F0346    0346  FBK2    BSTA,UN LOADAT    LOAD BK VALUE FOR DISPLAY
04CE 1F0529    0529          BCTA,UN GO3

04D1 3F031D    031D  FBK3    BSTA,UN ASMDAT
04D4 0417                    LODI,R0 H'17'     SPACE
04D6 CC0893    0893  IABK    STRA,R0 MLINE+3   TO INDICATE NOT ENTERED
04D9 1B70      04CB          BCTR,UN FBK2

04DB 0602            FBK4    LODI,R2 2         ENTER BK1
04DD 0516            FBK5    LODI,R1 H'16'     "="
04DF C9F6      04D7          STRR,R1 IABK+1 MLINE+2    TO INDICATE ENTERED
04E1 F420                    TMI,R0 H'20'      ENTER -?
04E3 BC0340    0340          BSFA,A1 RESDAT    THEN CLEAR BK
04E6 0502                    LODI,R1 2
04E8 0D48A2    08A2          LODA,R0 FSCRM+2,R1,-      NEW BK VALUE
04EB CE48B5    08B5          STRA,R0 BK1,R2,-          TO BK1/2
04EE 5978      04E8          BRNR,R1 $-6
04F0 CD089A    089A          STRA,R1 MFUNC     SET NO FUNCTION
04F3 1B56      04CB          BCTR,UN FBK2

04F5 60              FBK20   IORZ R0           TEST FOR ENTER
04F6 1A12      050A          BCTR,N FBK21      IF ENTER
04F8 02                      LODZ R2 FSCRM
04F9 9856      04D1          BCFR,Z FBK3       IF NOT 1ST
04FB 0502                    LODI,R1 2         COPY OLD BK2
04FD 0D48B7    08B7          LODA,R0 BK2,R1,-
0500 CD68A2    08A2          STRA,R0 FSCRM+2,R1
0503 5978      04FD          BRNR,R1 $-6
0505 0614                    LODI,R2 20        "BP2="
0507 1F04C3    04C3          BCTA,UN FBK1

050A 0604            FBK21   LODI,R2 4
050C 1B4F      04DD          BCTR,UN FBK5
```

```
LOC  OBJECT    ADDR  SOURCE  LINE

                             *GO TO ADDRESS FUNCTION

050E 1A30      0540  GO0     BCTR,N STRTUP      IF X-KEY ENTERED
0510 02                      LODZ R2            FSCRM: TEST FOR 1ST ENTRY
0511 981C      052F          BCFR,Z GO1         IF NOT 1ST
0513 0604                    LODI,R2 4          MESSAGE "PC="
0515 3F02E3    02E3  GO4     BSTA,UN NEWMES
0518 0D08BF    08BF          LODA,R1 PC+1       OLD VALUE OF PC
051B 0E08BE    08BE          LODA,R2 PC         TO SCRATCH
051E 0403                    LODI,R0 3          3 SPACES
0520 CC08A6    08A6          STRA,R0 FSCRM+6
0523 CC08A0    08A0          STRA,R0 FSCRM      ENABLE INPUT OF ADDRESS
0526 3F0339    0339          BSTA,UN STRDAT
0529 3F034E    034E  GO3     BSTA,UN OUTADD     OUTPUT OLD ADDRESS
052C 1F020E    020E  GOLIN    BCTA,UN LINE

052F 3F031D    031D  GO1     BSTA,ASMDAT        ASSEMBLE NEW ADDRESS
0532 3F0368    0368  GO2     BSTA,UN OUTNAD     OUTPUT NEW ADDRESS
0535 1BF6      052D          BCTR,UN GOLIN+1


                             *START UP

0537 9886              ZBRR1 DATA H'9B', > BVEC1+H'80'
0539 9888              ZBRR2 DATA H'9B', > BVEC2+H'80'

053B 09037700          UMODE DATA H'09,03,77,00,1F'      INDIRECT ADDRESS, PPSL, BCTA

0540 0502              STRTUP LODI,R1 2           ALL DOUBLE BYTE ACTIONS
0542 0D48A2    08A2          LODA,R0 FSCRM+2,R1,-      COPY NEW PC VALUE
0545 CD68BE    08BE          STRA,R0 PC,R1
0548 0DE8B5    08B5          LODA,R0 BK1,R1     SAVE DATA OF BK1
054B CD68A2    08A2          STRA,R0 FSCRM+2,R1
054E 0DE8B7    08B7          LODA,R0 BK2,R1     SAVE DATA OF BK2
0551 CD68A4    08A4          STRA,R0 FSCRM+4,R1
0554 0D6537    0537          LODA,R0 ZBRR1,R1   SET BREAKPOINT 1
0557 CDE8B5    08B5          STRA,R0 BK1,R1
055A 0D6539    0539          LODA,R0 ZBRR2,R1   SET BREAKPOINT 2
055D CDE8B7    08B7          STRA,R0 BK2,R1
0560 5960      0542          BRNR,R1 STRTUP+2

0562 0505                    LODI,R1 5          PREPARE RAM FOR START UP PROGRAM
0564 0D453B    053B          LODA,R0 UMODE,R1,-
0567 CD68B9    08B9          STRA,R0 PC-5,R1
056A 5978      0564          BRNR,R1 $-6

056C 0C08B3    08B3          LODA,R0 REGM+7     PSL STATUS
056F CC08BC    08BC          STRA,R0 PC-2

0572 7510                    CPSL RS            BANK 0
0574 0D08AD    08AD          LODA,R1 REGM+1
0577 0E08AE    08AE          LODA,R2 REGM+2
057A 0F08AF    08AF          LODA,R3 REGM+3
057D 7710                    PPSL RS            BANK 1
057F 0D08B0    08B0          LODA,R1 REGM+4
0582 0E08B1    08B1          LODA,R2 REGM+5
0585 0F08B2    08B2          LODA,R3 REGM+6
0588 0C08B4    08B4          LODA,R0 REGM+8     PSU STATUS
058B 92                      LPSU
058C 0C08AC    08AC          LODA,R0 REGM
058F 75FF                    CPSL H'FF'         CLEAR PSL STATUS
0591 1F08BB    08BB          BCTA,UN PC-3       START
```

```
LOC  OBJECT   ADDR  SOURCE   LINE

                             *BREAKPOINT ENTRIES FROM USER PROGRAM

0594 3B37     05CD  BREAK1   BSTR,UN BRKSBR    BREAKPOINT SUBROUTINE
0596 0D08B6   08B6           LODA,R1 BK1+1
0599 0E08B5   08B5           LODA,R2 BK1
059C C8F9     0597           STRR,R0 $-5       CLEAR BK1
059E C8FA     059A           STRR,R0 $-4
05A0 3B22     05C4           BSTR,UN PCEQBK    BRK ADDRESS IS NEW PC
05A2 0610                    LODI,R2 16        "BP1="
05A4 3F02EA   02EA  BREAKM   BSTA,UN MESSAG
05A7 0415                    LODI,R0 H'15'     CHANGE "BP" INTO "BR"
05A9 CC0891   0891           STRA,R0 MLINE+1
05AC 3F020E   020E           BSTA,UN LINE
05AF 1F0038   0038           BCTA,UN START1    TO DISPLAY & WAIT LOOP

05B2 3B19     05CD  BREAK2   BSTR,UN BRKSBR    BREAK SUBROUTINE
05B4 0D08B8   08B8           LODA,R1 BK2+1
05B7 0E08B7   08B7           LODA,R2 BK2
05BA C8F9     05B5           STRR,R0 $-5       CLEAR BK2
05BC C8FA     05B8           STRR,R0 $-4
05BE 3B04     05C4           BSTR,UN PCEQBK    BRK ADDRESS IS NEW PC
05C0 0614                    LODI,R2 20        "BP2="
05C2 1B60     05A4           BCTR,UN BREAKM

05C4 CD08BF   08BF  PCEQBK   STRA,R1 PC+1      STORE NEW PC ADDRESS
05C7 CE08BE   08BE           STRA,R2 PC
05CA 1F034E   034E           BCTA,UN OUTADD    DISPLAY BRKPNT ADDRESS

05CD 3F013E   013E  BRKSBR   BSTA,UN SAVE      SAVE 2650 & PVI STATUS
05D0 0502                    LODI,R1 2         RESTORE BREAKPOINT DATA
05D2 0D48A2   08A2           LODA,R0 FSCRM+2,R1,-
05D5 CDE8B5   08B5           STRA,R0 BK1,R1
05D8 0D68A4   08A4           LODA,R0 FSCRM+4,R1
05DB CDE8B7   08B7           STRA,R0 BK2,R1
05DE 5972     05D2           BRNR,R1 $-12
05E0 3F02CF   02CF  BRKSB1   BSTA,UN SCROLL
05E3 20                      EORZ R0           FOR CLB
05E4 CC089A   089A           STRA,R0 MFUNC
05E7 17                      RETC,UN
```

```
LOC  OBJECT   ADDR  SOURCE  LINE

                            *CASSETTE WRITE

                            *BEGIN ADDRESS:   FSCRM+4,5
                            *END ADDRESS:     FSCRM+7,8
                            *START ADDRESS:   FSCRM+2,3
                            *FILE NO:         FSCRM+1

05E8 1A26     0610  WCAS0   BCTR,N WCAS6      IF ENTER
05EA 01                     LODZ R1           RE-ENTRY?
05EB 1A06     05F3          BCTR,N WCAS1
05ED 02                     LODZ R2 FSCRM
05EE 1D0052F  052F          BCTA,P GO1        ADDRESS INPUT
05F1 1A15     0608          BCTR,N WCAS4      IF FILE NR INPUT
05F3 0401             WCAS1 LODI,R0 1         ENABLE INPUT OF BEG ADR
05F5 0618                   LODI,R2 24        "BEG="
05F7 CC08A0   08A0  WCAS2   STRA,R0 FSCRM     INDICATOR OF FUNCTION STAT
05FA 0403                   LODI,R0 3
05FC CC08A6   08A6          STRA,R0 FSCRM+6   NR OF SPACES
05FF 3F0340   0340          BSTA,UN RESDAT    RESET INPUT LOC
0602 3F02E3   02E3  WCAS3   BSTA,UN NEWMES    OUTPUT MESSAGE
0605 1F020E   020E  WCALIN  BCTA,UN LINE

0608 0C08A1   08A1  WCAS4   LODA,R0 FSCRM+1   FILE NR
060B CC0895   0895          STRA,R0 MLINE+5   FOR DISPLAY
060E 1BF6     0606          BCTR,UN WCALIN+1

0610 02               WCAS6 LODZ R2           FSCRM
0611 1E064A   064A          BCTA,N WCAS9      IF FILE NR ENTERED
0614 0602                   LODI,R2 2         FOR COPY
0616 E402                   COMI,R0 2         TEST F STATUS
0618 9A0E     0628          BCFR,LT WCAS7     IF NOT BEG
061A 0E48A2   08A2          LODA,R0 FSCRM+2,R2,-      BEGIN ADDRESS
061D CE68A4   08A4          STRA,R0 BEGA,R2
0620 5A78     061A          BRNR,R2 $-6
0622 0402                   LODI,R0 2         ENABLE INPUT OF END ADDRESS
0624 061C                   LODI,R2 28        "END="
0626 1B4F     05F7  WCAS2A  BCTR,UN WCAS2

0628 9812     063C  WCAS7   BCFR,EQ WCAS8     IF START ADDRESS
062A 7709                   PPSL C+WC         ADD 1 TO WRITE LAST ADDRESS
062C 0E48A2   08A2          LODA,R0 FSCRM+2,R2,-      END ADDRESS
062F 8400                   ADDI,R0 0
0631 CE68A8   08A8          STRA,R0 ENDA,R2
0634 5A76     062C          BRNR,R2 $-8
0636 0403                   LODI,R0 3         ENABLE INPUT OF START ADDRESS
0638 0620                   LODI,R2 32        "SAD="
063A 1B6A     0626          BCTR,UN WCAS2A
063C 04FF             WCAS8 LODI,R0 -1        ENABLE INPUT OF FILE NR
063E CC08A0   08A0          STRA,R0 FSCRM
0641 20                     EORZ R0           SET DEFAULT FILE IS 0
0642 CC08A1   08A1  WCASF1  STRA,R0 FSCRM+1
0645 0624                   LODI,R2 36        "FIL="
0647 1F0602   0602          BCTA,UN WCAS3
```

```
LOC  OBJECT    ADDR  SOURCE  LINE

                           *CASSETTE WRITE CONTINUED
064A 20                WCAS9   EORZ R0            PULSES FOR AVR
064B 071E                      LODI,R3 30          30 *256 = 7680 PULSES : 1,5 SEC
064D 3F06F7  06F7              BSTA,UN PULSE
0650 F879    064B              BDRR,R0 $-5
0652 3F06F2  06F2              BSTA,UN DELAY       END OF PULSE TRAIN DELAY
0655 0709              BLOCK   LODI,R3 9           SYNC PULSES
0657 3F06F0  06F0              BSTA,UN ZERO+2      PULSE + DELAY
065A 20                        EORZ R0
065B CC08AA  08AA              STRA,R0 BCC         RESET BCC
065E 044C                      LODI,R0 H'4C'       CHECK CHARACTER
0660 3BAF    0691              BSTR,UN IBOUT+1     BOUT AFFECTING BCC
0662 08DF    0643              LODR,R0 WCASF1+1 FSCRM+1   FILE NR
0664 3BAB    0691              BSTR,UN IBOUT+1     BOUT AFFECTING BBC
0666 770B                      PPSL C+WC+COM       NO BORROW
0668 0D08A9  08A9              LODA,R1 ENDA+1
066B 0C08A8  08A8              LODA,R0 ENDA
066E AD08A5  08A5              SUBA,R1 BEGA+1
0671 AC08A4  08A4      WCASB   SUBA,R0 BEGA
0674 1E05F3  05F3              BCTA,N WCAS1        IF NEG. RANGE: RE-ENTER ALL
0677 1908    0681              BCTR,P WCAS10
0679 01                        LODZ,R1             END OF RANGE
067A 1C06BB  06BB              BCTA,Z WCAS13       IF END
067D E410                      COMI,R0 16
067F 1A02    0683              BCTR,LT $+4         IF NOT FULL LINE
0681 0410              WCAS10  LODI,R0 16          16 BYTES PER LINE
0683 CC08AB  08AB              STRA,R0 MCNT
0686 08EA    0672              LODR,R0 WCASB+1 BEGA
0688 3B87    0691              BSTR,UN IBOUT+1     OUTPUT LINE BEG. ADDRESS
068A 08E3    066F              LODR,R0 WCASB-2 BEGA+1
068C 3B83    0691              BSTR,UN IBOUT+1     BOUT
068E 08F4    0684              LODR,R0 WCAS10+3 OUTPUT BLOCK COUNT
0690 3F06D0  06D0      IBOUT   BSTA,UN BOUT
0693 0890              06A5    LODR,R0 WCAS12+1 BCC        OUTPUT OF CHECK CODE
0695 3BFA    0691              BSTR,UN IBOUT+1     CLEARS BCC AUTOMATICALLY
0697 0700                      LODI,R3 0           BYTE INDEX
0699 0FE8A4  08A4      WCAS11  LODA,R0 BEGA,R3
069C EBE6    0684              COMR,R3 WCAS10+3 MCNT END OF LINE?
069E 1804    06A4              BCTR,EQ WCAS12
06A0 3BEF    0691              BSTR,UN IBOUT+1
06A2 DB75    0699              BIRR,R3 WCAS11
06A4 0C08AA  08AA      WCAS12  LODA,R0 BCC         BCC LINE END CODE
06A7 3BE8    0691              BSTR,UN IBOUT+1
06A9 040A                      LODI,R0 H'0A' WC+COM        CLR CARRY
06AB 93                        LPSL
06AC 0502                      LODI,R1 2           DOUBLE BYTE
06AE 03                        LODZ R3             INDEX
06AF 8D48A4  08A4              ADDA,R0 BEGA,R1,-
06B2 CD68A4  08A4              STRA,R0 BEGA,R1
06B5 20                        EORZ R0
06B6 5977    06AF              BRNR,R1 $-7
06B8 1F0655  0655              BCTA,UN BLOCK       NEXT DATA BLOCK
```

```
LOC  OBJECT   ADDR  SOURCE  LINE
                            *CASSETTE WRITE CONTINUED
068B 0C08A2   08A2  WCAS13  LODA,R0 SADR      OUTPUT ENDLINE
068E 3F06D0   06D0          BSTA,UN BOUT
06C1 0C08A3   08A3          LODA,R0 SADR+1
06C4 3BF9     06BF          BSTR,UN WCAS13+4  BOUT
06C6 20                     EORZ R0
06C7 CC089A   089A          STRA,R0 MFUNC     SET NO FUNCTION
06CA 3BF3     06BF          BSTR,UN WCAS13+4  BOUT ZERO BLOCK
06CC 08D7     06A5          LODR,R0 WCAS12+1  BCC
06CE 3BEF     06BF          BSTR,UN WCAS13+4

06D0 3F0726   0726  BOUT    BSTA,UN CBCC      CALCULATE BCC
06D3 7710           COUT    PPSL RS
06D5 7508                   CPSL WC
06D7 0608                   LODI,R2 8         BIT COUNTER
06D9 50                     RRR,R0            TO ENABLE TEST OF BIT 7
06DA D0             COUT 1  RRL,R0
06DB 1A04     06E1          BCTR,N COUT2      IF ONE
06DD 3B0F     06EE          BSTR,UN ZERO
06DF 1B02     06E3          BCTR,UN COUT3
06E1 3B07     06EA  COUT2   BSTR,UN ONE
06E3 FA75     06DA  COUT3   BDRR,R2 COUT1
06E5 3B0B     06F2          BSTR,UN DELAY     EXTRA DELAY AT CHAR. END
06E7 7519                   CPSL C+WC+RS
06E9 17                     RETC,UN
                            *CASSETTE WRITE CONTINUED
06EA 0706           ONE     LODI,R3 6         6 PULSES IS A ONE
06EC 1B02     06F0          BCTR,UN ZERO+2

06EE 0703           ZERO    LODI,R3 3         3 PULSES IS A ZERO
06F0 3B05     06F7          BSTR,UN PULSE

06F2 0532           DELAY   LODI,R1 50        500 µSEC DELAY
06F4 F97E     06F4          BDRR,R1 $
06F6 17                     RETC,UN

06F7 05FF           PULSE   LODI,R1 H'FF'
06F9 CD1DFF   1DFF          STRA,R1 CASW
06FC 0508                   LODI,R1 8         DELAY
06FE F97E     06FE          BDRR,R1 $
0700 C9F8     06FA          STRR,R1 PULSE+3
0702 0507                   LODI,R1 7         DELAY
0704 F97E     0704          BDRR,R1 $
0706 FB6F     06F7          BDRR,R3 PULSE
0708 17                     RETC,UN
```

```
LOC  OBJECT   ADDR  SOURCE  LINE
0709 771A           CHIN    PPSL RS+WC+COM
070B 0608                   LODI,R2 8           PRESET BIT COUNTER
070D CE089B 089B    CHI1    STRA,R2 MSCR
0710 3B1F   0731    CHI2    BSTR,UN RBIT        READ BIT
0712 E686           CHI3    COMI,R2 119+15
0714 1A05   071B            BCTR,LT CHI4        IF VALID BIT
0716 DA78   0710            BIRR,R2 CHI2        MORE THAN 14 EDGES BCC ERROR
0718 1F0788 0788            BCTA,UN RCAS1       NEW LEADER DETECTED
071B 0C089C 089C    CHI4    LODA,R0 MSCR+1
071E D2                     RRL,R2
071F D0                     RRL,R0              MAKE BYTE OF 8 SUCCESSIVE BITS
0720 C8FA   071C            STRR,R0 CHI4+1
0722 0AEA   070E            LODR,R2 CHI1+1      DECREMENT BIT COUNTER
0724 FA67   070D            BDRR,R2 CHI1

0726 7518           CBCC    CPSL WC+RS
0728 C1                     STRZ R1
0729 2C08AA 08AA            EORA,R0 BCC
072C D0                     RRL,R0
072D C8FB   072A            STRR,R0 CBCC+4 BCC
072F 01                     LODZ R1
0730 17                     RETC,UN

0731 098B   073E    RBIT    LODR,R1 RBIT4+1 CASR
0733 20                     EORZ R0
0734 CC0898 0898    RBIT1   STRA,R0 SILENC      SILENCE COUNTER
0737 0677                   LODI,R2 119         TRANSITION COUNTER
0739 0703           RBIT2   LODI,R3 3           SET PAUSE COUNTER
073B C0             RBIT3   NOP
073C 01                     LODZ R1             PREVIOUS SAMPLE
073D 0D19BF 19BF    RBIT4   LODA,R1 CASR        (see note 1)
0740 21                     EORZ R1             LOOK FOR TRANSITION
0741 9A04   0747            BCFR,N RBIT5
0743 DA74   0739            BIRR,R2 RBIT2       INCREMENT TRANSITION COUNTER
0745 FA72   0739            BDRR,R2 RBIT2       TO AVOID OVERFLOW OF R2
0747 E679           RBIT5   COMI,R2 119+2       ENOUGH EDGES?
0749 FB70   073B            BDRR,R3 RBIT3       NO EDGE, PAUSE COUNTDOWN
074B 15                     RETC,GT             IF MORE THAN 2 EDGES
074C 08E7   0735            LODR,R0 RBIT+1 SILENC
074E F864   0734            BDRR,R0 RBIT1       TIME OUT: 256 *(3 *18+8) *3. *3.4 μSEC = 54 MSEC
0750 0C08A8 08A8            LODA,R0 FSCRM+8     VERIFY MODE?
0753 185C   0731            BCTR,Z RBIT         THEN BCC ERROR
0755 1F0788 0788            BCTA,UN RCAS1       IF READ: RESTART READING
```

Note 1. The instruction at 073D should have read '0D1DBF'.

```
LOC  OBJECT   ADDR  SOURCE  LINE

                            *READ CASSETTE

0758 1A1E     0778  RCAS0   BCTR,N RCAS2      IF ENTER
075A 0C08A1   08A1          LODA,R0 FSCRM+1
075D 1905     0764          BCTR,P RCAS7      IF FILE NR ENTERED
075F 0624                   LODI,R2 36        "FIL="
0761 1F0602   0602          BCTA,UN WCAS3
0764 CC0895   0895  RCAS7   STRA,R0 MLINE+5
0767 1F020E   020E          BCTA,UN LINE

076A CE0897   0897  CHLINE  STRA,R2 MLINE+7   NUMBER OF PASSING FILE
076D 3BF9     0768  CHLIN1  BSTR,UN RCAS7+4 LINE
076F 0578                   LODI,R1 24 *(NLINES-1)      LAST LINE TO OBJECTS
0771 1F0071   0071          BCTA,UN FILELN

0774 3B74     076A  RCAS8   BSTR,UN CHLINE    DISPLAY NUMBER OF PASSING FILE
0776 1B10     0788          BCTR,UN RCAS1

0778 4420             RCAS2 ANDI,R0 H'20'     + OR -
077A CC08A8   08A8          STRA,R0 FSCRM+8   TO SET VERIFY FLAG
077D 0518                   LODI,R1 H'18'     "+"
077F 5802     0783          BRNR,R0 $ + 4
0781 0519                   LODI,R1 H'19'     "-"
0783 CD0893   0893          STRA,R1 MLINE+3
0786 3B65     076D          BSTR,UN CHLIN1
```

```
LOC  OBJECT   ADDR  SOURCE  LINE

0788 3F0731   0731  RCAS1   BSTA,UN RBIT      LOOK FOR SYNC
078B E686                   COMI,R2 119+15
078D 1A79     0788          BCTR,LT RCAS1     IF TOO SHORT
078F E68C                   COMI,R2 119+21
0791 1975     0788          BCTR,GT RCAS1     IF TOO LONG
0793 20                     EORZ R0
0794 CC08AA   08AA          STRA,R0 BCC
0797 3F0709   0709  ICHIN1  BSTA,UN CHIN      READ CHECK CHAR
079A E44C                   COMI,R0 H'4C'     TEST CHAR
079C 986A     0788          BCFR,EQ RCAS1     IF NOT EQUAL TO TEST CHAR
079E 0C1F15   1F15  IOBJA   LODA,R0 OBJ2+5    BLOCK SYNC FLASH POINT
07A1 240A                   EORI,R0 H'0A'     FAST FLASH IF NOT CORRECT FILE
07A3 C8FA     079F          STRR,R0 IOBJA+1 OBJ2
07A5 3BF1     0798          BSTR,UN ICHIN1+1 CHIN
07A7 C2                     STRZ R2           FILE NO
07A8 0C08A1   08A1  RCAS3   LODA,R0 FSCRM+1   FILE NO SPECIFIED
07AB 1803     07B0          BCTR,Z RCAS4      ZERO IS ANY FILE
07AD E2                     COMZ R2           FILE FOUND?
07AE 9844     0774          BCFR,EQ RCAS8     IF NOT FOUND
07B0 07FE                   RCAS4   LODI,R3 254       2 BYTES FOR ADDRESS
07B2 CF1F4A   1F4A          STRA,R3 X4        NO DISPLAY OF PREVIOUS FILE NO
07B5 3B9B     07D2          BSTR,UN ICHIN+1 CHIN
07B7 CF67A6   07A6          STRA,R0 FSCRM+4-254,R3
07BA CF67C0   07C0          STRA,R0 PC-254,R3
07BD DB76     07B5          BIRR,R3 $ - 8
07BF 3B91     07D2          BSTR,UN ICHIN+1   CHIN BLOCK LENGTH
07C1 60                     IORZ R0           TEST FOR ZERO BLOCK
07C2 980A     07CE          BCFR,Z RCAS9      CONTINUE IF NOT ZERO BLOCK
07C4 0617                   LODI,R2 H'17'     SPACE TO NUMBER OF PASSING FILE
07C6 3F076A   076A          BSTA,UN CHLINE
07C9 05E1                   LODI,R1 H'E1'     TO ENTER THE GO COMMAND
07CB 1F00BD   00BD          BCTA,UN MKEY      CONTINUE IN COMMAND HANDLER

07CE CC08AB   08AB  RCAS9   STRA,R0 MCNT      BLOCK COUNT
07D1 3F0709   0709  ICHIN   BSTA,UN CHIN BCC
07D4 0899     07EF          LODR,R0 RCAS6+1   BCC MUST BE ZERO NOW
07D6 981C     07F4          BCFR,Z RCASER     IF NOT ZERO BCC ERROR
07D8 C3                     STRZ R3           RESET INDEX
07D9 3BF7     07D2  RCAS5   BSTR,UN ICHIN+1 CHIN
07DB EBF2     07CF          COMR,R3 RCAS9+1 MCNT
07DD 180F     07EE          BCTR,EQ RCAS6     IF END OF BLOCK
07DF 0D08A8   08A8          LODA,R1 FSCRM+8   TEST FOR VERIFY MODE
07E2 1803     07E7          BCTR,Z $ + 5
07E4 CFE8A4   08A4          STRA,R0 FSCRM+4,R3        DATA STORE
07E7 EFE8A4   08A4          COMA,R0 FSCRM+4,R3
07EA 9808     07F4          BCFR,EQ RCASER    IF MEMORY ERROR
07EC DB6B     07D9          BIRR,R3 RCAS5
07EE 0C08AA   08AA  RCAS6   LODA,R0 BCC
07F1 1C0788   0788          BCTA,Z RCAS1      NEXT BLOCK
07F4 3F05E0   05E0  RCASER  BSTA,UN BRKSB1    RESET MFUNC, SCROLL
07F7 0608                   LODI,R2 8         "AD="
07F9 3F0515   0515          BSTA,UN GO4       DISPLAY BLOCK ADDRESS & RET
07FC 1F00D4   00D4          BCTA,UN MKEYR
```

```
LOC  OBJECT   ADDR  SOURCE   LINE
0800                        *ORG H'0800'

0800                MONOB    RES 24 *NLINES   MONITOR OBJECT IMAGES
0890                MLINE    RES 8            TEXT LINE

0898                FSEQ     RES 1            FUNCTION SEQUENCE INDICATOR (BP1/2)
           0898     SILENC   EQU FSEQ         SILENCE COUNTER FOR READ CASSETTE
0899                ENTM     RES 1            +, – ENTER KEY MEMORY
089A                MFUNC    RES 1            MONITOR FUNCTION INDEX

089B                MSCR     RES 2            SCRATCH RAM FOR KEYBOARD ROUTINE
089D                RKBST    RES 1            RIGHT KEYBOARD STATUS
089E                LKBST    RES 1            LEFT KEYBOARD STATUS
089F                MKBST    RES 1            MONITOR KEYBOARD STATUS

08A0                FSCRM    RES 12           FUNCTION SCRATCH MEMORY

                             *USE OF FSCRM DURING CASSETTE OPERATIONS

           08A4     BEGA     EQU FSCRM+4      MEMORY BEGIN ADRESS
           08A8     ENDA     EQU FSCRM+8      MEMORY END ADDRESS
           08A2     SADR     EQU FSCRM+2      START ADDRESS OF WRITE FILE
           08AA     BCC      EQU FSCRM+10     CHECK CHAR FOR READ AND WRITE
           08AB     MCNT     EQU FSCRM+11     BLOCK BYTE COUNT FOR READ AND WRITE

08AC                REGM     RES 9            2650 REGISTER STATUS
08B5                BK1      RES 2            BREAKPOINT ADDRESS 1
08B7                BK2      RES 2            BREAKPOINT ADDRESS 2

08B9                INTADR   RES 5            SPACE FOR STARTUP PROGRAM
08BE                PC       RES 2            2650 PC
```

ISBN 0-905705-08-4