A Level Computing Science Project

Computerised Burglar Alarm

By Richard TEER

Table of Contents

Preamble
Introduction 1
System Overview 1
Specifications 1
Using the Alarm 1

Problem Definition and Solution
Formal definition of scope and purpose 2
Analysis of problem 2

Implementation 3

Detailed Description of Hardware
General 4
CPU and Address Decoding Logic Board 4
Memory and I/O Board 5

Detailed Description of Software
General 6
Keypad subroutine 6
Check password subroutine 7
Password OK subroutine 7
Interrupt handler 7
Turn on alarms subroutine 7
Delay subroutine 8
Timer subroutine 8
Timer on subroutine 8
Timer off subroutine 8

Testing the system
General 9
Testing the Hardware 9
Testing the Software 11

Future Developments 13

Errata 13

Glossary 14

Diagrams

Memory Map 15
Chip Pinouts 16
Address Decoding Circuits 18
Clock Generation Circuit 20
Power-Up Reset Circuit 20
Output Circuit 21
Keypad Circuit 22

Listings

Password Generation Program 23
Hardware Test Program 24
Keypad Test Program 26
Operating System Constant Declarations 28
Operating System 29
Interrupt and Reset Vectors 34

Introduction

Being interested in both the hard- and software aspects of computing, I decided that a natural progression from the Security ROM I had written for my home micro would be a stand alone unit, and to that end I designed a computerised burglar alarm system, the prototype being designed with installation into a car in mind, although, with minor modifications, it can be installed elsewhere, e.g., a house.

System Overview

The alarm is constructed on two veroboards, one containing the CPU and address decoding, the other containing the RAM, ROM and interface components. Veroboards were used in preference to PCBs because a design on a veroboard is easier to modify, and less equipment is needed, e.g., acid, etch resistant transfers, etc. A telephone-style numeric keypad is used to type in the security code, and switches in stratigic locations monitor the doors, boot, and bonnet.

Specifications

Power supply	5V @ 250 mA.
Dimensions	28 x 10 cm (l x w).
Hardware used in design	BBC Micro Model B with one 40 track disk drive, a Micron EPROM programmer, and Acorn's 6502 Second Processor.
Software used in design	Acornsoft's 6502 Development Package.
Documentation written on	Acornsoft's VIEW Wordprocessor, Version 2.1 with modifications. Typeset in October 2026 using `vi`, `groff`, `pic`, and `tbl`.

Using the Alarm

Once the alarm is activated, the occupants have approximately one minute to evacuate the vehicle before opening the doors etc. initiate the anti-theft procedures.

Once armed, opening the bonnet or boot will turn on the alarm signal immediately, whilst opening a door starts a timer. This gives the owner approximately thirty seconds to type in his secret code number. If after thirty seconds, or three tries, the correct code hasn't been typed in, the alarm signal will sound. The alarm signal is flashing headlamps, and periodic sounding of the horn. After sounding the alarm, the alarm automatically re-arms itself.

Problem Definition and Solution

Formal definition of scope and purpose

The purpose of this project is to prevent the theft (in the case of this prototype) of one's car, by the use of the following deterants:

i) In the event of either the boot, bonnet, or alarm case being opened, the alarm is turned on,

ii) Once inside, the occupant has thirty seconds in which to deactivate the alarm by keying in his code number. Three attempts are allowed for this.

iii) The ignition is disabled until the correct code number is keyed in.

This project employs a wide variety of both hard- and software techniques. For example, it involves address decoding, clock generation and I/O on the hardware side, and, on the software side, it involves interrupt servicing, keypad decoding, and extensive I/O control.

Analysis of problem

Analysing the problem brought to light several things:

i) The hardware would require a CPU and support circuitry, address decoding logic (ADL), and some form of I/O interface.

ii) The software would have to be able to control the system, both efficently and quickly.

iii) From (ii) it was obvious that the Operating System (OS) had to be written in assembler, and not in a high level language to be compiled later (although C would probably have been a good alternative, had there been a compiler available for my computer).

Implementation

(NB In the following, terms like “then” and “next” etc. are used. Most of what follows took place concurrently, the adverbs merely being used to aid readability, although generally this account is in chronological order.)

The first thing I did was to draw a block diagram of the system. At this stage, everything was depicted as “black boxes”, so that I could get an idea of what address decoding would be required. I ended up with a CPU connected to some RAM, some ROM, and two VIAs, one for the keypad, and one for general I/O. This meant that the address decoding had to decode four addresses, and these were set at &0000, &8000, &A000, and &F000 for the sake of simplicity, although the ROM started at &F000 out of necessity (the reason why is explained later). At this point the choice memory ICs were decided upon, and they are a 2732 4K EPROM and a 6116 2K CMOS RAM.

The next thing to do was to work out the binary equivelents of the four base address, and then design the ADL, on the basis that each chip was selected when its select line was low. To do this, five MSBs of each address were subjected to various truth tables to determine which gates and in which order were required to produce an output of zero when the binary equivilent of each address was presented to the ADL. Then the gates were looked up in a book to determine which chips were required.

Then I looked up the clock generation and reset circuits; being standard they were on the data sheet I had for the 65XX familly of chips (I had already decided to use a 6502 CPU and 6522 VIAs because of my considerable expertise and experience with them).

Then my attention turned to the configuration of the I/O ports, and I decided that apart from the keypad, four outputs and four inputs would be required: the inputs being those from the doors, boot, bonnet, alarm case; and the outputs controling the horn, lights, ignition, and alarm active LED. For completeness’ sake, the I/O port is configured as follows (MSB to LSB): Alarm active LED, Ignition control, Lights, Horn, Doors, Alarm case, Boot, Bonnet; the MSN being the outputs and the LSN being the inputs.

The next thing to be designed was the keypad. Originally, each button was to have its own input line, but most keypads are made up of a matrix, and so an algorthm to decode the keypad was designed. It was also at about this time that I started to design the layout of the first (i.e., CPU and ADL) board.

Now that the design of the hardware was well under way, I started working on the OS, and a program to print out the data required for the keypad routine, given a code number. This, being very simple and short, was completed in a few minutes.

The algorthm for BUGOS started as list of things that it would have to do, and this was elaborated on when I thought of, or had to add, something new. However, the reset procedure was given priority because this would be the first thing that was executed. Because of my in-experiance of writing OSs from scratch, the designing of BUGOS was not always a strictly top-down affair (BUGOS is what I call the OS).

From here on, about three quarters of development time was spent on the software, the remainder being spent on the design of the second (Memory and I/O) board. Being a much simpler design than the first board, the second board layout was finished very quickly, and after checking the design of both boards against my notes, construction began. During the construction of the boards I “knocked up” a simple hardware testing program.

When construction was complete, the hardware was tested as detailed later, and the software completed and debugged; this is also elaborated on later.

Detailed Description of Hardware

The alarm is constructed on two circuit boards, and for convenience, the two boards will be considered seperately. It was originally intended for the seperate boards to be connected together using ribbon cable, but I felt that this arrangement was too flimsy and unreliable, so the completed boards were joined together by soldering together the vero pins that the wires would have been soldered to, and the tracks on the under side of the boards.

CPU and Address Decoding Logic Board

The component schedual for the first board is shown below.

Semiconductors
IC1 .....6502 CPU
IC2 .....555 Timer
IC3 .....7404 Hex Inverters
IC4, 5 .....74LS04 Hex Inverters
IC6, 7 .....74LS20 Dual 4-input NAND Gates
IC8 .....74LS27 Triple 3-input NOR Gates
IC9 .....74LS32 Quad 2-input OR Gates
X1.....1 MHz Crystal

Resistors
R1, 2, 3, 4.....3.3 kΩ
R5, 6.....1.8 kΩ
R7, 8.....1 MΩ

Capacitors
C1, 2, 3.....100 nF Ceramic

Miscellaneous
IC sockets, Wire links, Veroboard pins, 36 X 50 hole Veroboard.

All the ICs on this board are decoupled by 100 nF ceramic capacitors, except for the timer which is decoupled using a 33 µF electrolytic capacitor. Although not really necessary, it is good design practice to include these decoupling capacitors, to smooth out any ripples that may appear on the supply lines caused by the switching gates, or by any inadequacies in the power supply.

When the system is powered-up, IC2 holds $\overline{\text{RST}}$ low for 100 ms so that the system has time to stabilize before operation. This is essential becuase the system might behave erratically if this was not done. The crystal, Xl, together with IC3, and R3, 5, and 6 generates the system's 1 MHz clock.

Pins four, six and thirty eight of IC1 are connected, via R2, 1, and 4 to the 5V supply to pull them up to logic state one. These are the $\overline{\text{IRQ}}$, $\overline{\text{NMI}}$, and Set Overflow Flag ($\overline{\text{SO}}$) pins respectively. This is done to ensure that a correct WIRE-OR operation is carried out on the $\overline{\text{IRQ}}$ and $\overline{\text{NMI}}$ pins, and that the $\overline{\text{SO}}$ pin doesn't go low (which would set the overflow flag in the Status Register if it did).

The five MSBs of the address bus, i.e., A11-A15, are connected to the decoding logic, whilst A1-A10 are routed to connectors at the edge of the board. A11 is unique in that it is the only line to be connected to both the address decoding logic and the I/O board. The address decoding logic decodes four base addresses; &0000, &8000, &A000, and &F000. These will now be described in acending order of address.

As the RAM's address range starts at &0000, A11-A15 must be low. To check for this, the address lines are fed, two at a time to IC9. Because there are five lines to check, and six inputs are used, one of the inputs is tied to ground. The outputs of the OR gates are connected to IC8, which is in turn connected to IC5. The output of IC5 is connected to the RAM chip's $\overline{\text{CS}}$. The RAM was placed at &0000 for two reasons; first the stack is at page one in a 6502 system, and second to allow BUGOS to use page zero for its workspace, page zero being the most economical (in terms of instruction length and execution time) to use.

The system VIA is located at &8000. To be selected, A15 must be high, and A12-A14 must be low. A12-A14 are fed into IC4, and the inverted signals, along with A15, are fed into IC6, whose output is connected to the VIA's $\overline{CS}$. This, and the Auxiliary VIA were placed at their respective address arbitrarily.

The auxiliary VIA is located at &A000, and is selected similarly to the system VIA except that A13 and A15 must be high, A12 and A14 must be low. This VIA is incorperated into the design to allow for future expansion.

The ROM is at the other end of the memory, starting at &F000, so to be selected, A12-A15 must be high. A12-A15 are fed into the inputs of IC7, whose output is connected to the ROM chip's $\overline{CS}$. The ROM was placed at &F000 out of necessity, as explained later.

The chip select signals are brought, together with the various other signals (i.e., $\overline{RST}$, Φ_2, $\overline{IRQ}$, R/$\overline{W}$, $\overline{R}$/W, D0-D7, and A0-A11), to the right hand edge of the board.

Memory and I/O board

The component schedual for the second board is shown below.

Semiconductors
IC10 .....6116 2K CMOS RAM
IC11.....2732 4K EPROM
IC12 .....6522A VIA
IC13 .....74LS126 Quad buffers
Q1, 2, 3, 4 .....BC108C Transistor
D1, 2, 3, 4 .....Red LED

Resistors
R9, 10, 11, 12 .....4.7 kΩ
R13.....470 Ω
R14, 15 ... 21.....4.7 kΩ

Miscellaneous
IC sockets, Wire links, Veropins, 36 X 50 hole Veroboard.

This board really acts as just a carrier for the memory and I/O components. The memory chips are connected to the Address, Data, and Control buses via wire links, whilst the VIA is connected to the Address, Data, and Control buses, as well as IC13 and the numeric keypad. Again, all the ICs are decoupled using 100 nF ceramic capacitors.

The numeric keypad is connected to Port A of the VIA, all of whom's lines are connected to 5V via the 4.7 kΩ pull-up resistors R14, 15, 16, 17, 18, 19, 20, and 21, whilst the outputs of Port B are bufferd by IC13. Connection to the transistors, Q1, 2, 3, and 4, is via resistors R9, 10, 11, and 12. These have a value of 4.7 kΩ because at 5V the maximum current will be roughly 1 mA. In this situation, the buffer IC isn't probably necessary, but as this was my first attempt at designing a computer system, I felt that it was a wise precaution just in case I got something wrong—a 42p TTL chip is cheaper to replace than a £6.00 VIA! It was felt that the inputs didn't need buffering, as these are just microswitches connected to ground. (Addendum: At a later date, the switches will be presented to the VIA via(!) debounce cicuitry, as it is envisaged that the alarm will be converted to a single board micro. This will necessitate a re-write of the operating system, better quality switches, the inclusion of the second VIA, etc. For further details refer to the section on future developments.)

For the sake of this project, the outputs were connected to LEDs, the LEDs reflecting the status of the output bits of the I/O port. The LEDs are grounded through a common cathode 470 Ω resistor (R13) to allow a current of roughly 10 mA to flow.

Detailed Description of Software

These notes are intended to be used in conjunction with the annotations on the program listing. The 6522 VIA has two I/O ports. In the alarm, one is used for the keypad, the other for general I/O. To differentiate between them, the port used for the keypad will be refered to as "the keypad I/O port" and the other as "the I/O port", although it should be noted that in the program listing they are refered to as "I/O Register A" and "I/O Register B" respectively.

The program itself follows and will be described in three parts. The first describes the workspace allocations, the second the initialisation routine and main program loop, and the third describes the subroutines.

First of all, the variables and constants are defined, the majority of which are the zero-page workspace allocations. The rest are the addresses for various registers in the VIA, and the entered code number buffer. The constants are masks for use in checking the flags that the Operating System refers to.

Next is the initialisation routine, reset. This sets up the CPU and VIA to their default states. In the CPU, the Interrupt Disable flag is set, the Decimal flag is cleared, and the Stack Pointer is reset. Next the Interrupt Enable Register in the VIA is set to disable all interrupts, the Data Direction Registers (DDR) are set up, and the I/O ports are set to zero. Then, timerl is set up to count for 10,000 clock cycles (i.e., 1/100 of a second). Next timerl is programmed to operate continuously, and the CPU's Interrupt Disable flag is cleared. Then a call is made to the subroutine `delay`, with the Accumulator set to sixty. This causes a delay of one minute, after which the owner of the car is deemed to have left it, and the alarm arms itself. The number of attempts at entering the alarm is set to three, and the entered password buffer is cleared (all locations are set to &FF). Next the alarm-set LED is lit, and the main loop is entered.

Data from the I/O port is loaded into the Accumulator, and the output bits (i.e., those connected to the ignition, alarm-set LED, and horns) are masked out. If all the bits are one (ie doors, boot, bonnet and alarm case are closed) then if a door has been opened, the password entering routine is entered. If just a door is open, then if this is the first time, a flag is set, a thirty second timer starts, and the I/O port is polled again. If something else is open, the alarm is turned on.

The subroutines will now be described.

Name : `keypad`
Entry conditions : None
Exit conditions : Buffer contains entered password
Function : Collect and process data from the numeric keypad

The X Register is set to zero, and the DDR in the VIA is set to make the keypad's rows inputs, and the columns outputs (see accompaning diagram later). The input bits are then polled by reading the keypad I/O port and masking out the column bits. If the result contains a zero, the value is temporarily stored, and the DDR is changed to make the rows outputs and the columns inputs. The above is then repeated, with the exception that if all the input bits are one, the previous reading is discarded and presumed to have occured because of outside interferance, e.g., a passing car, etc. If one of the bits is non-zero, the current value is added to the previous one, giving a value unique to the key pressed. This value is stored in the entry buffer and then checked to see if it is one of the "enter" keys. If it is, the password is checked against the stored ones. Otherwise, the X Register is incremented, and the routine repeated, after ensuring that all the buttons on the keyboard have been released.

Here is a table of values receivable from the keypad:

Key	0	1	2	3	4	5	6	7	8	9	*	#
Value	AF	DD	BD	7D	DB	BB	7B	D7	B7	77	CF	6F

(NB All values are in hex.)

Name : `ckpswd`
Entry conditions : Buffer contains entered password
Exit conditions : As detailed below
Function : Compare entered password with stored ones

First of all, a zero byte is placed at the end of the entered code number to mark the end of it, and the X Register is set to zero. Then the next character is loaded into the Accumulator, and compared to zero. If it is zero, and X isn't zero then the correct password has been entered and control is passed to the alarm de-arm routine, `pswdok`. If X is zero, the entry is disallowed and control is returned to `keypad`. If the character isn't zero, it is compared with the corresponding character in the stored password. If they aren't the same the character is compared to the alternative password.

If they are the same in either case, the X Register is incremented, and the next character is checked. If neither attempt is matched, the number of tries is decremented and if zero, the alarms are turned on. If the number of tries is greater than zero, another attempt is allowed by jumping to the `keypad` routine.

Name : `pswdok`
Entry conditions : None
Exit conditions : No exit from this routine
Function : Deactivate alarm

First Timer1 is turned off (to prevent any subsequent Timeout interrupts). Then sixty four is stored in the I/O register to turn off the alarm set LED and enable the ignition. Then the program idles in an infinate loop.

Name : `timout`
Entry conditions : None
Exit conditions : As detailed below
Function : Handles the interrupts from Timer1

First of all, the Accumulator, X Register, and Y Register are preserved by storing them in zero page. Next the 1/100 second counter is checked to see if it is zero. If it is, the second counter is checked. If not, the 1/100 second counter is decremented and the routine returns. If the second counter isn't zero, it is decremented, the 1/100 second counter set to 100 and the routine returns. If the second counter is zero, the clock flag is loaded into the accumulator. If set (i.e., the timer was set using the `delay` routine) the routine is left, but a different exit routine is used to the two above. If the clock bit is zero, (i.e., the timer was set using the `timer` routine) the alarms are turned on. The first exit routine, used for all returns except for the one detailed above, is short and simple. The Registers are restored, and the interrupt handler left by the use of an `RTI` (ReTurn from Interrupt) instruction. The second is slightly more complicated because it has to alter the return address to jump over the infinate loop in the `delay` routine. First the Processor Status is popped from the stack and stored elsewhere, and the return address transfered into the zero page workspace. Two (the length of the `JMP wait` instruction -1 in `delay`) is added to the address, and it is returned to the stack. The Registers are restored, and the routine is left by means of an RTS (ReTurn from Subroutine) instruction. The reason why two is added to the return address as opposed to three (the length of a `JMP` instruction) is that an. `RTS` instr uction automatically adds one to the return address, whereas `RTI` doesn't.

(NB the listing shows a slightly modified version of `timout` which determines whether timerl or timer2 caused the interrupt. This feature isn't used by this project and is therefore ignored from here on.)

Name : `alarms`
Entry conditions : None
Exit conditions : None
Function : Turns on the alarm signal

After initialising the X Register to six, the alarm bits in the I/O port are set to one, i.e., the lights and horn are turned on. After a pause of ten seconds—by calling `delay` with the Accumulator equal to ten—the alarm bits are reset to zero. After a further delay of two seconds, the alarm bits are set to one again. This is repeated 6 times (i.e., for about one minute). The alarm then resets itself by jumping to `reset`.

Name : `delay`
Entry conditions : Accumulator holds number of seconds delay required
Exit conditions : None
Function : Causes a variable length delay

The Accumulator is stored in a location called `secs`. `secs` is then decremented by one to take into account the fact that zero counts as a second. This isn't a bug, it is merely the way the routine works. The 1/100 second counter is then initialised to one hundred and the delay bit in `flags` is set to one. A call is then made to `timon`, and then an infinate loop is entered. This loop is terminated when the appropriate delay has passed by the timeout interrupt. The timer is turned off by `timoff`, and the subroutine returns to the main program.

Name : `timer`
Entry conditions : Accumulator holds number of seconds to be timed
Exit conditions : Timer1 set up to count for Accumulator seconds
Function : Starts a variable length clock

The Accumulator is stored in a location called `secs`. `secs` is then decremented by one to take into account the fact that zero is counted as a second. This isn't a bug, it is merely the way the routine works. The 1/100 second counter is then initialised to one hundred, and the delay bit in `flags` is reset to zero.

Name : `timon`
Entry conditions : None
Exit conditions : Timeout interrupt flag set
Function : Enables an interrupt to occur at timerl timeout

The Accumulator is loaded with one hundred and ninety two, and this is stored in the VIA's Interrupt Flag Register (this clears any unserviced interrupts), and in the Interrupt Enable Register.

Name : `timoff`
Entry conditions : None
Exit conditions : Timeout interrupt flag reset
Function : Disables interrupt at timerl timeout

The Accumulator is loaded with one hundred and ninety two and this is stored in the VIA's IFR. This acknowledges the interrupt and clears it. Then the Accumulator is loaded with sixty four and this is stored in the VIA's IER.

Testing the System

The testing of this system was carried out in two parts: the first part consisted of the testing of the hardware, and the second the testing of the software. The testing was carried out in this order because for the software to work, the hardware has to be working correctly; if the software was tested first and the system didn't work correctly, it would be impossible to ascertain the source of the error without checking the hardware. Therefore by testing the hardware first, any errors can automatically be attributed to the software.

Testing the Hardware

After a final visual inspection of the boards (to make sure that there were no shorted tracks, etc.), power was applied to the system, and, using a multimeter, the supply rails to the ICs were checked. These were found to be correct. Equipment used in the testing of this project included a dual trace Tektronix oscilloscope, an RS logic probe with built-in pulse stretcher, and a Radio Shack 50 kΩ/volt multimeter.

At this point, it is worth explaining what happens when the 6502 is reset or powered up. This will help explain the logic behind my testing strategy. Following a positive going edge on the reset pin ($\overline{\text{RST}}$), the CPU initiates the reset sequence. After a system initialisation time of six clock cycles, the interrupt bit in the Processor Status register is set, and the Program Counter is loaded with the contents of the address pointed to by the reset vector at locations &FFFC, &FFFD (lo byte, hi byte), i.e., a `JMP (&FFFC)` is executed, and the program which starts at that location run. In this project the reset vector points to &F000, the start of the ROM.

After power-up, the ROM $\overline{\text{CS}}$ was tested, this should have been low, but it was high. The power was turned off and re-applied. Again the ROM $\overline{\text{CS}}$ was high. Further investigation showed the RAM $\overline{\text{CS}}$ to be low. My first thought was that the addressing logic had been incorrectly designed, so this was the next thing to be tested. The idea was to remove the CPU and simulate the various address that the logic decoded. To do this, each of the address lines that was required to be zero was connected to ground and the ones left unconnected, as unconnected TTL (should) float high, and therefore produce a logic one. With no pins connected to ground (i.e., &FFFF was being "accessed") the ROM $\overline{\text{CS}}$ should be low, which it was. The other $\overline{\text{CS}}$s were also checked to ensure that they were high, which they were. The other addresses were checked in a similar manner, and all were found to be correct, so the address decoding was working correctly.

Seeing as though the address decoding was working, I assumed that the CPU was getting the wrong data from the ROM so the data bus was checked. This involved switching the multimeter to Ohms and placing one one probe on the data line under test on the ROM and the other on the corresponding pin on the CPU. A very low resistance (20.1 Ohms) indicates that the connection is sound. This was repeated for each of the eight lines in the data bus, and each was found to be connected correctly. This opportunity was taken to check the rest of the lines, that is the address and control buses. These were also found to be OK.

Next to be checked was the reset circuitry. To do this a scope probe was connected to the $\overline{\text{RST}}$ pin of the CPU and power applied. If all was OK, then $\overline{\text{RST}}$ should go low for about 100 msec, then go, and stay, high. This was found to be the case, so the reset cicuitry was working.

The only thing left was the time base (or clock) generation circuitry. To test this, a scope probe was connected to the clock input pin of the CPU (Φ_0). However, the scope was not quick enough to measure the 1 MHz clock. I was stumped for a while, until I remembered that a computer I was fixing (a Commodore Pet) also used a 1 MHz clock. So I made up a lead which led from the clock input pin in the Pet to the clock input in my project, and applied the power. Success!! At last I had narrowed down the fault, and all that remained was to discover exactly what was wrong with the clock generator. The circuit was compared to that in the data sheet, and was found to be correct, so I checked the layout of the board to ensure that what I had wired up corresponded to the circuit diagram, which it did. The only thing that was different was that the circuit in the spec sheet had specified a 7404, and I was using the higher spec 74LS04, I wouldn't have thought that a higher spec chip would have affected the clock so adversly, but a

7404 was ordered and inserted into the appropriate place in the board (IC3). With the 7404 in position, the power was re-applied and the ROM $\overline{CS}$ tested as described above. It was, as expected, LOW, which meant that the clock generator was now working correctly. To verify the correct operation of the clock, the clock input pin of the CPU (Φ_0) was connected to the logic probe. Both LEDs were lit, indicating that a fast square wave was being measured. (I still don't know why the 74LS04 was unsuitable, but I suspect it's something to do with the Schottky inputs on the LS family.)

To test the rest of the hardware, a simple testing program was written. After initialising the VIA, the program enters an infinate loop. With one end of a probe attached to ground, the $\overline{NMI}$ and $\overline{IRQ}$ pins were touched, one at a time. The program was designed so that an interrupt would toggle the state of the LEDs (which default to off in the test program). If the hardware is working correctly, the LEDs should change states every time either of the $\overline{NMI}$ or $\overline{IRQ}$ pins are touched. The pins were touched and the LEDs did change states, proving that all the hardware was working correctly.

The VIA was working because it controlled the LEDs correctly, the ROM was working because the system was acting under the influence of the program, and the RAM was working because of the way the 6502 handles interrupts. When an interrupt occurs, the PC is pushed onto the stack, followed by the SR (Status Register), and on return from an interrupt, the above are popped from the stack and program control resumes from where it left off. For the PC and SR to be stored and restored, the RAM has to be working, which it was. The numeric keypad's testing is included in the software testing section (as it's really the decoding software that ensures correct operation).

Testing the Software

The logic probe assisted with most of the software testing, although the on-board LEDs also played a part. (NB Because of the nature of this project, I didn't think it was appropriate to "prove" that a routine works by using "before" and "after" screen dumps (also highly impractical: no screen!). I decided it best to relate my testing in a narratative manner—similar to the hardware testing section—stating expected, and actual results, and any corrections necessary before retesting. This is a better way of documenting test results as it allows the reader to follow my line of thinking, and therefore, testing strategy better.)

Looking at the program listing, it is the subroutine `delay` which does the first thing, the `reset` routine being included in the hardware test. For correct operation to be indicated, a logic probe connected to the $\overline{\text{IRQ}}$ pin of the CPU should indicate a high state, with regular (centisecond) low pulses for about one minute, and then stay high. Unfortunately, the $\overline{\text{IRQ}}$ went, and stayed low. At first I thought that the interrupt wasn't being serviced, so I consulted the listing. This was found not to be the case. I then disassembled the Beeb's OS to see how it serviced timer interrupts and discovered that it wrote &C0 to the IFR. Reading the 6522 data sheet, I found out that writing a binary one to the MSB and the bit which flags what part of the VIA caused the interrupt acknowledges and clears the interrupt. So I added the appropriate code to the beginning of the interrupt service routine, and retested the system. This time, everything acted as anticipated—or so I thought. At the end of the minute, instead of staying high, the $\overline{\text{IRQ}}$ pin stayed low.

My first thought was that the `timoff` subroutine wasn't working correctly so I spent quite a long time reviewing the algorthm and code for it, but all appeared to be OK, so I wondered if the routine was actually being executed. To test this I added a short routine to turn on the LEDs, and retried the program. When `timoff` is entered, the LEDs would light up, but they didn't, indicating that `timoff` wasn't even being entered. So I mentally dry ran `leave2` in the interrupt handler. All appeared to he correct, but further and more detailed investigation led me to discover that the `RTS` instruction added one to the return address, whereas the code I had written assumed that nothing was added to the return address. I altered the code to add two, instead of three, to the return address, and re-ran the program. This time the LED did light, and the $\overline{\text{IRQ}}$ line stayed high after the one minute pulsing.

The next routine to be tested was `alarm`. This was tested next as it was used to check on the correct operation of `maloop`. What should have happened was the alarm LEDs would go on for ten seconds, go off for two, and then on again, five more times, and the $\overline{\text{IRQ}}$ line should pulse similar to before. However, after the one minute delay, the LED stayed on, and the $\overline{\text{IRQ}}$ line went, and stayed low. Examining the listing didn't produce any answers, so I added a bit of code that would turn on the LEDs on the second execution of delay. Then I powered up the alarm, and waited for the LEDs to light. If they lit I knew that the bug was further on in the program, and so I moved the LED lighter bit of code to the next instruction in the algorithm. All was OK until I placed the lighter bit at the beginning of the interrupt handler, when the LED failed to light, indicating that the interrupt handler wasn't even being entered. On discovering this, I suddenly remembered that one of the things the 6502 does when an interrupt occurs is to disable any further interrupts. Normally this has no effect, as the `RTI` instruction restores all flags to their pre-interrupt state. However, when the interrupt handler turns off the timer, it exits via an `RTS`, which doesn't restore the Processor Status. So I made the required corrections to the end of interval exit routine, and retested it. This time the LEDs flashed as required. Of all the bugs in the program, this was the most bewildering, because the timer had already been (partly) tested with favourable results.

Next, the keypad testing program was typed in and ran, but this program differed from the others in that it was executed on my Beeb, not on my project. (Note that the test program uses Port B in the Beeb, whereas my project uses Port A. This is because Port A in the Beeb is dedicated to the printer port.) The keypad was connected to its connector, and the test cable was connected to the Beeb's user port. I had previously drawn up a table of expected values and part of this program's job was to see if my calculations were correct. It wouldn't have mattered if they had been wrong, so long as each button on the keypad resulted in a different number being displayed. At this stage, the testing of the keypad was only partially successful. Values that didn't relate to my pre-determined ones in any way were being printed. My first thought was that the masks were wrong, and so, away from the keyboard, I rewrote the keypad routine from scratch, ensuring that the masks were correct (the masks in this rewritten version

were different to those used in the original program). The amended input routine was tested, but still produced errornous results. Closer comparison with the printed values and my pre-determined ones showed that several of the bits were always zero, so the keypad hardware was slightly modified, the modification being the inclusion of 4.7 kΩ resistors between 5V and each keypad bit, to tie them up to logic 1. The program was re-run, and this time every button produced a different value—all of which matched my pre-determined ones. The modifications were incorperated into the main program, and testing continued onto the next stage.

Being such a small and simple routine, `pswdok` was next on the testing list. The routine was run, and the output lines checked using the logic probe. All was as anticipated, i.e., all the bits were low, with the exception of the ignition bit, which was high.

Next to be tested was `maloop`. To do this, the program was run, and each of the inputs (i.e., boot, bonnet, alarm case, and doors) was touched to ground and the result noted. In the case of the door input being grounded, the $\overline{\text{IRQ}}$ should pulse, indicating that the keypad subroutine had been entered, and in the other cases, the LEDs should flash, indicating that the alarm subroutine had been entered; this is what hegpaned..

The last subroutine to be tested was `ckpswd`. Because everything up to this point had been debugged, I decided to kill two birds with one stone, and test `ckpswd` in conjunction with the other routines. So the EPROM was programmed with the complete OS and tested. After the one minute delay, each of the inputs was grounded, the door one being grounded last. Everything was OK, which was to be expected seeing as though I had already tested the other routines, albeit in isolation from one another.

To test the `ckpswd` routine, I typed in the first code number (32991), followed by the hash key. This should have resulted in the alarm being de-activated by the `pswdok` subroutine, but instead it tried to collect another code number, indicating that it thought that I had typed in the wrong number. On a hunch I typed in the alternative code number (28645). This time the alarm turned itself off. I assumed that the data for the code number was incorrect, and looked at the program to check this. This was the case: I had misread BD as 8D when copying down the number from the code number creation program. I made this correction and retried the program. This time it accepted the code number. A final check was made to ensure that it allowed only three entries, and that it only allowed thirty seconds to enter the number(s). This concluded the testing.

Future developments

It is hoped that one day I will get round to designing the remaining hardware to convert this project in to a stand alone, single board computer. This will entail adding some form of output display (probably seven segment LEDs), more RAM, and a higher quality keypad, in the form of a hex keypad. The OS will also have to extensively redesigned, the main alteration being the inclusion of a (relatively) simple machine code monitor. Luckily, I've already written most of the routines for the monitor, in a program I am writing for my computer, which I have already mentioned.

Errata

Due to time pressures some un-corrected errors have been published. These are as follows:

i) The connections to the switches in the upper right hand corner of the Memory and I/O board diagram have been moved down to two holes below the keypad connector, and are now connected to ground, not five volts.

ii) The keypad connector is incorrectly labled in the same diagram. The correct connections are (top to bottom): Unconnected, Row one, Row two, Row three, Row four, Column one, Column two, Column three.

iii) Zero volts now enters the board from a connector one hole up from the keypad connector.

iv) The first decision box in the Main loop flowchart should read "All bits = 0", and the decision box which follows a 'No!' answer to the above should read "Just door open bit = 1".

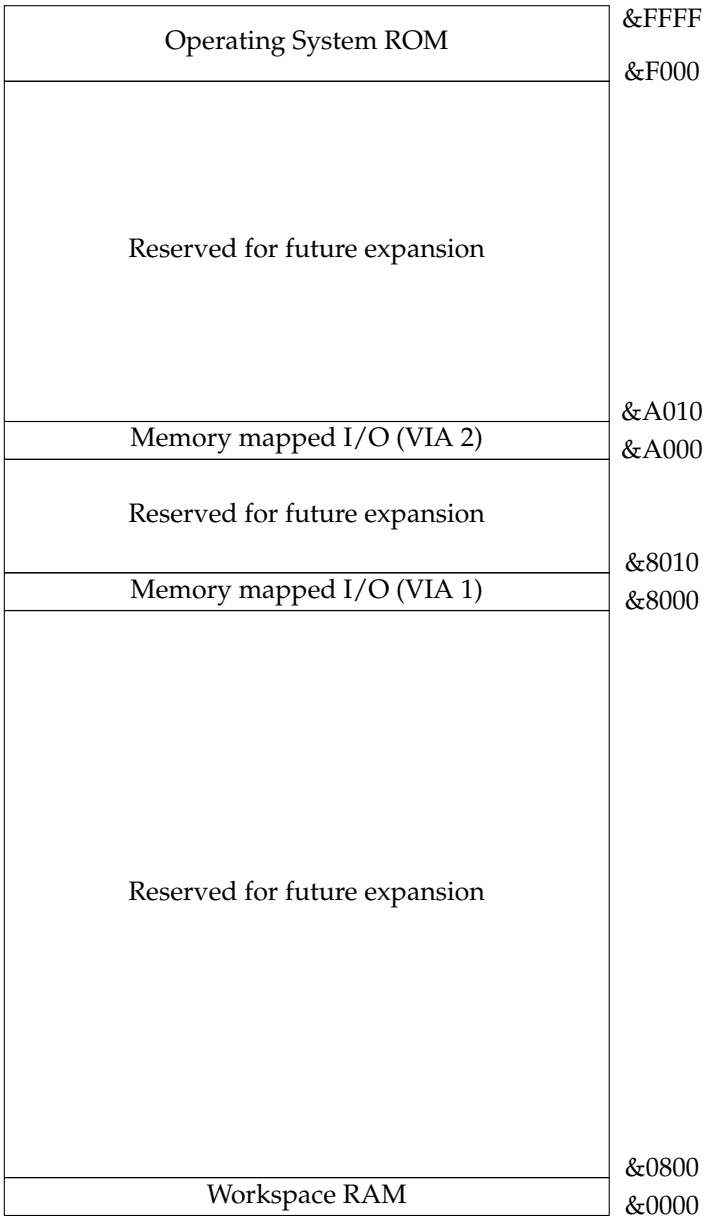
v) All occurences of "All bits = 0" in the enter password flowchart should read "All bits = 1".

vi) The box that reads "Retrieve return address from stack, & add 3" in the interrupt handler flowchart should read "Retrieve return address from stack, & add 2". Also note that the flowchart doesn't show the logic for the timer2 timeout handler for the reasons previously stated.

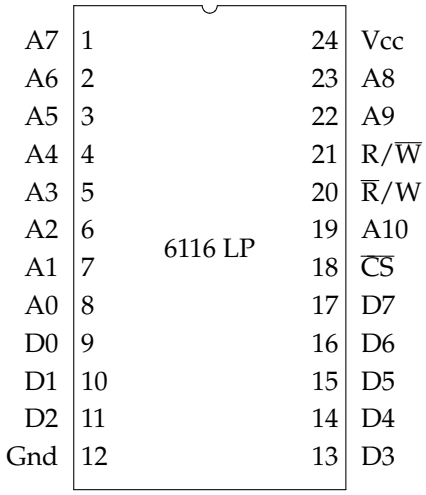
Glossary

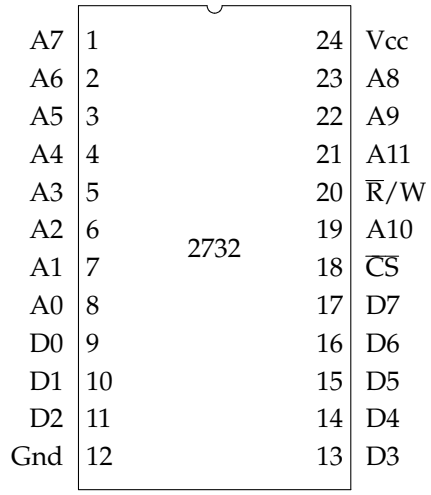
ADL	An initialism for Address Decoding Logic.
CPU	An initialism for Central Processing Unit. The heart of any micro based machine.
$\overline{\text{CS}}$	An initialism for Chip Select. When this is low, the chip it is connected is selected. $\overline{\text{CS}}$ is usually produced by address decoding logic.
I/O	An abbreviation for Input/Output.
$\overline{\text{IRQ}}$	An initialism for Interrupt ReQuest. This interrupt may be ignored by the CPU.
LSB	An initialism for Least Significant Byte. It is the lower eight bits of a sixteen bit word. Also means Least Significant Bit. Its context should determine which meaning is appropriate.
LSN	An initialism for Least Significant Nibble. The lower four bits in a byte.
MSB	An initialism for Most Significant Byte. It is the upper eight bits in a sixteen bit word. Also stands for Most Significant Bit. Its context should determine which meaning is appropriate.
$\overline{\text{NMI}}$	An initialism for Non-Maskable Interrupt. This interrupt may be not ignored by the CPU.
VIA	An initialism for Versatile Interface Adapter.
&	Means a hexadecimal number is to follow.
•	On the chip pinout sheets this means (logical) AND.
+	On the chip pinout sheets this means (logical) OR.

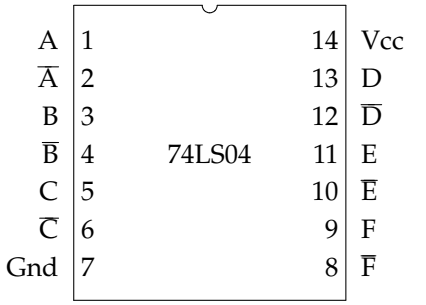
Memory Map

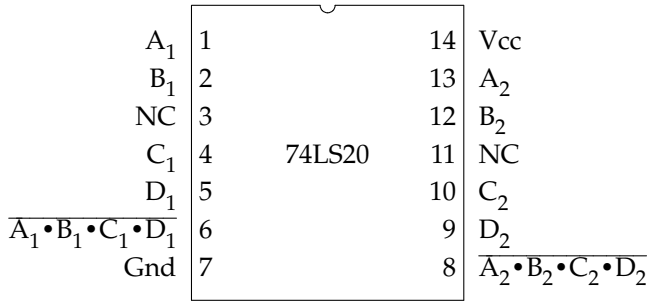

Operating System ROM
&FFFF
&F000
Reserved for future expansion
&A010
Memory mapped I/O (VIA 2)
&A000
Reserved for future expansion
&8010
Memory mapped I/O (VIA 1)
&8000
Reserved for future expansion
&0800
Workspace RAM
&0000

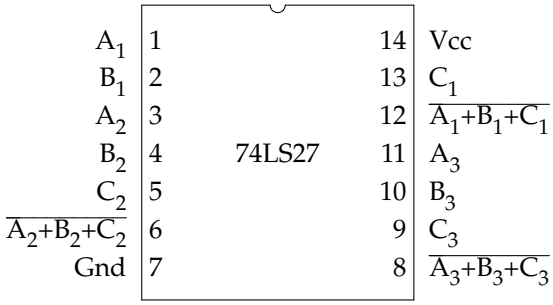
Chip Pinouts (Sheet One)

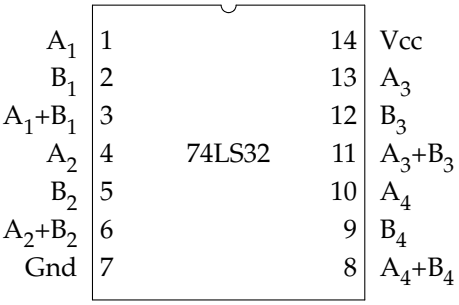




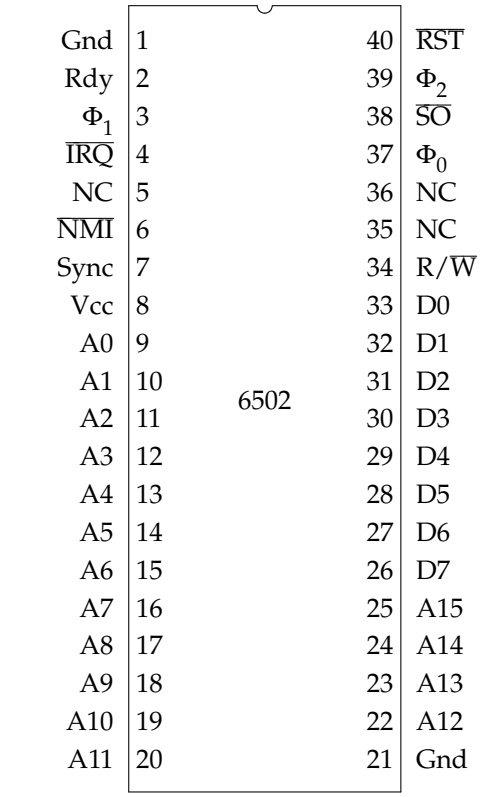


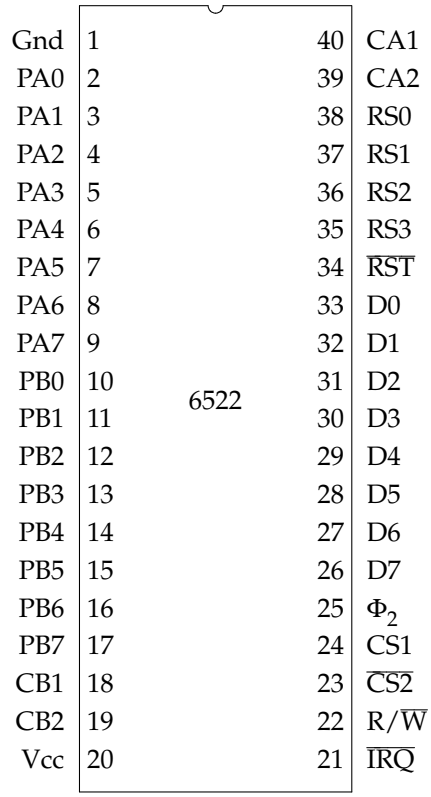


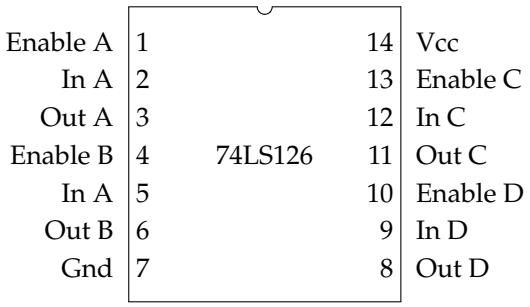


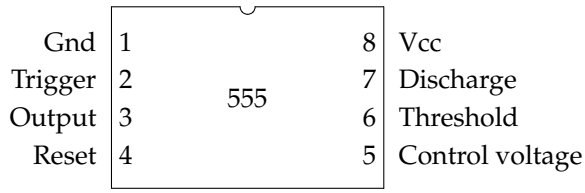


Chip Pinouts (Sheet Two)

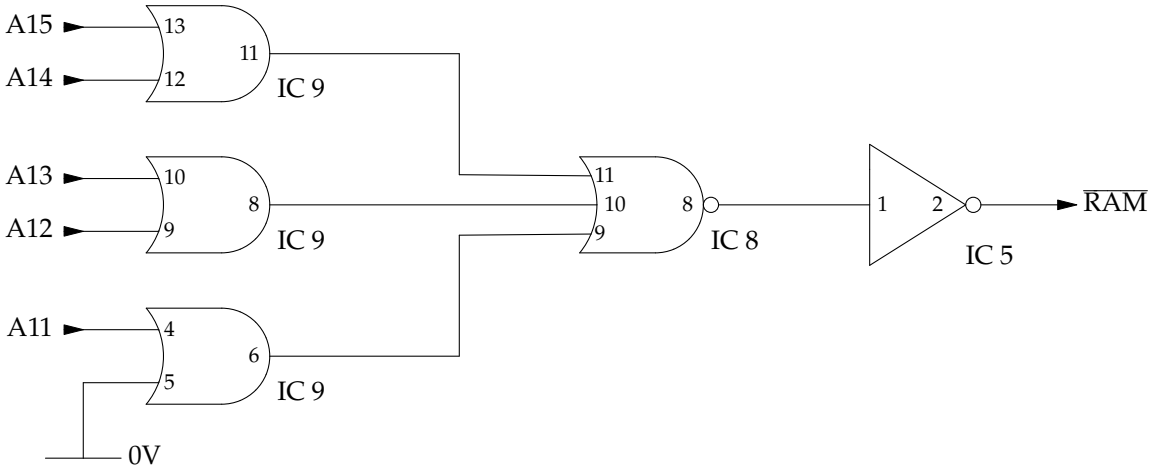




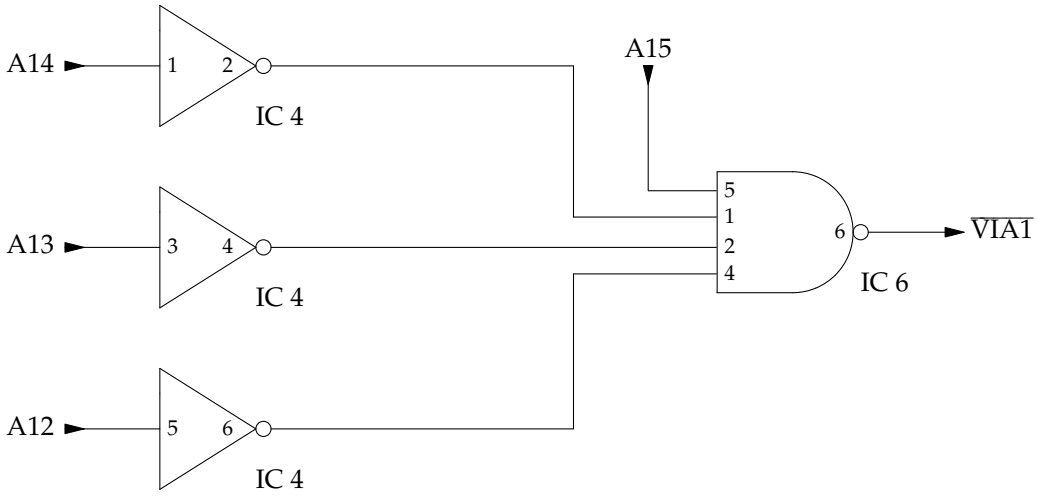




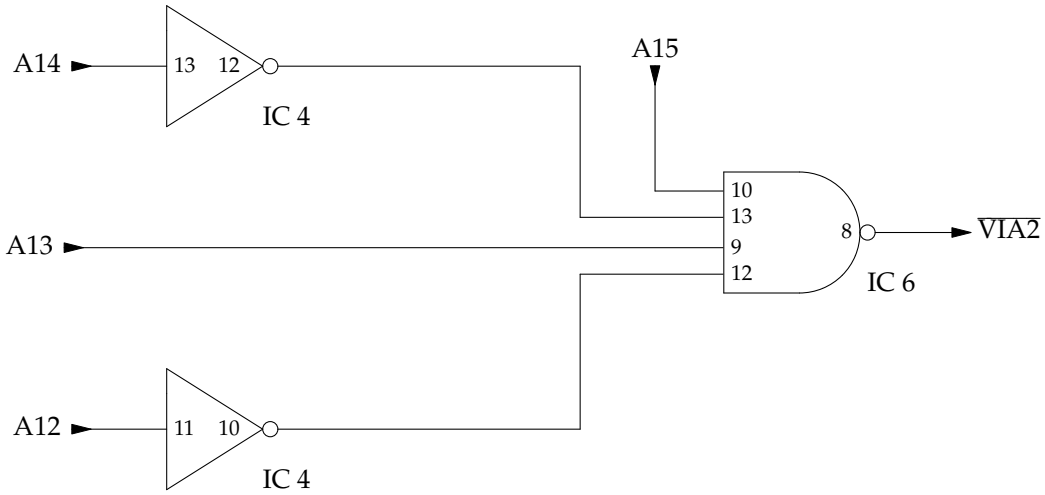
RAM Address Decoding Circuit



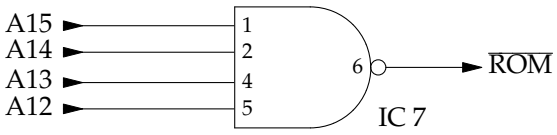
VIA 1 Address Decoding Circuit



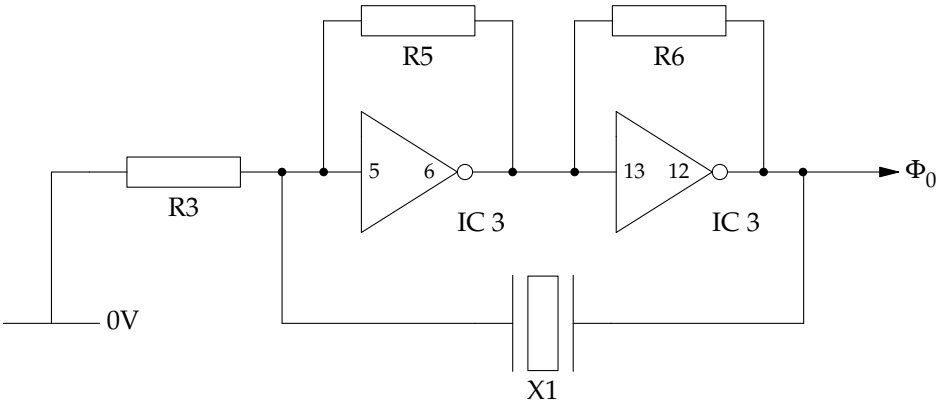
VIA 2 Address Decoding Circuit



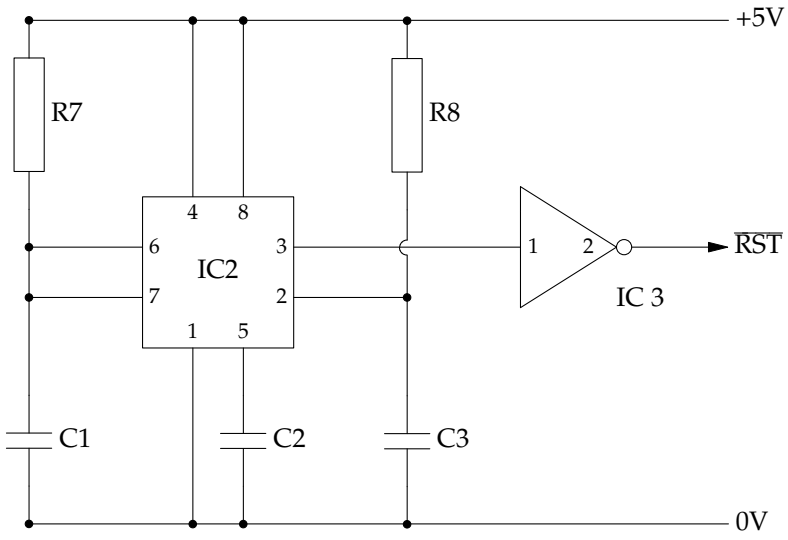
ROM Address Decoding Circuit



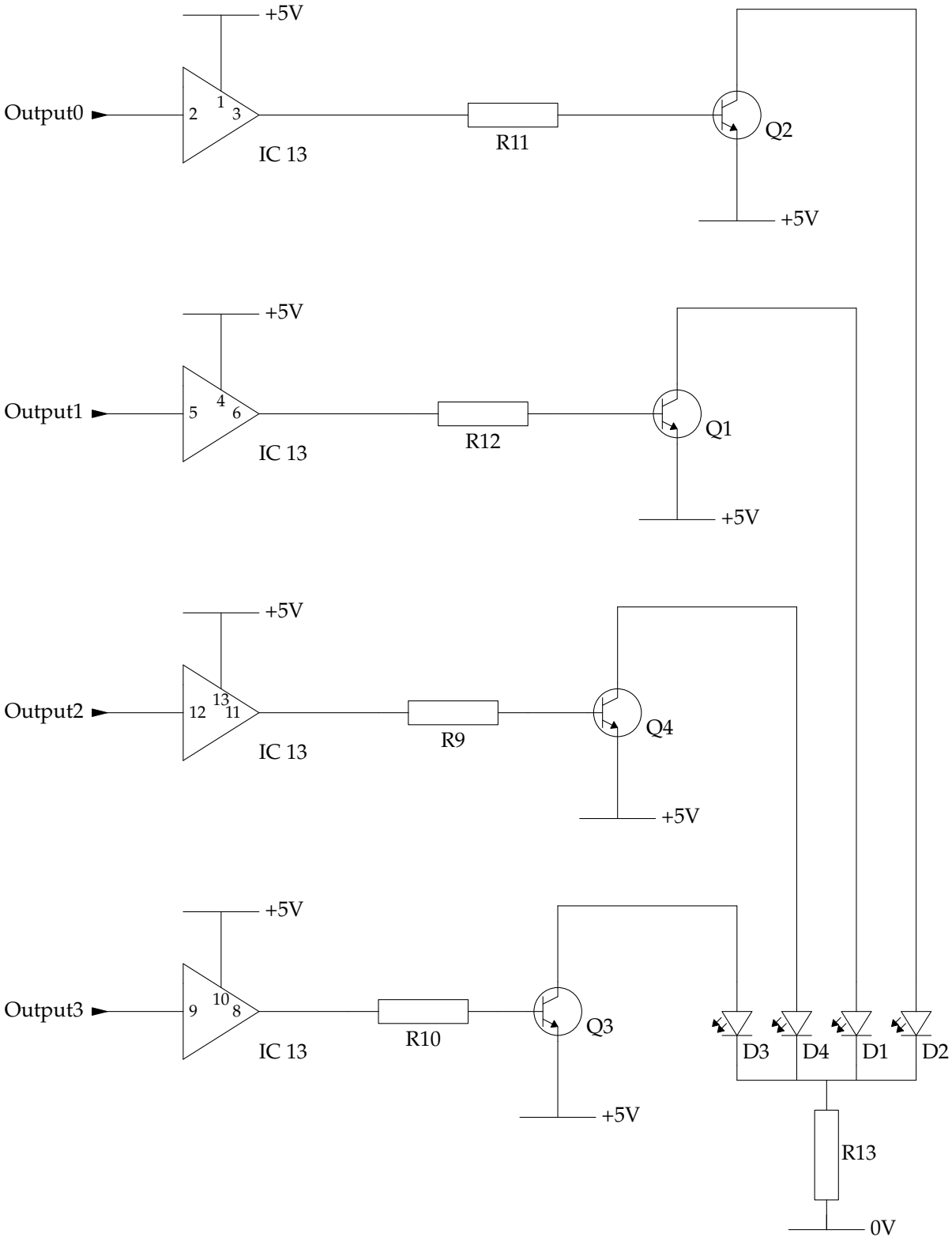
Clock Generation Circuit



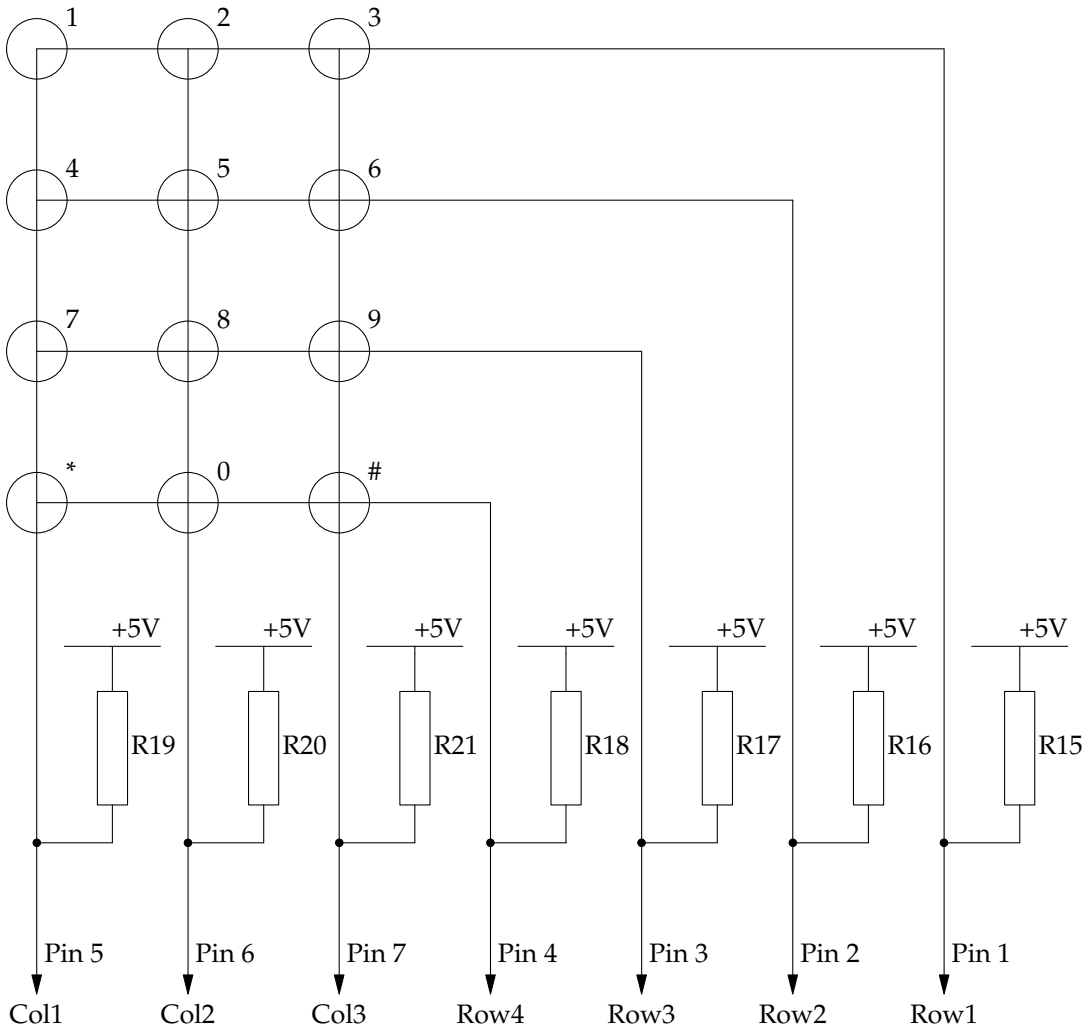
Power-Up Reset Circuit



Output Circuit



Keypad Circuit



```
10 REM **********************************
20 REM *** Password generation program ***
30 REM *** For use with Burglar Alarm  ***
40 REM ***   (C) 1987, 2026 Rich Teer  ***
50 REM **********************************
60 PROC_init
70 MODE 0
80 PROC_centre("PASSWORD CREATOR", 5)
90 INPUT TAB(0, 10); "Please enter required Password: " password$
100 PRINT
110 PRINT "The encoded Password is: ";
120 FOR loop = 1 TO LEN(password$)
130   PRINT "&";
140   value = FN_encode(MID$(password$, loop, 1))
150   PRINT; ~value; ", ";
160   NEXT loop
170 VDU 127
180 PROC_centre("Press <RETURN> when ready ", 15)
190 REPEAT UNTIL GET = 13
200 END
210
220 DEF PROC_init
230 *TV 0 1
240 DIM keys(9)
250 FOR loop = 0 TO 9
260   READ data
270   keys(loop) = data
280   NEXT loop
290 ENDPROC
300
310 DEF PROC_centre(x$, y)
320 x = 40 - (LEN(x$) DIV 2)
330 PRINT TAB (x, y); x$
340 ENDPROC
350
360 DEF FN_encode(start$)
370 index = (ASC(start$)) - 48
380 = keys(index)
390
400 DATA &AF, &DD, &BD, &7D, &DB, &BB, &7B, &D7, &B7, &77, &CF, &6F
```

```
; A Level Computing Science Project: Car Burglar Alarm.
;
; This program tests the hardware.  After initialising
; the hardware, the program waits in an infinite loop
; waiting for an interrupt, either with an IRQ or an NMI.
; In the interrupt service routine, the output port is
; inverted, thereby causing the LEDs attached to it to
; "blink" when an interrupt occurs.  The nature of this
; program tests all the hardware with the exception of
; the keypad, which was tested using a different method
; which didn't involve any system hardware.
;
; (C) 1987, 2026 Rich Teer.


flags   *       &0000           ; Various flags.
tries   *       &0001           ; Number of attempts at entering code number.
sec100  *       &0002           ; 1/100 second timer.
secs    *       &0003           ; Number of seconds before timeout.
pclo    *       &0004           ; PC lo.
pchi    *       &0005           ; PC hi.
acc     *       &0006           ; Accumulator store.
xreg    *       &0007           ; X Register store.
yreg    *       &0008           ; Y Register store.
temp    *       &0009           ; Temporary storage.
enterd  *       &0200           ; Entered password stored from here.

v1iorb  *       &8000           ; VIA 1 I/O Register B.
v1iora  *       &8001           ; VIA 1 I/O Register A.
v1ddrb  *       &8002           ; VIA 1 Data Direction Register B.
v1ddra  *       &8003           ; VIA 1 Data Direction Register A.
v1t1cl  *       &8006           ; VIA 1 Timer1 Counter Low.
v1t1ch  *       &8007           ; VIA 1 Timer1 Counter High.
v1acr   *       &800B           ; VIA 1 Auxiliary Control Register.
v1ifr   *       &800D           ; VIA 1 Interrupt Flag Register.
v1ier   *       &800E           ; VIA 1 Interrupt Enable Register.


        ORG     &F000           ; Assemebled code to start at &F000.

reset   SEI                     ; Disable interrupts.
        CLD                     ; Clear Decimal mode.
        LDXIM   &FF             ; Load X Register with 255,
        TXS                     ; and reset Stack Pointer.
        LDAIM   &80             ; Load Accumulator with 128,
        STA     v1ier           ; and disable interrupts from VIA.
        LDAIM   &1F             ; Load Accumulator with 31,
        STA     v1ddra          ; and initialise DDRA.
        LDAIM   &F0             ; Load Accumulator with 240,
        STA     v1ddrb          ; and initialise DDRB.
        LDAIM   &00             ; Load Accumulator with 0 to
        STAZ    flags           ; zero all flags,
        STA     v1iora          ; zero I/O port A, and
```

```
        STA     v1iorb          ; zero I/O port B.
        LDAIM   &10             ; Load Accumulator with 16, i.e., 10,000 MOD 256,
        STA     v1t1cl          ; and store in counter low.
        LDAIM   &27             ; Load Accumulator with 39, i.e., 10,000 DIV 256,
        STA     v1t1ch          ; to set up Timer 1 to count for 10,000 cycles.
        LDAIM   &40             ; Load Accumulator with 64, and
        STA     v1acr           ; set Timer1 to continuous operation.
        CLI                     ; Enable interrupts.
goto    JMP     goto            ; Infinite loop for the purposes of testing.

timout
nmi     LDA     v1iorb          ; Load Accumulator with I/O port B.
        EORIM   &FF             ; Perform a one's complement on it,
        STA     v1iorb          ; and store back in I/O port B.
        RTI                     ; Return from interrupt.

        LNK     bugos2          ; Continue assembly with the vectors.
```

```
; A Level Computing Science Project: Car Burglar Alarm.
;
; Program to test the numeric keypad, designed to run on a
; BBC Micro, Model B.  Note the change of VIA addresses
; and the addition of OSWRCH.  This is a routine in the MOS
; to print the character in the Accumulator on the screen.
; The subroutine phex was borrowed from my on-going long
; term project, a 65C02 machine code monitor.
;
; (C) 1987, 2026 Rich Teer.


temp    *       &0070           ; Temporary storage.

v1iorb  *       &FE60           ; VIA 1 I/O Register B.
v1iora  *       &FE61           ; VIA 1 I/O Register A.
v1ddrb  *       &FE62           ; VIA 1 Data Direction Register B.
v1ddra  *       &FE63           ; VIA 1 Data Direction Register A.
v1t1cl  *       &FE64           ; VIA 1 Timer1 Counter Low.
v1t1ch  *       &FE65           ; VIA 1 Timer1 Counter High.
v1t2cl  *       &FE68           ; VIA 1 Timer2 Counter Low.
v1t2ch  *       &FE69           ; VIA 1 Timer2 Counter High.
v1acr   *       &FE6B           ; VIA 1 Auxiliary Control Register.
v1ifr   *       &FE6D           ; VIA 1 Interrupt Flag Register.
v1ier   *       &FE6E           ; VIA 1 Interrupt Enable Register.
oswrch  *       &FFEE           ; OSWRCH.


        ORG     &2000           ; Assemebled code to start at &2000.

        LDAIM   &0C             ; Load Accumulator with 12,
        JSR     oswrch          ; and clear screen.

keypad  LDAIM   &E0             ; Load Accumulator with 31, and
        STA     v1ddrb          ; change DDRB's mask.
        LDAIM   &00             ; Load Accumulator with 0,
        STA     v1iorb          ; and make all outputs low.
        LDA     v1iorb          ; Load Accumulator with I/O Register B.
        ANDIM   &1F             ; Mask out output bits.
        CMPIM   &1F             ; Compare to 31.
        BEQ     keypad          ; If all bits are 1, check again.
        STAZ    temp            ; Store Accumulator in temp.
        LDAIM   &1F             ; Load Accumulator with 224, and
        STA     v1ddrb          ; change DDRB's mask.
        LDAIM   &00             ; Load Accumulator with 0,
        STA     v1iorb          ; and make all outputs low.
        LDA     v1iorb          ; Load Accumulator with I/O Register B.
        ANDIM   &E0             ; Mask out output bits.
        CMPIM   &E0             ; Compare to 224.
        BEQ     keypad          ; If all bits are 1, check again.
        CLC                     ; Clear Carry.
        ADCZ    temp            ; Add temp to the Accumulator.
        JSR     output          ; Print number on screen,
```

```
        JMP     keypad          ; and do again.

output  PHA                     ; Preserve Accumulator.
        LDAIM   &1A             ; Load Accumulator with 26, i.e., Cursor Home,
        JSR     oswrch          ; and print on screen.
        PLA                     ; Restore Accumulator.
        JSR     phex            ; Print Accumulator as a 2 digit hex number,
        RTS                     ; and return.

phex    PHA                     ; Preserve Accumulator.
        LSRA                    ; This bit
        LSRA                    ; shifts the MSN
        LSRA                    ; into
        LSRA                    ; the LSN.
        JSR     conv            ; Convert to ASCII.
        PLA                     ; Restore Accumulator.
        ANDIM   &0F             ; Mask out MSN.
conv    CMPIM   &0A             ; If the Accumulator is greater than 10,
        BCS     letter          ; then digit is a letter.
        CLC                     ; Else clear the Carry flag,
        ADCIM   &30             ; add 48,
        JSR     oswrch          ; print on screen,
        RTS                     ; and return.
letter  ADCIM   &36             ; A letter so add 54,
        JSR     oswrch          ; print on screen,
        RTS                     ; and return.

        END
```

```
; A Level Computing Science Project: Car Burglar Alarm.
;
; This file declares all the constants for use with the
; Burglar Alarm Operating System, BugOS.
;
; (C) 1987, 2026 Rich Teer.


flags   *       &0000           ; Various flags.
tries   *       &0001           ; Number of attempts at entering code number.
sec100  *       &0002           ; 1/100 second timer.
secs    *       &0003           ; Number of seconds before timeout.
pclo    *       &0004           ; PC lo.
pchi    *       &0005           ; PC hi.
acc     *       &0006           ; Accumulator store.
xreg    *       &0007           ; X Register store.
yreg    *       &0008           ; Y Register store.
sr      *       &0009           ; Processor Status store.
temp    *       &000A           ; Temporary storage.
enterd  *       &0200           ; Entered password stored from here.

v1iorb  *       &8000           ; VIA 1 I/O Register B.
v1iora  *       &8001           ; VIA 1 I/O Register A.
v1ddrb  *       &8002           ; VIA 1 Data Direction Register B.
v1ddra  *       &8003           ; VIA 1 Data Direction Register A.
v1t1cl  *       &8004           ; VIA 1 Timer1 Counter Low.
v1t1ch  *       &8005           ; VIA 1 Timer1 Counter High.
v1t2cl  *       &8008           ; VIA 1 Timer2 Counter Low.
v1t2ch  *       &8009           ; VIA 1 Timer2 Counter High.
v1acr   *       &800B           ; VIA 1 Auxiliary Control Register.
v1ifr   *       &800D           ; VIA 1 Interrupt Flag Register.
v1ier   *       &800E           ; VIA 1 Interrupt Enable Register.

doors   *       &02             ; Mask to see if a door has been opened.
clock   *       &04             ; Mask to see if timer is a delay or a clock.
timer2  *       &20             ; Mask for Timer2 interrupt.

        LNK     bugos1          ; Continue assembly with the actual BugOS code.
```

```
; A Level Computing Science Project: Car Burglar Alarm.
;
; This file shows the complete Burglar Alarm Operating
; System, BugOS.
;
; (C) 1987, 2026 Rich Teer.


        ORG     &F000           ; Assemebled code to start at &F000.

nmi
reset   SEI                     ; Disable interrupts.
        CLD                     ; Clear Decimal mode.
        LDXIM   &FF             ; Load X Register with 255,
        TXS                     ; and reset Stack Pointer.
        LDAIM   &80             ; Load Accumulator with 128,
        STA     v1ier           ; and disable interrupts from VIA.
        LDAIM   &1F             ; Load Accumulator with 31,
        STA     v1ddra          ; and initialise DDRA.
        LDAIM   &F0             ; Load Accumulator with 240,
        STA     v1ddrb          ; and initialise DDRB.
        LDAIM   &00             ; Load Accumulator with 0 to
        STAZ    flags           ; zero all flags,
        STA     v1iora          ; zero I/O port A, and
        STA     v1iorb          ; zero I/O port B.
        LDAIM   &10             ; Load Accumulator with 16, i.e., 10,000 MOD 256,
        STA     v1t1cl          ; and store in counter low.
        LDAIM   &27             ; Load Accumulator with 39, i.e., 10,000 DIV 256,
        STA     v1t1ch          ; to set up Timer 1 to count for 10,000 cycles.
        LDAIM   &40             ; Load Accumulator with 64,
        STA     v1acr           ; set T1 to free-run mode, T2 to one-shot mode.
        CLI                     ; Enable interrups.
        STAIM   &3C             ; Load Accumulator with 60.
        JSR     delay           ; Wait for 1 minute.
        LDAIM   &03             ; A minute has elapsed, so set up alarm.
        STAZ    tries           ; Initialise number of password attempts.
        LDAIM   &FF             ; Load Accumulator with 255.
        LDXIM   &00             ; Load X Register with 0.
setupl  STAAX   enterd          ; Store 255 at enterd, X.
        INX                     ; Increment X, and
        BNE     setupl          ; if X not 0, loop again.
        LDAIM   &80             ; Load Accumulator with 128.
        STA     v1iorb          ; Turn on alarm-set LED.

maloop  LDA     v1iorb          ; Get data from VIA 1 I/O register B.
        ANDIM   &0F             ; Mask out output bits.
        CMPIM   &0F             ; Check for a zero bit.
        BEQ     t4dor1          ; If not, check if a door has been opened.
        CMPIM   &0E             ; Check if just a door is open.
        BEQ     t4dor2          ; If so, see if this is first occurence.
        JMP     alarm2          ; Another bit is set so activate alarms.
t4dor1  LDAIM   doors           ; Load Accumulator with doors open mask.
        BITZ    flags           ; Check to see if a door has been opened.
```

```
        BEQ     maloop          ; If not, check again,
        JMP     keypad          ; else branch to keypad handling routine.
t4dor2  LDAIM   doors           ; Load Accumulator with doors open mask.
        BITZ    flags           ; Check to see if this first occurence.
        BNE     maloop          ; If not, return to the main loop.
        LDAZ    flags           ; Load Accumulator with flags,
        ORAIM   doors           ; Set doors flag,
        STAZ    flags           ; and store Accumulator in flags.
        LDAIM   &1E             ; Load Accumulator with 30,
        JSR     timer           ; and set timer.
        JMP     maloop          ; Poll input bits again.

keypad  LDXIM   &00             ; Initialise X Register.
keypdl  LDAIM   &E0             ; Load Accumulator with 31, and
        STA     v1ddra          ; change DDRA's mask.
        LDAIM   &00             ; Load Accumulator with 0,
        STA     v1iora          ; and make all outputs low.
        LDA     v1iora          ; Load Accumulator with I/O Register A.
        ANDIM   &1F             ; Mask out output bits.
        CMPIM   &1F             ; Compare with 31.
        BEQ     keypdl          ; If 31, all bits are 1, so check again.
        STAZ    temp            ; Store Accumulator in temp.
        LDAIM   &1F             ; Load Accumulator with 224, and
        STA     v1ddra          ; change DDRA's mask.
        LDAIM   &00             ; Load Accumulator with 0,
        STA     v1iora          ; and make all outputs low.
        LDA     v1iora          ; Load Accumulator with I/O Register A.
        ANDIM   &E0             ; Mask out output bits.
        CMPIM   &E0             ; Compare with 224.
        BEQ     keypdl          ; If 224, all bits are 1, so check again.
        CLC                     ; Clear Carry.
        ADCZ    temp            ; Add temp to the Accumulator.
        STAAX   enterd          ; Store number in buffer.
        CMPIM   &6F             ; Check for a "#".
        BEQ     ckpswd          ; If it is, compare entry with password.
        CMPIM   &CF             ; Check for a "*".
        BEQ     ckpswd          ; If it is, compare entry with password.
        INX                     ; Increment counter.
kypd12  LDA     v1iora          ; Load Accumulator with I/O Register A.
        ANDIM   &E0             ; Mask out output bits.
        CMPIM   &E0             ; Compare to 224,
        BNE     keypdl2         ; and if not 224, check again,
        JMP     keypdl1         ; else get another character from keypad.

ckpswd  CPXIM   &00             ; Compare number of entered characters to 0.
        BEQ     keypad          ; If 0, disallow entry.
        LDAIM   &00             ; Load Accumulator with 0,
        STAAX   enterd          ; and store at enterd, X to mark end of entry.
        LDXIM   &00             ; Initialise X Register.
pswdlp  LDAAX   enterd          ; Load Accumulator with Xth character entered.
        BEQ     pswdok          ; If 0, correct password has been entered.
        CMPAX   psswrd          ; Compare A with Xth character of password.
        BEQ     nxtchr          ; If they are the same, get next character.
        CMPAX   stored          ; Compare A with alternative password.
```

```
        BEQ     nxtchr          ; If they are the same, get next character.
        DECZ    tries           ; They aren't, so decrement number of tries,
        BNE     keypd           ; if tries > 0 then allow another attempt,
        JMP     alarm2          ; else turn on alarms.
nxtchr  INX                     ; Increment X Register,
        JMP     pswdlp          ; and compare the next character.
keypd   JMP     keypad          ; This gets around the out of range branch.

pswdok  JSR     timoff          ; Turn off timer.
        LDAIM   &40             ; Load Accumulator with 64.
        STA     v1iorb          ; Turn on ignition, turn off alarm active LED,
pwoklp  JMP     pwoklp          ; and idle.

timout  STAZ    acc             ; Preserve Accumulator.
        STXZ    xreg            ; Preserve X Register.
        STYZ    yreg            ; Preserve Y Register.
        LDAIM   timer2          ; Load Accumulator with timer2 mask,
        BIT     v1ifr           ; see if timer2 caused interrupt.
        BNE     tm2int          ; If it did, branch to timer2 handler.
        LDAIM   &C0             ; Load Accumulator with 192,
        STA     v1ifr           ; and clear Timer1 interrupt.
        LDAZ    sec100          ; Load Accumulator with 1/100 second counter,
        BEQ     second          ; and if zero check seconds.
        DECZ    sec100          ; Decrement 1/100 second counter.
        JMP     leave1          ; and leave.
second  LDAZ    secs            ; Load Accumulator with second counter,
        BEQ     timeup          ; and if zero take appropriate alarm action.
        DECZ    secs            ; Decrement second counter.
        LDAIM   &64             ; Load Accumulator with 100,
        STAZ    sec100          ; reset 1/100 second counter,
        JMP     leave1          ; and leave.
timeup  LDAIM   clock           ; Load Accumulator with clock mask,
        BITZ    flags           ; and check to see if bit is set.
        BNE     leave2          ; If so, use delay exit routine, else
        JMP     alarms          ; turn on alarms.
leave1  LDAZ    acc             ; Restore Accumulator.
        LDXZ    xreg            ; Restore X Register.
        LDYZ    yreg            ; Restore Y Register,
        RTI                     ; and return.
tm2int  LDAIM   &A0             ; Load Accumulator with 160,
        STA     v1ifr           ; and clear timer2 interrupt.
leave2  PLA                     ; Get Processor Status,
        STAZ    sr              ; and store.
        PLA                     ; Get return address Lo,
        STAZ    pclo            ; and store.
        PLA                     ; Get return address Hi,
        STAZ    pchi            ; and store.
        CLC                     ; Clear carry.
        LDAZ    pclo            ; Retrieve return address Lo,
        ADCIM   &02             ; add 2,
        STAZ    pclo            ; and store.
        LDAZ    pchi            ; Retrieve return address Hi,
        ADCIM   &00             ; adjust for beyond page jumps,
        PHA                     ; and store.
```

```
        LDAZ    pclo            ; Retrieve return address from storage,
        PHA                     ; and store on stack, ready for return.
        LDAZ    sr              ; Get Processor Status Register,
        PHA                     ; and store on Stack.
        LDAZ    acc             ; Restore Accumulator.
        LDXZ    xreg            ; Restore X Register.
        LDYZ    yreg            ; Restore Y Register.
        PLP                     ; Restore Processor Status register,
        RTS                     ; and return.

alarms  PLP                     ; Restore Processor Status register.
        PLA                     ; Clear garbage
        PLA                     ; off Stack.
alarm2  JSR     timoff          ; Turn off timer.
        LDXIM   &06             ; Initialise X Register.
alarm1  LDAIM   &B0             ; Load Accumulator with 176,
        STA     v1iorb          ; and store in I/O Register B.
        LDAIM   &0A             ; Load Accumulator with 10,
        JSR     delay           ; and wait for 10 seconds.
        LDAIM   &80             ; Load Accumulator with 128,
        STA     v1iorb          ; and store in I/O Register B.
        LDAIM   &02             ; Load Accumulator with 2.
        JSR     delay           ; Wait for 2 seconds.
        DEX                     ; Decrement X Register, and
        BNE     alarm1          ; if not 0, loop again.
        JMP     reset           ; Reset the alarm.

delay   STAZ    secs            ; Store number of seconds delay required.
        BNE     dnzero          ; If required delay isn't 0, skip this bit.
        LDAIM   &01             ; Load Accumulator with 1,
        STAZ    sec100          ; and initialise 1/100 sec. counter for debounce.
        JMP     dover           ; Skip over dnzero.
dnzero  DECZ    secs            ; Decrement required delay by 1.
        LDAIM   &64             ; Load Accumulator with 100, and
        STAZ    sec100          ; initialise 1/100 second counter.
dover   LDAZ    flags           ; Load Accumulator with flags.
        ORAIM   clock           ; Set delay bit to 1,
        STAZ    flags           ; and store in flags.
        JSR     timon           ; Turn on the timer.
wait    JMP     wait            ; Wait indefinitely, waiting for timer.
        JMP     timoff          ; Turn off timer, and return.

timer   STAZ    secs            ; Store number of seconds required to be timed.
        DECZ    secs            ; Decrement required time by 1.
        LDAIM   &64             ; Load Accumulator with 100, and
        STAZ    sec100          ; initialise 1/100 second counter.
        LDAIM   clock           ; Load Accumulator with clock mask.
        EORIM   &FF             ; Do an Accumulator = 255 - clock,
        STAZ    temp            ; and store result in temp.
        LDAZ    flags           ; Load Accumulator with flags.
        ANDZ    temp            ; Reset delay bit to 0,
        STAZ    flags           ; and store in flags.

timon   LDAIM   &C0             ; Load Accumulator with 192.
```

```
        STA     v1ifr           ; Clear Timer1 interrupt flag.
        STA     v1ier           ; Enable interrupt from Timeout, and
        RTS                     ; return.

timoff  LDAIM   &C0             ; Load Accumulator with 192.
        STA     v1ifr           ; Clear Timer1 interrupt.
        LDAIM   &40             ; Load Accumulator with 64.
        STA     v1ier           ; Disable interrupt from Timeout, and
        RTS                     ; return.

psswrd  =       &7D             ; 3
        =       &BD             ; 2
        =       &77             ; 9
        =       &77             ; 9
        =       &DD             ; 1

stored  =       &BD             ; 2
        =       &B7             ; 8
        =       &7B             ; 6
        =       &DB             ; 4
        =       &BB             ; 5

        =       &00
        =       " (C) 1987, 2026 by Rich Teer"

        LNK     bugos2          ; Continue assembly with the vectors.
```

```
; A Level Computing Science Project: Car Burglar Alarm.
;
; This file initialises the vectors for my Burglar Alarm
; Operating System, BugOS.
;
; (C) 1987, 2026 Rich Teer.


        ORG     &FFFA           ; Assemebled code to start at &FFFA.

        &       nmi             ; The NMI vector.
        &       reset           ; The power-up reset vector.
        &       timout          ; The IRQ vector.

        END
```