# Vulcan: A Proprietary Cipher of the 1970s 

Algorithm Description and Instant Cryptanalysis
Cornelius Jenkins Riddler
ZJq2eE5A6BidPLyG1xMPTpcTqHa5ms
September 2014


#### Abstract

In the 1970s, Motorola developed a proprietary cipher known internally as Vulcan, implemented this cipher in a custom integrated circuit, and marketed a secure communications system based upon Vulcan under the trade name DVP. In this paper we reveal the Vulcan cipher algorithm and develop an effective real-time ciphertext-only cryptanalytic attack against it. We additionally present as much historical information as we have been able to obtain.


## 1 Introduction

Vulcan is a proprietary self-synchronizing stream cipher developed by Motorola in the mid-1970s for use in secure two-way radios. It was marketed under the trade name DVP, an acronym for digital voice protection.

In this paper we reveal a complete description of the Vulcan cipher and provide cryptanalysis capable of recovering the cryptovariable from a small amount of ciphertext in real time. To our knowledge, such information has never before appeared in the open literature. Internet searches reveal little more than a few pages of sparse and often inaccurate information $[6,7,8]$.

Because Vulcan was developed in secret by Motorola, virtually nothing has been publically known about the details of its design. Although we are now certain of the technical aspects of Vulcan, we will never know the reasons why certain design decisions were made, nor can we be confident of the precise history of its development.

### 1.1 Caveats

Aside from a few marketing brochures, which are increasingly difficult to find nowadays, we are unaware of any substantial technical information that has been published regarding Vulcan or DVP. We wish to clarify in advance that some of the information contained in this paper, especially our statements regarding the history of Vulcan and the various code names used internal to Motorola,
cannot be independently verified and must therefore be considered mere conjecture. Nonetheless, every claim in this paper is made in good faith, in that we believe the information to be as accurate as possible, and we are not trying to deliberately deceive or mislead anyone. We certainly welcome any corrections or additions to the information presented in this paper.

We are absolutely confident in our technical reporting of Vulcan because that information was determined solely via reverse-engineering. Furthermore, we verified the accuracy of our analysis by comparing simulation data to that obtained from genuine DVP hardware.

### 1.2 Terminology

Unless specifically stated to the contrary, all mathematical operations described in this paper are on the Galois Field $G F(2)$. Accordingly, multiplication corresponds to a logical AND function and addition corresponds to a logical XOR function. Additionally, we adopt digital filter terminology for shift register delay elements, using the notation $z^{-1}$ to indicate a delay of one bit.

In our figures, we use a square with the text $z^{-1}$ inside it to denote a 1 bit delay element that is equivalent to a D flip-flop. We use a circle with a multiplication sign in to denote multiplication over $G F(2)$, which is equivalent to a logical AND gate. We use a circle with a plus sign in it to denote addition over $G F(2)$, which is equivalent to a logical XOR gate.

As much as possible, we try to use cryptographic vocabulary appropriate for stream ciphers in this paper. As such, we prefer that the term key refer to the keystream generated by the stream cipher instead of the user-selected secret key, which we prefer to identify with the term cryptovariable (CV).

Thus when we refer to a bit of key, we are referring to a keystream bit that is added modulo- 2 to plaintext to produce ciphertext during an encrypt operation, and added modulo- 2 to ciphertext to produce plaintext during a decrypt operation. Likewise when we refer to a bit of cryptovariable, we are referring to a bit of the (preferably randomly selected) secret that must be shared amongst all users of the cryptosystem in order for secure communication to be possible.

Additionally, we prefer the term ciphertext autokey (or the equivalent acronym CTAK) instead of cipher feedback (CFB) for describing the mode in which the Vulcan stream cipher operates. Ciphertext autokey implies that the keystream is generated automatically based solely upon prior ciphertext and the cryptovariable. This provides a self-synchronizing property that is advantageous for channels that do not easily provide for cryptographic synchronization [1].

As we previously mentioned, Vulcan is the internal code name by which the cipher presented in this paper is known to Motorola. Motorola marketed products based on the Vulcan cipher as DVP. We use the terms Vulcan and DVP more or less interchangeably in this paper, but observe the convention that Vulcan can only refer to the cipher itself, whereas DVP can refer to either the cipher or a product that incorporates it.

As a final comment concerning Motorola trade names, we briefly mention the trademark Securenet [15]. Securenet refers to a broad family of secure voice products that are all based on similar technology. Vulcan/DVP was the first member of the Securenet family, followed by DES, DES-XL, DVP-XL, DVI-XL, and others. A DVP encryption module can be referred to as Securenet, but Securenet does not necessarily imply DVP encryption [6].

### 1.3 Background

Based on the best information available, we believe that Motorola first began product development of digital encryption for two-way radios in 1973, although they had developed some of the necessary building blocks somewhat earlier than this. For example, US Patent \#3639690, filed in 1969, describes a linear cipher algorithm allegedly known to Motorola as Zeus [12]. Zeus was the immediate predecessor of Vulcan and both ciphers have much in common.

Zeus, shown in Figure 1, is a trivial stream cipher that generates a key bit from a linear sum modulo-2 of some combination (selected by the 26 -bit cryptovariable) of the previous 26 ciphertext bits. Since Zeus is catastrophically weak, we will not discuss it further in this paper, aside from mentioning that Vulcan was allegedly the successor to Zeus. We are uncertain as to whether Zeus was purely experimental or whether it was ever sold to an end user in an actual product.


Figure 1: Zeus Cipher
Rumors suggest that Vulcan was originally implemented using discrete logic ICs, but we have not been able to find any evidence of this. Such an implementation during the mid-1970s would have been physically quite large and rather power hungry, although it is nearly certain that Vulcan would have been prototyped using discrete logic prior to designing a custom integrated circuit.

Further rumors suggest that the earliest customers for DVP included the then newly-formed US Drug Enforcement Agency, as well as the Organization of Petroleum Exporting Countries. Salt Lake City was also an early customer of DVP [5].

Products marketed under the DVP trade name use the Vulcan cipher to encrypt digitized voice. The technique Motorola selected to digitize voice is continuously variable slope delta modulation (CVSD) at 12Kbps. Motorola developed a custom CMOS integrated circuit (code named Butterscotch) to perform CVSD encoding and decoding. This same IC was also used in secure
voice products that succeeded DVP, such as Motorola's DES encryption device, introduced around 1980.

In a DVP-equipped radio, at the transmitter, microphone audio is converted to 12 Kbps CVSD plaintext and is then subsequently encrypted using the Vulcan cipher, which operates in a self-synchronizing mode known as ciphertext autokey (CTAK). The resulting ciphertext is then modulated onto the RF carrier as 2level FM and transmitted to the receiving radios.

In a DVP-equipped radio, at the receiver the 2-level FM signal is demodulated and the digital ciphertext is recovered. These functions are performed by an IC (believed to be a MC6800 family microprocessor) known as the control and interface IC, also referred to using the code name Vanilla. The recovered ciphertext is decrypted by the Vulcan cipher and the resulting plaintext is converted back to analog audio by the CVSD decoder IC (Butterscotch).

We believe it likely that Motorola's development of 12Kbps CVSD Securenet was inspired by NSA's VINSON (KY-57/KY-58) series of 16Kbps CVSD tactical voice security equipment, which was developed in the early 1970s, immediately prior to DVP [9].

The first radios to be offered with DVP were Motorola's Micor mobile and MX300 portable, both of which entered the market in 1975 [5]. In 1980 Motorola added the DES cipher to its Securenet encryption products as a more secure alternative to DVP. During the mid-1980s, Motorola replaced DVP with DVPXL and DVI-XL, both incorporating new proprietary cipher algorithms that remain as yet unpublished.

As a side note, the XL term when used regarding Securenet indicates a range extension technology best described in US Patent \#4893339 [14]. The XL method, known internally to Motorola as REX, a diminutive of "range extension", is a clever technique of converting a block cipher operating in 1-bit cipher feedback (CFB) mode into a block cipher operating in a sort of counter addressing mode. The essential idea is to eliminate the error propagation characteristics of CFB mode. Note that DVP-XL is not REX applied to DVP (Vulcan), but is instead REX applied to Linus, the proprietary cipher that replaced Vulcan.

### 1.4 Motivation

As to why we went to the effort to reverse-engineer and cryptanalyze Vulcan, we have several motives. First, we wish to preserve a historical cipher that would otherwise have been lost to the black hole that is corporate history. Nearly all proprietary ciphers have suffered this fate, and all seem destined to it unless some enterprising third party intervenes. Our hope is that future cryptographers will find this paper an interesting and useful historical reference.

Second, we seek to reinforce the strong argument that one must never trust ciphers that have not been exposed to open academic scrutiny. As our cryptanalysis will show, Vulcan is a catastrophically weak cipher, even when measured by 1970s standards. Vulcan would never have withstood public inspection and DVP would have been far less successful in the marketplace had its customers
known how vulnerable this cipher actually was. We strongly suspect that many other proprietary ciphers are similarly weak and we warn against their use.

Third, we wish to assist budding cryptographers and reverse-engineers in learning the basic skills. Vulcan, being both simple and weak, offers an excellent learning opportunity for a beginning cryptographer. We believe that breaking a Vulcan-encrypted message would be a suitable homework assignment for students enrolled in an undergraduate cryptography course. Other suitable assignments might include investigating ways of making simple changes to Vulcan in order to improve its cryptographic strength.

Fourth, we undertook this effort for sheer enjoyment. This project required us to learn many new skills and overcome many technical difficulties. Nothing beats the satisfaction gained from solving a giant puzzle. Nothing more thoroughly whets the appetite for knowledge than a secret.

We imagine that Motorola would prefer that this paper did not exist, and we would not be surprised if they seek to remove it from the public eye, presumably under the feeble justification that our work threatens the security of their customers. No doubt the real reason they might object to our work is to avoid embarrassment resulting from the disclosure that DVP is far from secure.

In spite of these imagined protests, we are completely confident that no harm will come to any end users for the simple reason that Vulcan and DVP are long obsolete and have not been used to protect sensitive communications in decades. For these same reasons, we are equally confident that no harm will be done to any active NSA signal intelligence efforts. Furthermore, we are confident that we have broken no laws in the course of our work, and we believe that this paper is both legally and technically legitimate.

We want to make it perfectly clear that we do not wish to disparage Motorola with our comment that Vulcan is weak; quite the contrary, in its era DVP was revolutionary. We commend Motorola for developing the first digitally encrypted two-way radio available to non-military customers at a time when simple analog frequency inversion scrambling was considered "high tech".

Furthermore, we wish to allay any concerns that our revelations will in any way harm Motorola's intellectual property. Had this paper been published in 1976, such a concern would have been legitimate, but in 2014 the technology underlying Vulcan and DVP is not only long obsolete, but absolutely archaic. None of Motorola's competitors have anything to gain, or even learn, from our disclosures.

### 1.5 Technique

We were surprised at how easy it was for us to learn the details of Vulcan. Although the Vulcan cipher is implemented in a custom CMOS integrated circuit, techniques for reverse-engineering ICs are well known. We do not know why the myth still persists that reverse-engineering hardware is more difficult than reverse-engineering software. We caution that secrets cannot be hidden in silicon or software; only a fool believes otherwise.

The first step in our process was to acquire several DVP modules and a DVP key loader, both of which are inexpensive and readily available on eBay and at ham fests. We then examined the DVP modules to identify the purpose of the various ICs these modules contain. Our attention quickly focused on an IC labeled only with a Motorola logo and bearing the identifier SC76807 (see Figure 2).


Figure 2: SC76807 Integrated Circuit
Unlike plastic IC packages, which require strong acids to unveil the die, the SC76807 uses a ceramic and metal chip carrier package that can be de-capped with a large soldering iron. Once we removed the metal lid, the die was exposed for our inspection (see Figure 3).

We imaged the die at a magnification of 200 using a metallurgical microscope and a digital camera, resulting in a clear view of the top metal layer of the die. We quickly determined that the SC76807 is a medium scale integration (MSI) metal gate CMOS chip of large geometry, typical of mid to late 1970s IC process technology. Although we did not count the individual gates, this is a fairly simple device, well within the ability of a single individual to analyze.

Once we had an image of the M1 layer of the SC76807, we began analyzing the various circuits and their functions. Due to the large amount of repeated circuitry, this task was not nearly as time-consuming as we had originally expected. Our detailed analysis of the SC76807 IC follows in Section 2.

Once we fully understood the workings of the SC76807, we realized that this knowledge by itself was insufficient for a complete understanding of DVP. This is because the DVP key loader plays a significant role in the overall operation of DVP. We therefore had to obtain a DVP key loader and perform some reverseengineering on it as well.

Like the SC76807, the DVP key loader was easy to reverse-engineer, although in this case it was primarily software rather than hardware that we had to reverse-engineer. After reading the firmware ROM and disassembling the code, we quickly discovered the portions of this software that determine how the 71-

bit user-entered cryptovariable is converted into the 138-bit cryptovariable that the SC76807 requires. Our full analysis of the DVP key loader follows in a Section 4.

Once we understood the requisite technical details of both the SC76807 IC and the DVP key loader, we then created some software simulations that implement the Vulcan cipher. We used GNU Octave (an open-source alternative to Matlab) for our simulations and analysis, but our results can be replicated easily in other languages such as $\mathrm{C}++$ or Python. All of our Octave source code is presented in the appendix.

To test the validity of our simulations, we used the DVP key loader itself to create test vectors where the cryptovariable, plaintext, and ciphertext were all known. This allowed us to confirm that our software simulations were correct, and also provided verification that our analysis of both the SC76807 IC and the DVP key loader is accurate.

After we had proven the validity of our software simulations, we then set about analyzing Vulcan from a mathematical perspective. Although we are not mathematicians per se, cryptanalysis of Vulcan requires little more than an undergraduate understanding of linear algebra. We note later in this paper that more elegant and efficient cryptanalytic attacks against Vulcan are almost certainly possible, and thus a topic ripe for further research. Nonetheless, our simple-minded attacks proved all too effective.

## 2 SC76807 CMOS IC

In this section, we present the findings of our hardware reverse-engineering effort. Figure 3 illustrates the top metal (M1) layer of the SC76807 Vulcan IC. This chip is a metal-gate CMOS IC of large ( $>1$ micron) geometry. To avoid cluttering Figure 3 with labels, we will instead describe the various functions of the circuit elements in the paragraphs below.

### 2.1 Technique

Understanding the SC76807 IC was a new and interesting challenge for us since we had not previously reverse-engineered silicon integrated circuits. Fortunately for us, the large geometry and low gate count of 1970s-era medium scale integration made it relatively easy to proceed.

Understanding how bits are stored in volatile memory on an IC is a critical first step to reverse-engineering the SC76807. Many excellent tutorials can be found on the web, so we will not duplicate that effort here [10]. The key observation is that a single bit is stored in a pair of inverters that are permanently coupled together. Such pairs of coupled inverters are plentiful in the SC76807.

We assumed the SC76807 would consist largely of shift registers and static random access memory, something the visual presence of a large number of coupled inverters confirmed. We present a detailed analysis of the specific circuitry
of the SC76807 in a later section, but for now we will briefly mention what we discovered at our first glance.

The large regular structure in the center of the chip is clearly an SRAM array. This is obvious due to the address decoding logic adjacent to the array, along with the pre-charge circuitry and the word and bit lines that connect to each cell. We quickly counted 128 individual bit cells.

The numerous ladder-like structures are clearly shift registers, as evidenced by numerous pairs of coupled inverters separated by clocked gates. The repetitive nature of these circuits makes them easy to analyze.

We strongly suspected that the single ladder-like structure that differed from all the rest was likely a tree of XOR gates, an assumption that later proved mostly correct. Other smaller circuits required more careful analysis to fully understand. Examples of these include the various signal switching circuits and the differential encoder and decoder.

Although we had signal traces of all I/O pins that we captured with a logic analyzer, we did not have access to a schematic that provided names for these pins. We had to figure out the functions of the I/O pins by examining both the logic analyzer traces and the associated circuitry of the SC76807.

### 2.2 A Guided Tour

Before we discuss specific circuits, we must first establish pin numbers and the "correct" orientation of the chip. Figure 3 (when viewed with the caption at the bottom) displays the IC in our preferred orientation, with Pin 1 clearly visible in the upper right hand corner. Pin 1 is easily identified by the distinctly rounded corners of its bonding pad.

We choose to numbers the pins, starting with Pin 1, in counterclockwise ascending order. We observe that there are seven pins along the top of the chip and six pins along the bottom. However, Figure 3 reveals that only six of the seven pins along the top of the die are bonded, with one unavailable external to the IC. We thus choose to omit the un-bonded pin and instead number only the twelve bonded pins from one to twelve, counterclockwise from the upper right hand corner.

Using this scheme, in Figure 3 we have: Pin 1 in the upper right hand corner, Pin 6 in the upper left hand corner, Pin 7 in the lower left hand corner, and Pin 12 in the lower right hand corner. With this numbering scheme in place, we now proceed to discuss the name and function of each pin.

### 2.2.1 Inputs and Outputs

Pin 1 is $\overline{\mathrm{C} 1} / \mathrm{C} 2$ (code select) input. It controls the cryptographic transformation and acts as an extra bit of cryptovariable. This signal connects to the first XOR gate in the XOR tree and inverts the sense of the AND function that is used to enable or disable this XOR gate (this feature will be explained in a later section). This pin is normally low (selecting C1).

Pin 2 is cryptovariable input. It conveys the cryptovariable from the key loader to the IC during key loading operations. It is connected directly to the CV shift register seen at the far right of the chip.

Pin 3 is plaintext input. It connects to some switching logic near the top center of the die. The switching logic routes plaintext and ciphertext to the appropriate circuits depending on whether the chip is operating in encrypt (transmit) or decrypt (receive) mode.

Pin 4 is $\overline{\mathrm{TX}} / R X$ input. It selects whether the chip encrypts (TX) or decrypts (RX). This signal connects to the same switching logic circuitry as Pin 3.

Pin 5 is ciphertext input. It connects to a differential decoder and then to the switching circuitry that routes it appropriately based on whether the chip is encrypting or decrypting.

The un-bonded pin between Pin 5 and Pin 6 is a test output that is inaccessible external to the chip. This test pin allows the contents of the 128-bit SRAM (discussed later) to be read, presumably to enable testing of the SRAM during manufacturing. We have deliberately chosen not to give this pin a number since it has no function during normal operation.

Pin 6 is $\mathrm{V}+$, the positive supply voltage. It supplies Vdd to all the transistors on the chip.

Pin 7 is ciphertext output. It is differentially encoded and is valid only during encrypt (transmit) operations.

Pin 8 is clock input. It allows an externally sourced clock signal to act as a time base for all synchronous logic on the chip. External inputs to the chip are clocked in on the rising edge of this clock. The nominal clock rate should be 12 KHz during encrypt and decrypt operations, but we observe that the DVP key loader supplies a faster clock than this during key loading.

Pin 9 is $\overline{\mathrm{WE}}$ (write enable) input. When active (low), it allows the CV shift register to operate, clocking in CV bits from Pin 2 in accordance with the clock signal on Pin 8. Circuitry on the chip creates a single clock cycle delay between the $\overline{\mathrm{WE}}$ input and the clock and CV inputs, an observation that is important when analyzing the exact temporal relationship of these three signals.

Pin 10 is output-enable input. This signal must be high in order for ciphertext out (Pin 7) and plaintext out (Pin 11) to produce outputs. If Pin 10 is low, ciphertext out and plaintext out will be in high impedance states.

Pin 11 is plaintext output. It connects to the bottom of the XOR tree and is only valid during decrypt operations.

Pint 12 is ground. It provides Vss to all the transistors on the chip.

### 2.2.2 Circuitry

The large repetitive structure directly in the center of the chip is a static random access memory (SRAM) array of 128 bits, arranged as eight rows by sixteen columns. Address decoding circuitry is present immediately to the left of and immediately below the SRAM array. The SRAM address is determined by bits held in the shift register along the left side of the SRAM array.

At the far right side of the chip is a vertical column of circuitry, the CV shift register. It receives input from CV in (Pin 2) and is controlled by the clock (Pin 8) and $\overline{\mathrm{WE}}(\operatorname{Pin} 9)$. When $\overline{\mathrm{WE}}$ is high (inactive), the CV shift register does not shift. When $\overline{\mathrm{WE}}$ is low, CV input bits from Pin 2 are clocked into the CV shift register on the rising edge of the clock.

The CV shift register has a capacity of 10 bits and also supplies an output to the SRAM array. When $\overline{\mathrm{WE}}$ is low, CV bits are clocked through the CV shift register and written into the 128-bit SRAM array in accordance with the address contained in the shift register to the left of the SRAM array.

At the far left side of the chip are two vertical columns of circuitry. Together these form a 21-bit ciphertext shift register, with 14 bits in the left column and 7 bits in the right column. The ciphertext shift register input comes from a switching circuit located directly above the SRAM array.

The ciphertext shift register input is at the top of the right column and the ciphertext shift register output is at the top of the left column. Bits in the right column move down the circuitry and then across to the left column where they then move up the circuitry. The output of the ciphertext shift register connects to an XOR gate located above the SRAM array.

The circuit immediately above the right column of the ciphertext shift register supplies bits of the cryptovariable to the SRAM array during key loading. The circuit below the right column of the ciphertext shift register is a differential encoder that encodes the output of the ciphertext shift register and supplies it to the ciphertext output (Pin 7).

The newest seven bits in the ciphertext shift register form an address to the SRAM row and column address decoding logic. Each cell of the first seven bits in the ciphertext shift register supplies both inverted and non-inverted bits for use in addressing the SRAM array.

The horizontal row of circuitry above the SRAM array performs a variety of functions. The leftmost circuit in this row is a read sense amplifier for the SRAM array, followed by an XOR gate that adds modulo- 2 the output of the ciphertext shift register to the bit of cryptovariable that is addressed by the first seven bits of the ciphertext shift register. We will call the result of this operation a modified ciphertext bit.

Modified ciphertext bits enter a 10-bit shift register that is the vertical column of circuitry immediately to the right of the SRAM array. Modified ciphertext bits enter this shift register at the top and move down the circuitry. The modified ciphertext bits in this shift register are also transferred in parallel to the XOR tree, which is the column of circuitry immediately to the right of this shift register.

Returning now to the circuitry above the SRAM array, immediately to the right of the circuit that forms the modified ciphertext bit, we have a differential decoder that decodes the ciphertext input signal of Pin 5 and supplies it to a switching circuit to the right of this differential decoder.

Two such switching circuits are located at the right side of the row of circuitry directly above the SRAM array. The leftmost of these switching circuits selects the ciphertext to be transferred to the ciphertext shift register. This
ciphertext comes from the output of the XOR tree during encrypt (TX) and comes from the differential decoder during decrypt (RX).

The rightmost of these switching circuits selects the input of the XOR tree. The XOR tree input comes from plaintext in (Pin 3) during encrypt (TX) and comes from the output of the differential decoder during decrypt (RX).

The XOR tree, visible as a vertical column of circuitry in between the modified ciphertext shift register and the CV shift register, has 10 stages, each of which accepts an input from the previous stage, conditionally modifies this input, and then provides an output to the next stage.

The modification performed by each stage of the XOR tree is simply a conditional exclusive-or (addition modulo-2), conditioned on the value of the corresponding bit of CV from the CV shift register. The two bits being added modulo-2 are the input to this stage of the XOR tree and a corresponding bit of modified ciphertext from the modified ciphertext shift register.

Certain stages of the XOR tree behave differently from the aforementioned general rule. Specifically, the first stage of the XOR tree is further modified by the value of $\overline{\mathrm{C} 1} / \mathrm{C} 2(\operatorname{Pin} 1)$. The last stage of the XOR tree never performs an XOR (i.e. acts as though the corresponding CV bit is always 0 ). The second-to-last stage of the XOR tree always performs an XOR (i.e. acts as though the corresponding CV bit is always 1). All other stages perform the XOR only if the corresponding CV bit is 1 . If the corresponding CV bit is 0 , the output of this stage is the same as its input.

Immediately below the modified ciphertext shift register is a circuit that delays the $\overline{\mathrm{WE}}$ signal by one clock cycle and then combines this signal with the master clock to create a clock signal for the CV shift register. The overall effect is that the CV shift register only clocks when $\overline{\mathrm{WE}}$ is 0 , and the effect of transitions on $\overline{\mathrm{WE}}$ is delayed by one clock cycle relative to all other signals.

The circuitry between Pin 7 and Pin 8 is a pair of test transistors. The thin column of circuitry at the far right edge of the SRAM array is pre-charge circuitry for the SRAM. All I/O input pins have diode protection and series resistance to guard against damage. This concludes our description of the SC76807 circuitry. We turn now to the overall operation of the device.

### 2.3 Modes of Operation

The SC76807 IC can operate in at least three different modes: encrypt, decrypt, and key loading. Each of these modes affects how input and output signals are processed by the IC. Although the SC76807 IC is the device that performs the cipher operations, the overall DVP module itself consists of additional circuitry that we have not described here because it is not particularly interesting or relevant.

During encrypt, the DVP module delivers plaintext CVSD bits to Pin 3 of the SC76807 and receives ciphertext bits from Pin 9 of the SC76807. The $\overline{\mathrm{TX}} /$ RX input on Pin 4 must be low. Each bit-wise encrypt operation requires the clock signal on Pin 8 to transition from low to high once the plaintext input is stable. The ciphertext output bit is retrieved on the falling edge of the clock
signal. The $\overline{\mathrm{WE}}$ input on Pin 9 and the output enable input on Pin 10 must both be high during encrypt operation.

During decrypt, the DVP module delivers received ciphertext bits to Pin 5 of the SC76807 and receives plaintext CVSD bits from Pin 11 of the SC76807. The $\overline{T X} / R X$ input on Pin 4 must be high. Each bit-wise decrypt operation requires the clock signal on Pin 8 to transition from low to high once the ciphertext input is stable. The plaintext output bit is retrieved on the falling edge of the clock signal. The $\overline{\mathrm{WE}}$ input on Pin 9 and the output enable input on Pin 10 must both be high during decrypt operation.

During key loading, the DVP module, in conjunction with an external key loader, must supply cryptovariable bits on Pin 2, ciphertext input bits (used for SRAM addressing) on Pin 5, and an appropriate clock signal on Pin 8. The $\overline{\mathrm{WE}}$ input on Pin 9 and the $\overline{\mathrm{TX}} / \mathrm{RX}$ input on Pin 4 must both be low.

The DVP key loader only delivers cryptovariable and ciphertext input signals. All other necessary inputs (including the clock) are supplied to the SC76807 IC by other circuitry on the DVP module itself.

The DVP key loader specially constructs the ciphertext stream to ensure that the appropriate cryptovariable bits are written into the appropriate SRAM addresses. The first 128 bits of the cryptovariable are written into the SRAM array and the last 10 bits of the cryptovariable remain in the CV shift register.

## 3 Vulcan Cipher Algorithm

Vulcan is a self-synchronizing stream cipher with a 138-bit cryptovariable. In this section we describe the cipher in complete detail.

Vulcan uses a ciphertext delay shift register to maintain a history of the most recent 31 bits of ciphertext for use in ciphering operations. The first seven of these 31 ciphertext bits form a 7 -bit address into the 128-bit SRAM array.

The SRAM array holds 128 bits of cryptovariable, and each time a message bit is encrypted or decrypted, the bit of cryptovariable addressed by the first seven bits in the ciphertext shift register is read from the SRAM array and is then added modulo-2 to the 21st bit in the ciphertext shift register. Therefore, the last 10 bits in the ciphertext shift register have been modified by certain bits of the cryptovariable.

The primary ciphering mechanism of Vulcan is a tree of interconnected XOR gates that add modulo-2 certain ciphertext bits and cryptovariable bits to the input bit, where the input bit is a plaintext bit during encrypt, and is a ciphertext bit during decrypt.

Figure 4 illustrates a Vulcan encrypt operation. Plaintext bits serially enter at the lower left in accordance with a clock signal (not shown). Differentially encoded ciphertext bits serially exit at the lower right. The differential encoder protects against arbitrary phase inversions on the communications channel and is of no cryptographic importance. A non-differentially-encoded version of the ciphertext output bit feeds back to the ciphertext shift register near the upper left.


Figure 4: Vulcan Cipher Encrypt

The most recent seven ciphertext bits in the ciphertext shift register form a 7bit address to the SRAM, addressing one bit of the first 128 bits of cryptovariable $\left(k_{000} \ldots k_{127}\right)$. The specifically addressed bit of cryptovariable transfers to an XOR gate at the far right of the ciphertext shift register, where it adds modulo2 to the 21st bit of ciphertext. The bit resulting from this XOR then enters the modified ciphertext shift register, which holds 10 bits (only 9 of which are shown).

The bits in the modified ciphertext shift register are added modulo- 2 to the plaintext bit, under conditional control of the last 8 cryptovariable bits (with the exception of the last bit in the modified ciphertext shift register). In effect, the 8 bits of cryptovariable $\left(k_{130} \ldots k_{137}\right)$ act as switches that either enable or disable the corresponding XOR gate.

If the given cryptovariable bit is 1 , then the corresponding XOR gate adds the modified ciphertext bit to the partial ciphering result. If the given cryptovariable bit is 0 , then the XOR gate is disabled and the partial ciphering result simply passes on to the next XOR gate unmodified.

Although the $G F(2)$ multiplication (equivalent to a logical AND) shown here is nonlinear, we can think of the overall effect of this multiply as simply enabling or disabling the $G F(2)$ addition (logical XOR) for this stage of the operation. This perspective becomes important when we attempt cryptanalysis of Vulcan in a later section.

A quick review of the XOR tree and its associated circuitry on the SC76807 IC reveals that the XOR tree has a total of 10 stages, whereas we show only 9 stages in Figure 4. This is because the logic of the last two stages is different from that of the first eight stages.

The last stage of the XOR tree, which would have been controlled by cryptovariable bit $k_{128}$, simply passes its input to its output without modification. This is why we omitted it from Figure 4.

The second to last stage of the XOR tree always adds the modified ciphertext bit to the partial ciphering result, which is why this stage lacks the AND gate that would have been controlled by cryptovariable bit $k_{129}$.

As a consequence of how the last two stages of the XOR tree operate, cryptovariable bits $k_{128}$ and $k_{129}$ have no effect whatsoever. Not surprisingly, these
bits and are always set to 0 by the DVP key loader. Therefore, even though the loaded cryptovariable is in fact 138 bits long, only 136 bits influence ciphering operations.

Interestingly, although Vulcan uses a 138-bit cryptovariable (of which only 136 bits have any effect), the DVP key loader only allows the end user to enter a 71-bit cryptovariable. Of the remaining 67 bits, two bits are always set to 0 , and 65 bits are formed from linear combinations of the user-supplied 71 bits. We will discuss this fact in much greater detail in later sections of this paper.

Figure 5 illustrates a Vulcan decrypt operation, which is nearly identical to the Vulcan encrypt operation shown in Figure 4. Differentially encoded ciphertext bits serially enter at the lower left in accordance with a clock signal (not shown). These bits are then differentially decoded to remove any dependence on phase characteristics of the communications channel. Differential decoding is of no cryptographic significance, but is shown here for completeness.


Figure 5: Vulcan Cipher Decrypt
The differentially decoded ciphertext bit then enters both the XOR tree logic at the bottom of Figure 5, and the ciphertext shift register at the top of Figure 5. Once the ciphering operation is complete, plaintext bits serially exit at the lower right. All other decrypt operation is identical to the encrypt operation previously described.

### 3.1 Mathematical Description

For all stream ciphers, encrypt and decrypt operations are essentially identical except for the inputs and outputs [1]. To encrypt, add the keystream to the plaintext input to obtain the ciphertext output. To decrypt, add the keystream to the ciphertext input to obtain the plaintext output. In both cases the cipher is fully described by the keystream it generates.

Vulcan generates its keystream as a linear combination of prior ciphertext bits and cryptovariable bits. In the description below, we consider the theoretical case where all 138 bits of the Vulcan cryptovariable can be freely specified by the end user. In actuality, DVP only allows the end user to specify 71 of these 138 cryptovariable bits, with the remaining 67 bits being a linear combination of the 71 .

We have designated the 138 cryptovariable bits as $k_{000}$ through $k_{137}$ in the order that they are clocked into the SC76807 IC during key loading. We note that the first 128 bits of the cryptovariable are clocked into various locations in the SRAM array, whereas the last 10 bits of the cryptovariable are clocked into the CV shift register.

For our purposes here, we do not care about the actual addressing of the SRAM so long as we remain consistent in our approach to how this addressing works. This will not be the case once we consider the specifics of the actual user-entered 71-bit cryptovariable, but for now it simplifies our mathematical description of Vulcan.

Let each SRAM storage cell contain the cryptovariable bit identified by the number that is the SRAM address (i.e. SRAM location 000 contains $k_{000}$, SRAM location 001 contains $k_{001}$, etc.) We therefore have $k_{000}$ through $k_{127}$ in SRAM locations 000 through 127. Furthermore, let the CV shift register contain cryptovariable bits $k_{128}$ through $k_{137}$ in locations 10 (the bottom) through 1 (the top) respectively.

Using the notation that $c(n-x)$ designates a ciphertext bit from $x$ clock cycles ago, and using ordinary (as opposed to $G F(2)$ ) arithmetic for the SRAM address computation, we can express the present keystream bit, $K(n)$, as:

$$
\begin{align*}
& K(n)=\left(k_{137} \operatorname{AND}\left(c(n-22) \operatorname{XOR} k_{(64 c(n-8)+32 c(n-7)+16 c(n-6)+8 c(n-5)+4 c(n-4)+2 c(n-3)+c(n-2))}\right)\right) \\
& \text { XOR }\left(k_{136} \operatorname{AND}\left(c(n-23) \operatorname{XOR} k_{(64 c(n-9)+32 c(n-8)+16 c(n-7)+8 c(n-6)+4 c(n-5)+2 c(n-4)+c(n-3))}\right)\right) \\
& \operatorname{XOR}\left(k_{135} \operatorname{AND}\left(c(n-24) \operatorname{XOR} k_{(64 c(n-10)+32 c(n-9)+16 c(n-8)+8 c(n-7)+4 c(n-6)+2 c(n-5)+c(n-4))}\right)\right) \\
& \text { XOR }\left(k_{134} \text { AND }\left(c(n-25) \text { XOR } k_{(64 c(n-11)+32 c(n-10)+16 c(n-9)+8 c(n-8)+4 c(n-7)+2 c(n-6)+c(n-5))}\right)\right) \\
& \text { XOR }\left(k_{133} \operatorname{AND}\left(c(n-26) \text { XOR } k_{(64 c(n-12)+32 c(n-11)+16 c(n-10)+8 c(n-9)+4 c(n-8)+2 c(n-7)+c(n-6))}\right)\right) \\
& \operatorname{XOR}\left(k_{132} \operatorname{AND}\left(c(n-27) \operatorname{XOR} k_{(64 c(n-13)+32 c(n-12)+16 c(n-11)+8 c(n-10)+4 c(n-9)+2 c(n-8)+c(n-7))}\right)\right) \\
& \text { XOR ( } \left.\left.k_{131} \text { AND ( } c(n-28) \text { XOR } k_{(64 c(n-14)+32 c(n-13)+16 c(n-12)+8 c(n-11)+4 c(n-10)+2 c(n-9)+c(n-8))}\right)\right) \\
& \text { XOR }\left(k_{130} \text { AND }\left(c(n-29) \text { XOR } k_{(64 c(n-15)+32 c(n-14)+16 c(n-13)+8 c(n-12)+4 c(n-11)+2 c(n-10)+c(n-9))}\right)\right) \\
& \text { XOR }\left(c(n-30) \text { XOR } k_{(64 c(n-16)+32 c(n-15)+16 c(n-14)+8 c(n-13)+4 c(n-12)+2 c(n-11)+c(n-10))}\right) \text {. } \tag{1}
\end{align*}
$$

This equation reveals that the Vulcan keystream depends only on the value of certain prior ciphertext bits as well as certain cryptovariable bits. With the exception of the $G F(2)$ multiplication operations (the logical ANDs), the rest of the equation is entirely linear (on $G F(2)$ ). This fact is of paramount importance in cryptanalysis.

Given that DVP does not allow the end user to specify all 138 bits of the Vulcan cryptovariable, we now turn to the issue of how the DVP key loader transforms the 71-bit user-entered cryptovariable into the 138-bit version required by the SC76807 IC.

## 4 DVP Key Loader

Based on our analysis of the SC76807 IC and the Vulcan cipher it implements, we soon realized that our understanding of DVP was incomplete. We have just described the Vulcan cipher itself, but we have not yet described how the 71bit cryptovariable entered by the end user into the DVP key loader affects the cipher. That is the topic of this section.

Although there are several models of DVP key loaders, we acquired the Motorola T3010AX key variable loader (KVL) shown in Figure 6. The T3010AX is apparently the first revision of a second generation of DVP key loaders, the original being the relatively rare (and collectible) P1001 series of code inserters [8].


Figure 6: T3010AX DVP Key Loader

### 4.1 Description and Operation

Our T3010AX KVL has a TRN6777B cryptographic hybrid in it. This hybrid contains the SC76807 Vulcan IC and some additional non-cryptographic circuitry that is of no interest here. It is the presence of the TRN6777B crypto hybrid within the KVL that allows us to gather a known triplet of cryptovariable, plaintext and ciphertext that we can use to verify the correctness of our software simulations of the Vulcan cipher.

The KVL is a relatively simple device that allows an end user to manually enter (via a membrane keypad) a cryptovariable into the key loader and then transfer this cryptovariable into a DVP-equipped radio. Unlike more modern key loaders (e.g. the KVL4000), the T3010 cannot generate a random cryptovariable (this fact alone is a rather serious security flaw). Instead, the user must manually enter 24 octal digits, with the final digit being restricted to values 0 to 3 . Key entry proceeds in four groups of six octal digits each, with the intermediate result being displayed on a LED numeric display. The T3010 thus provides the user the ability to enter a 71-bit cryptovariable even though the Vulcan cipher itself requires a 138 -bit cryptovariable.

Internally, the KVL is based on a Motorola 6802 microprocessor that performs all the required functions except for encryption. Encryption is handled exclusively by the embedded TRN6777B hybrid. The 6802 processor primarily performs user interface functions such as handling the membrane keypad inputs and numeric LED display outputs.

Interestingly, when transferring a cryptovariable from the KVL to a target radio (a process commonly referred to as key loading), communication is strictly one-way: from the KVL to the target. No provision exists for the target to communicate to the KVL. This means that the KVL cannot be certain that a key load operation was successful. Later generations of Securenet key loaders implemented bidirectional communications between the KVL and the target device.

To overcome the problem of not knowing if a key loading process was successful or not, the KVL sends a short burst of encrypted CVSD immediately following the cryptovariable transfer. If the CV was successfully transferred, the target radio will correctly decrypt this ciphertext and play the resulting plaintext (a tone) on the radio speaker. Therefore, if the user hears a beep from the radio immediately after key loading, that means the CV was successfully loaded. If no beep occurs, the key load process failed and only static will be heard.

If it were not for this unusual form of verification, the KVL would not require an internal crypto hybrid. The only reason the T3010 KVL has an internal TRN6777B hybrid is so that it can generate the correct ciphertext to send to the target radio to make it beep after a successful key load. We exploited this operation by capturing data with a logic analyzer and then used that data to confirm the validity of our software simulations of Vulcan.

### 4.2 Reverse Engineering

Understanding the DVP key loader required us to reverse-engineer the 6802 microprocessor firmware. To do this, we extracted the contents of the ROM that contains the 6802 firmware and then disassembled it. The firmware disassembly, along with a few of our comments, is listed in Appendix E.

We quickly located several lookup tables and soon identified one as containing messages that are displayed on the LED. From our experience using the DVP key loader, we knew that it displayed the message "beep?" when key loading was complete. We therefore determined the address of this message in the lookup table and then searched the code for a reference to this address.

This led us to the section of firmware that contained the algorithm for converting the 71-bit user supplied cryptovariable into the 138-bit cryptovariable that the Vulcan IC requires. We call this conversion algorithm the key schedule and describe it fully below. Interested readers can refer to the firmware disassembly in Appendix E, but we caution that the code there is rather convoluted and far more difficult to understand than the summary description we provide here.

### 4.3 Key Schedule

As we have previously noted, DVP allows the user to specify a 71-bit cryptovariable, but the Vulcan cipher itself requires a 138-bit cryptovariable. The DVP key loader converts the 71 user-entered bits into a 138-bit cryptovariable and then transfers this to the Vulcan IC. This conversion process is loosely analogous to the key schedule of many common ciphers.

For convenience, we prefer the following terminology. Suppose that the end user enters a 24-digit octal cryptovariable into the KVL. We will designate these 24 octal digits as $\kappa_{23}$, the most significant digit and the first digit entered, through $\kappa_{00}$, the least significant digit and the last digit entered (which is furthermore restricted to the octal values 0 through 3 ).

Converting the 24-digit user-entered octal CV into 71 bits, we proceed left to right (i.e. from most significant to least significant) in a straightforward manner. We will designate the 71 bits of user-entered CV as $v_{70}$ (most significant) through $v_{00}$ (least significant). Therefore: $\kappa_{23}=v_{70} \cdot v_{69} \cdot v_{68}, \kappa_{22}=v_{67} \cdot v_{66} \cdot v_{64}, \ldots$, $\kappa_{01}=v_{04} \cdot v_{03} \cdot v_{02}$, and $\kappa_{00}=v_{01} \cdot v_{00}$ (since it has only 2 bits, having been restricted to octal values 0 through 3 ).

The DVP key loader converts this 71-bit user-entered CV, $v_{70} \ldots v_{00}$, into a 138-bit cryptovariable and then loads this into the SC76807 IC. The exact conversion and key loading process is somewhat convoluted, so we will describe it in general terms and then give the precise final result.

Loosely speaking, the key loader creates a 64 -bit data word from the 64 most significant user key bits (roughly bits $v_{70} \ldots v_{07}$ ). This process involves iterated right shifts of each octal digit with the result that the final 64-bit data word is not a straightforward copy of $v_{70}$ through $v_{07}$ (nor does it actually include $v_{07}$ ).

Once the DVP key loader has built the 64-bit data word, the DVP key loader
then computes a 64 -bit parity word from the data word. The parity word is generated using a Galois linear feedback shift register and a dense feedback polynomial [2]. Precise details as to how this is done can be found in our key schedule simulation in Appendix A.

Since the key loader started with 71 bits and used 64 of these to form the data word, seven bits remain leftover. The key loader gathers these seven leftover bits and creates an 8-bit data word from them. This data word contains user key bits $v_{00}$ through $v_{04}, v_{06}, v_{07}$, as well as a fixed 0 bit. Details regarding the precise construction of this 8-bit word can be found in our key schedule simulation in Appendix A.

If the Hamming weight of this 8-bit word is less than four, the DVP key loader complements this word so that the Hamming weight of the final result is always four or greater. This curiosity restricts the final permissible values of this 8 -bit word to 163 values, of which only 128 are possible; a fact that is very useful for cryptanalysis.

Once the DVP key loader has assembled the 64-bit data word, 64-bit parity word, and 8-bit data word, the key loader then builds a table of SRAM addresses and corresponding cryptovariable bits. Both the SRAM addresses (differentially encoded) and the cryptovariable bits are then clocked into the SC76807 IC during the key loading operation. A pair of null bits (always 0 ) fills the unused space between the bits placed in the SRAM and the bits placed in the CV shift register. Details regarding the key loading clock schedule can be found in our key schedule simulation in Appendix A.

Ultimately the key loader effectively creates a key schedule that permutes the 71 bits of user-entered cryptovariable across 128 bits that are stored in the SRAM and 8 bits that are stored in the CV shift register. Our cryptanalysis simulation in Appendix D contains a matrix that lists the SRAM contents in terms of the user-supplied CV bits $\left(v_{70} \ldots v_{00}\right)$ by address. This matrix is obviously useful for cryptanalysis.

## 5 Simulations

Our ultimate goal in reverse-engineering Vulcan was to understand it sufficiently well to allow us to create bit-exact software simulations. Accurate software simulations of Vulcan eliminate the need for the custom IC and associated hardware and also enable experimentation and cryptanalysis.

We chose to divide our software simulations into three major parts: the key schedule, encryption, and decryption. We used GNU Octave as our development environment of choice, and our simulations are all written in that language (which is also compatible with Matlab). It should be very easy for anyone with software skills to port our Octave simulations to another environment, such as Python or Java.

### 5.1 Key Schedule

We created an Octave script to simulate the portion of the DVP key loader that transforms the 71-bit user-entered cryptovariable into the 138-bit cryptovariable that gets clocked into the SC76807 IC during key loading. We provide a full listing of our script in Appendix A. Although our source code is well commented, we provide a broad overview of the operation here.

We first check the 24-octal-digit user-supplied cryptovariable for simple errors and substitute a default test CV if necessary. Next we convert the 24 octal digits into 71 bits and then extract 64 of these bits to form the 64 -bit data word. We then compute the 64 -bit parity word from the 64 -bit data word by using a Galois linear feedback shift register. We also form the 8 -bit CV shift register contents from the leftover cryptovariable bits and complement it if necessary based on the Hamming weight rule. Finally, we create the 128 -bit SRAM array and load it with the correct bits. The outputs of our key schedule script are the 128-bit SRAM contents and the 8 -bit CV shift register contents.

### 5.2 Encrypt and Decrypt

We created Octave scripts to simulate both Vulcan encrypt and Vulcan decrypt operations. Both require the outputs of the key schedule simulation in order to properly set up the 138 -bit cryptovariable. As was the case with our key schedule simulation, our commented source code listings are in the appendix, with board overviews presented here.

To encrypt, we first check for some common errors, and then we create a random initialization vector if none was supplied. We then iterate over each bit of user supplied plaintext to create a bit of ciphertext according to the Vulcan encryption algorithm. We compute the output of the XOR tree with each stage being conditionally controlled by cryptovariable bits in the CV shift register. Next we differentially encode the resulting ciphertext bit and save it. We then compute the SRAM address of the cryptovariable bit, and then add modulo2 this CV bit to the 21 st bit of the ciphertext shift register, thus producing the modified ciphertext bit. Finally, we clock the ciphertext shift register in preparation for the next plaintext input bit.

Our decrypt simulation is nearly identical to our encrypt simulation; only the order of the operations changes. We once again check for some simple initialization errors and then proceed to iterate on all the bits of the user supplied ciphertext. We differentially decode the ciphertext bit and then compute the SRAM address of the cryptovariable bit. We then add modulo-2 this cryptovariable bit to the 21st bit of the ciphertext shift register, thus producing the modified ciphertext bit. We then compute the output of the XOR tree using the same procedure as the encrypt simulation. Finally, we save the resulting plaintext bit and then clock the ciphertext shift register in preparation for the next ciphertext input bit.

We note here that our software simulations in the appendix were optimized for simplicity of understanding, not for performance. When we turn to crypt-
analysis, we streamline our simulations somewhat, making them more efficient, but somewhat less easy to understand.

## 6 Cryptanalysis

Vulcan is especially vulnerable to cryptanalysis because it is very nearly a linear cipher. Although it has a long (for that era) cryptovariable, the linearity of the cipher defeats any beneficial effect of having a long cryptovariable. Indeed, we consider it ironic that Vulcan has such a long cryptovariable (of either 71 or 138 bits, depending on how you look at it) given that it can be broken almost trivially. In the sections that follow, we show that far from requiring an exhaustive search of order $2^{71}$, Vulcan can be broken with an exhaustive search of only $2^{7}$.

We emphasize that no special knowledge of cryptanalysis is necessary to break Vulcan; an undergraduate-level understanding of linear algebra is more than sufficient. And whereas a more knowledgeable and experienced cryptographer might well come up with a far better attack than the one we present here, ours works in real time and is easy to understand.

From (1) of Section 3, we obtain the necessary insight as to how to break Vulcan. The crucial observation is this: if we view the effect of $G F(2)$ multiplication as an enable/disable mechanism for the individual stages of the XOR tree, then the remainder of Vulcan is completely linear on $G F(2)$. We now expand upon this idea in greater detail.

Suppose that we know the 10-bit contents of the CV shift register (also recall that two of these 10 bits are always 0 ). This implies that we know which stages of the XOR tree actively add (over $G F(2)$ ) modified ciphertext bits to the final result, and which stages merely pass the intermediate result onto the next stage unmodified. Therefore, we can compute the present ciphertext bit as a linear combination of prior ciphertext bits and certain of the 128 unknown CV bits stored in the SRAM.

If we so choose, we can approach cryptanalysis with the assumption that 136 bits of the 138-bit cryptovariable are unknown (recall that two bits are always 0 ). In fact however, only 71 bits of the 138 -bit cryptovariable are linearly independent due to the processing of the DVP key loader. It matters not whether we assume 136 unknown CV bits or 71 unknown CV bits. Our attack works in either case, revealing all 138 bits of the cryptovariable, although slightly less ciphertext is required when we assume only 71 unknown CV bits.

We have repeatedly emphasized that the user-specified DVP cryptovariable is only 71 bits even though the Vulcan cipher itself requires a 138-bit cryptovariable. The matrix in our cryptanalysis simulation in Appendix D lists the 128-bit SRAM contents in terms of the 71 user-entered CV bits. We can apply this matrix to the linear ciphertext equations to produce a second matrix that can be solved to reveal the cryptovariable bits. Our exact procedure now follows.

For each bit of ciphertext we assume a known value of the corresponding
plaintext bit (i.e. we are mounting a known plaintext attack). For our simulation we chose an all-zero plaintext for convenience; however, in a real-world situation a dotting pattern (0101...) would be preferred due to the nature of the CVSD encoder [3].

We now digress briefly to emphasize that a known plaintext attack is completely legitimate when mounted against any cryptosystem based on CVSD audio encoding. This is because approximately half of all spoken speech is silence, and the CVSD waveform coder encodes silence as a simple dotting pattern. Therefore, approximately half of all plaintext will consist of this simple dotting pattern, making the known plaintext assumption a valid one [3].

For any stream cipher, assuming knowledge of the plaintext and having access to the corresponding ciphertext allows us to determine the keystream via modulo-2 addition [1]. In the case of Vulcan, (1) specifies the keystream precisely.

To attack Vulcan, we collect delayed ciphertext bits to determine the corresponding SRAM addresses of the cryptovariable. These CV bits from the SRAM are used to form the keystream, as indicated by (1). Although we do not know the value of these CV bits, we do know which specific CV bits were involved in forming a specific bit of the keystream. Furthermore, since we are mounting a known plaintext attack, we also know the value of the keystream.

The net result is that we end up with a linear equation that relates certain SRAM CV bits to specific keystream bits. Again, we know the addresses of the SRAM contents, but not the values contained in these addresses. By iterating over the ciphertext bits we can create a matrix of linear equations that can be solved to reveal the unknown values of the SRAM contents. Our procedure for accomplishing this is in Appendix D. We now explain how it works.

### 6.1 Automated Procedure

We created two lookup tables and an Octave script to recover the entire cryptovariable from a ciphertext stream. The first lookup table is a 128 row by 64 column matrix that describes the contents of the SRAM in terms of the 64 -bit data word referred to in Section 4. Recall that this data word consists of 64 of the 71 user-entered cryptovariable bits, although not in a straightforward order. Given an SRAM address, this matrix provides the linear combination of CV bits that are stored in that address.

The second lookup table is simply a list of the 128 possible values of the 8 -bit data word mentioned in Section 4. These eight bits reside in the CV shift register and control the operation of the corresponding stages in the XOR tree. Our procedure uses simple linear algebra to solve for 64 of the 71 unknown cryptovariable bits, and uses this list to guess the remaining seven bits until a consistent solution is found. In the worst case we must try all 128 values in the list, although typically the solution comes much sooner than this.

Our cryptanalysis script begins by forming a null solution matrix of 64 rows by 65 columns, and by using the first entry from the second lookup table as the starting value of the eight CV bits in the CV shift register. We then proceed to
process the ciphertext, creating a single linear equation relating SRAM contents to keystream bits for each bit of ciphertext. For each such equation, we update our partially completed solution matrix until one of the following occurs: we found an inconsistent solution, we ran out of ciphertext bits, or we completely filled the solution matrix, which is now in row echelon form.

When we guess the CV shift register bits incorrectly, we will eventually encounter an inconsistent equation (i.e. $0=1$ ) and thus know that our guess was incorrect. Should this happen, we simply move on to the next value in the second lookup table, load that entry as the eight CV shift register bits, and start processing ciphertext bits from the beginning once again.

When we guess the CV shift register bits correctly, our solution matrix will eventually be completely filled and will also be in row echelon form. At this point all we have to do is put the solution matrix in reduced row echelon form and read the solved cryptovariable bits from the rightmost column. Our final solution consists of our correct guess of the CV shift register bits and the rightmost column of the consistently completed solution matrix.

The precise details of our procedure are evident in our commented source code listing of Appendix D, but here we offer a few additional remarks to aid understanding. From (1) of Section 3 we can compute a keystream bit from delayed ciphertext bits and cryptovariable bits. This equation is central to our cryptanalysis routine.

For each bit of ciphertext, our cryptanalysis routine forms a linear equation based on (1) such that we have a linear (on $G F(2)$ ) combination of 64 unknown variables (cryptovariable bits) equal to a single known value (the keystream bit). We place this linear equation in our solution matrix and perform elementary row operations on it to ensure that our solution matrix is in row echelon form at all times.

As we continue to process ciphertext bits (assuming we have an adequate supply), we will eventually reach one of two possible outcomes: a correct solution or an inconsistent solution. An inconsistent solution occurs when, after performing elementary row operations on a candidate equation, we obtain the impossible result that $0=1$. A correct solution occurs when we have completely filled our solution matrix with 64 linearly independent equations, none of which resulted in an inconsistency.

Once we have 64 linearly independent consistent equations in our solution matrix, we again use elementary row operations to fully reduce the matrix into reduced row echelon form. This simply places the 64 solved cryptovariable bits in the rightmost column of the matrix, thus completing our solution.

Experimentally, we have found that approximately 100 ciphertext bits are necessary to generate a complete solution for the Vulcan cryptovariable. If our technique is expanded to allow all 138 bits of the Vulcan cryptovariable to be independently specified - a situation not permitted by the DVP key loader - then we can still solve for the 138 bits, but we require about an order of magnitude (i.e. 1000 bits) more ciphertext to obtain a full solution. Either way, our cryptanalysis routine runs in real time and does not require any significant computational resources. We are fully confident in our ability to recover a DVP
cryptovariable in real time solely from ciphertext captured over-the-air from an actual DVP-equipped radio (although we did not actually do so because we were unable to procure a working DVP-equipped radio).

Given that it is possible to recover the Vulcan cryptovariable from ciphertext in real time, we must declare this cipher to be trivially weak and unsuitable for providing even rudimentary privacy. Although modern computing resources have made such an attach simple, they are not necessary.

Such an attack could have been accomplished 40 years ago, albeit somewhat less conveniently. We therefore speculate that DVP-secured radio communications were routinely compromised during the era of their use, although we have no proof of this. Believing otherwise simply cannot be supported given the ease of breaking Vulcan. The inherent weakness of Vulcan raises a number of interesting questions, which are the topic of our next section.

## 7 Conclusion

In this paper we have completely exposed the proprietary Vulcan cipher used in Motorola DVP encryption modules of the 1970s. Furthermore, we have shown that Vulcan is trivially weak and we have presented a simple technique that enables real-time ciphertext-only cryptanalysis. Before ending this paper, we wish to present a few open questions, propose future work, and offer some general comments.

### 7.1 Open Questions

During our reverse-engineering and analysis of Vulcan, a number of questions came to mind. In this section we list some of those questions and offer our opinions regarding likely answers.

Our first question is why Vulcan uses a 71-bit cryptovariable. This is a very unusual length and we are not aware of any other cipher with such a strange length for the cryptovariable. Furthermore, the end user is required to enter the cryptovariable as 24 octal digits, but this creates the awkward situation where the very last octal digit is restricted to the range of 0 to 3 . Why did Motorola choose to truncate this last octal digit instead of simply allowing the user to enter 72 bits? This makes no sense to us.

For that matter, why did Motorola not allow the user to enter all 138 bits of the Vulcan cryptovariable? We can only speculate as to the reasons behind this decision, but our best guess is that Motorola marketing felt that 138 bits, presumably entered as 46 octal digits, was simply too long for most users to manage.

Even entering 24 octal digits in four groups of six digits, as must be done on the DVP key loader, is somewhat cumbersome. Certainly entering 46 octal digits using a primitive user interface such as that of the DVP key loader would be bothersome. Nonetheless, Vulcan requires a 138-bit cryptovariable. It seems
to us that it would have made more sense to either have allowed the user to enter all 138 bits, or to have made Vulcan use a shorter cryptovariable.

As an aside, we also note that the DVP key loader lacks the ability to generate a random cryptovariable, requiring the user to enter a CV of their own creation. This is certain to lead to poorly-chosen values, as our experience is that many users of cryptographic equipment, when allowed to choose their own CV, will simply enter an easily guessed sequence such as $1234 \ldots$, etc.

One might intuitively think that limiting the CV to 71 bits greatly reduces overall security, given that the Vulcan CV is natively 138 bits; however, since the Vulcan cipher is effectively linear, the length of the CV is of little consequence. Solving a system of linear equations in 138 unknowns is scarcely more difficult than solving a system of linear equations in 71 unknowns (or far fewer, for that matter). Motorola most likely realized this and made their decision on cryptovariable length accordingly.

Another possibility regarding the length of the cryptovariable is the influence of external constraints, such as export restrictions. We know that in the 1990s, the US government would only allow export of cryptography restricted to 40 or fewer bits of cryptovariable [4]. We do not know what the restrictions were in the 1970s, but we can safely assume they were no more permissive than they were in the 1990s, and were likely less so. Again we are left to guess as to what the true reasons behind Vulcan's unusual CV length were.

Our second question is why Vulcan is so weak. Admittedly, cryptography was much less well understood in the early 1970s than it is today; nonetheless, anyone with an undergraduate education in mathematics would have known at that time that Vulcan was seriously weak. It would not surprise us if this was deliberate.

Inspection of the SC76807 IC layout reveals certain clues as to alternatives the designers might have had in mind. Specifically, the XOR tree has provisions to accommodate additional bits from the CV shift register, yet this circuitry is not hooked up. Also, the ciphertext shift register has a provision for insertion or extraction of bits approximately half-way through its 21 bits, a feature that was not used and the purpose of which is not clear to us.

Strictly speaking, Vulcan is nonlinear due to the $G F(2)$ multiplications present in the XOR tree; however, as we have shown, we can linearize Vulcan by assuming these $G F(2)$ multiplications simply represent bits that must be solved for via exhaustive search. Since there are only seven unknowns, this exhaustive search is trivial. The designers could have made exhaustive search far less fruitful by including more bits from the CV shift register in the $G F(2)$ multiplications, but they chose not to do so. We are puzzled by this.

Furthermore, even a simple nonlinear operation in the ciphertext mixing function would have eliminated the simple attack of linear algebra, but the designers chose not to do this either. We cannot offer a reasonable explanation why a simple nonlinearity was not included in order to strengthen Vulcan. Our suspicion is that Vulcan is deliberately weak, although we have no proof of this. Nonetheless, we prefer this explanation to the alternative possibility that the designers were simply incompetent. Now that Vulcan has been thoroughly
exposed, it would be enlightening if one of its creators stepped forward with the rationale behind the design decisions.

If Vulcan was designed solely to prevent casual eavesdropping then it was undoubtedly successful in achieving that goal. We are unaware of so much as a rumor of anyone publicly admitting success in recovering DVP encrypted voice. Indeed, most hams and hobbyists seem to have been sufficiently frustrated by various analog scrambling schemes, which were widely used to protect radio communications during the same era as DVP [11].

On the other hand, if Vulcan was intended to prevent determined adversaries from monitoring secret communications, we cannot believe it would have had any success in this regard. No intelligence agency anywhere in the world would have had difficulty cracking Vulcan in the 1970s. Perhaps when NSA declassifies some of their signals intelligence operations of the 1970s - something not likely to occur for another 60 years - future cryptographers will have solid evidence of this claim.

### 7.2 Future Work

Regarding future work, we have little doubt that a skilled cryptographer can greatly improve upon our simple cryptanalytic attack, although there is little motivation to do so. Such talent should be reserved for greater challenges, which leads us to comment that that Vulcan is merely one of several proprietary ciphers developed by Motorola. We suspect that the others might prove far more interesting.

Specifically, we know that Vulcan was replaced with an allegedly much stronger cipher known internally as Linus sometime in the early 1980s. Linus was sold under the trade name DVP-XL and is alleged to have a 96-bit cryptovariable [6]. We know that the early implementations of Linus were done on a fully custom programmable cryptographic IC known as Son of Vulcan or SOV and marked as SC380001.

The challenge in reverse-engineering SOV is that it is a programmable device that can implement many different ciphers, including Vulcan itself, depending on how it is programmed. Not only would one have to reverse-engineer the hardware, one would also have to successfully extract the contents of the EEPROM memory of the SOV IC and determine how this data controls the chip.

Son of Vulcan was also used to implement algorithms other than Vulcan and Linus. Specifically, we know of an export algorithm referred to internally as Lucy and marketed at DVI-XL, as well as several different variations of Lucy that were sold as DVI-SPFL for Lucy Special Flavor. There are rumored to be many special flavors of Lucy, but we have no confirmation of this.

Nothing is publically known about Linus or any of the various flavors of Lucy. We would be eager to learn the details of these ciphers, but reverse-engineering them will be considerably more difficult than reverse-engineering Vulcan. We would hope that these algorithms are more secure than Vulcan, although we have no doubt that Lucy was deliberately weakened to comply with US export restrictions on cryptography.

Linus and Lucy might also have been implemented in forms other than the SOV IC, perhaps even in software. We are simply unaware of specific details and would welcome further information and analysis. As with Vulcan, all of these proprietary ciphers have been rendered obsolete in favor of more recent ciphers such as AES. Nonetheless, Linus and Lucy are tempting targets.

For that matter, Motorola is far from the only manufacturer that sold proprietary cryptographic devices. We know of many different manufacturers of equipment, much of which is now cheaply available at ham fests and on eBay. Recent advances in tools and techniques for reverse-engineering place most historical cryptographic equipment within easy reach of anyone with the will to understand how these devices work.

### 7.3 Epilog

We undertook this endeavor primarily as a learning exercise. We hope that others will find this material both interesting and informative. Furthermore, we hope this project inspires others to undertake similar projects, uncovering the hidden secrets lurking beneath every proprietary cryptographic device. Many such proprietary devices exist, providing ample material for would-be reverseengineers and cryptographers. The Son-of-Vulcan device we mentioned is but one of a plethora.

Many older cryptographic devices are simple enough to allow a complete analysis by a single individual. However, some devices are sufficiently complex to require a team effort to fully reverse-engineer and understand. To this end, we recognize and wish to call attention to the need for an anonymous and secure collaboration framework that would allow multiple team members to remain anonymous while still enabling cooperation on potentially sensitive work such as cryptographic reverse engineering. Anyone seeking a noble thesis project should give serious thought to this most critical need.

## References

[1] A. Klein, "Introduction to Stream Ciphers", in Stream Ciphers, 1st ed., London, UK, Springer, 2013, ch. 1, sec. 1.2, pp. 6-7.
[2] S. Golomb, Shift Register Sequences, 1st ed., Laguna Hills, CA, Aegean Park Press, 1982.
[3] J. Greefkes, K. Riemens, "Code Modulation with Digitally Controlled Companding for Speech Transmission", Philips Technical Review, vol. 31, no. 11-1, pp. 335, 1970.
[4] Export of Cryptography from the United States, http://en.wikipedia.org/wiki/Export_of_cryptography_from_the_United_States
[5] A Legacy of Innovation: Timeline of Motorola History since 1928, http://www.motorolasolutions.com/US-EN/About/Company+Overview/History/Timeline
[6] Motorola Encryption Topics, http://wiki.radioreference.com/index.php/Motorola_Encryption_Topics
[7] Encryption Protocols: A Brief Look at Analog and Digital Encryption Devices, http://www.radioreference.com/trunked/stuff/encrypt.html
[8] 2-Way Encryption Primer, http://www.batlabs.com/encrypt.html
[9] KY-57 Vinson Voice Encryption Unit, http://www.cryptomuseum.com/crypto/usa/ky57/
[10] The Layman's Guide to IC Reverse Engineering, http://siliconzoo.org/tutorial.html
[11] Voice Security FAQs, http://www.midians.com/voice-security-faqs
[12] W. Braun, A. Leitich, "Digital Privacy System", U.S. Patent 3,639,690, February 1, 1972.
[13] R. Coe, D. McQuade, D. Weiss, "Digital Voice Protection System and Method", U.S. Patent 4,167,700, September 11, 1979.
[14] M. Bright, E. Ziolko, A. Wilson, M. Bray, H. Hennen, D. Weiss, "Secure Communication System", U.S. Patent 4,893,339, January 9, 1990.
[15] Motorola, Inc., "Securenet", U.S. Trademark Reg. Num. 1842824, January 1, 1982.

## A Vulcan Key Schedule Simulation

```
%------------------------------------------------------------------
% Vulcan "key schedule" simulation
% 2014-05-28 --- CJR
% input: 24-digit octal cryptovariable "cv"
% outputs: 128-bit SRAM array "sram"
% 8-bit CV shift register "cv8"
% converts 71-bit user-supplied CV into 138-bit Vulcan CV
% prints out various intermediate results for inspection
%----------------------------------------------------------------
% check for user-supplied input & substitute default if none
if((exist('cv') == 0)||(length(cv) != 24))
    cv = [7 [7 6 5 4 4 3 2 1 0,\ldots..
                7654 3 2 1 0,...
            7654 3 2 1 0];
end
% convert CV from 24 octal digits to 72 bits
cvb = zeros(1,72);
for ii = 1:24
    dig = cv(ii);
    for jj = 1:3
        cvb(3*(ii-1)+(3-jj)+1) = mod(dig,2);
        dig = floor(dig/2);
    end
end
printf("CV = %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %Od %Od %Od %Od %Od %Od %Od\n", reshape(cvb,3,24)'*[4 2 1]');
% toss out MSB of last octal digit to get correct 71-bit CV
cvb = [cvb(1:69) cvb(71:72)];
% form 64-bit data word from 64 bits of CV
cv64 = reshape(cvb(1:66),3,22)';
cv64 = reshape(flipud(cv64)',1,66);
cv64 = cv64(3:66);
printf("data64 = %02X %02X %02X %02X %02X %02X %02X %02X\n",...
    reshape(cv64,8,8)'*[128 64 32 16 8 4 2 1]');
% compute 64-bit parity word from 64-bit data word
pr64 = zeros(1,64);
taps = [0, 1, 0, 1, 0, 0, 1, 0,...
            0, 0, 0, 0, 0, 0, 0, 0,...
            1, 0, 0, 1, 1, 0, 1, 1,...
            1, 1, 1, 1, 0, 0, 0, 1,...
            1, 1, 1, 0, 1, 1, 0, 1,...
            0, 1, 0, 0, 0, 0, 0, 0,...
            1, 1, 1, 0, 1, 0, 1, 0,\ldots
            1, 1, 0, 0, 0, 0, 1, 0];
for ii = 64:-1:1
    bb = bitxor(cv64(ii),pr64(64));
    pr64 = [0 pr64(1:63)];
    if(bb == 1)
            pr64 = bitxor(pr64,taps);
    end
end
pp = mod(sum(cv64),2);
pr64 = [pp pr64(2:64)];
printf("parity64 = %02X %02X %02X %02X %02X %02X %02X %02X\n",...
    reshape(pr64,8,8)'*[128 64 32 16 8 4 2 1]');
```

```
% form 8-bit data word from 7 bits of CV
cv8 = [0 cvb(70) cvb(71) cvb(67) cvb(68) cvb(69) cvb(64) cvb(65)];
if(sum(cv8) <= 3)
    cv8 = bitxor(cv8, ones(1,8));
end
printf("data8 = %02X\n", cv8*[128 64 32 16 8 4 2 1]');
% clock out the 138-bit cryptovariable in correct order
clk = [pr64(64) pr64(63) pr64(62) pr64(61) pr64(60) pr64(59) pr64(58) cv64(64)];
clk = [clk pr64(57) pr64(56) pr64(55) pr64(54) pr64(53) cv64(63) cv64(62) pr64(52)];
clk = [clk pr64(51) pr64(50) pr64(49) cv64(61) pr64(48) cv64(60) pr64(47) pr64(46)];
clk = [clk pr64(45) cv64(59) cv64(58) cv64(57) cv64(56) pr64(44) pr64(43) cv64(55)];
clk = [clk pr64(42) pr64(41) pr64(40) cv64(54) pr64(39) cv64(53) cv64(52) pr64(38)];
clk = [clk pr64(37) cv64(51) cv64(50) cv64(49) pr64(36) cv64(48) pr64(35) cv64(47)];
clk = [clk pr64(34) pr64(33) cv64(46) cv64(45) cv64(44) cv64(43) cv64(42) pr64(32)];
clk = [clk cv64(41) pr64(31) pr64(30) pr64(29) pr64(28) cv64(40) cv64(39) cv64(38)];
clk = [clk pr64(27) pr64(26) pr64(25) cv64(37) pr64(24) pr64(23) cv64(36) pr64(22)];
clk = [clk pr64(21) cv64(35) cv64(34) pr64(20) cv64(33) cv64(32) pr64(19) cv64(31)];
clk = [clk pr64(18) cv64(30) cv64(29) pr64(17) cv64(28) cv64(27) cv64(26) cv64(25)];
clk = [clk pr64(16) cv64(24) cv64(23) pr64(15) pr64(14) pr64(13) cv64(22) cv64(21)];
clk = [clk pr64(12) cv64(20) pr64(11) pr64(10) cv64(19) pr64(09) cv64(18) cv64(17)];
clk = [clk cv64(16) pr64(08) cv64(15) cv64(14) cv64(13) pr64(07) pr64(06) cv64(12)];
clk = [clk cv64(11) pr64(05) pr64(04) cv64(10) pr64(03) cv64(09) pr64(02) cv64(08)];
clk = [clk pr64(01) cv64(07) cv64(06) cv64(05) cv64(04) cv64(03) cv64(02) cv64(01)];
clk = [clk 0 0 cv8(8) cv8(7) cv8(6) cv8(5) cv8(4) cv8(3) cv8(2) cv8(1)];
printf("138 bits clocked out of DVP key loader:\n");
printf("%Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %Od %Od %Od %Od %Od\n", clk(1:23));
printf("%Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %Od %Od %Od %Od %Od\n", clk(24:46));
printf("%Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %Od %Od %Od %Od %Od\n", clk(47:69));
printf("%Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %Od %Od %Od %Od %Od\n", clk(70:92));
printf("%Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %0d %Od %0d %0d %Od\n", clk(93:115));
printf("%Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od %Od...
    %0d %Od %Od %Od %Od\n", clk(116:138));
% place 64 data bits and 64 parity bits into 128-bit SRAM at correct addresses
sram = zeros(1,128);
addr = [000, 001, 002, 004, 008, 016, 032, 065,...
003, 006, 012, 024, 048, 097, 066, 005,...
010, 020, 040, 081, 035, 071, 015, 030,...
060, 121, 114, 100, 072, 017, 034, 069,...
011, 022, 044, 089, 051, 103, 078, 029,...
058, 117, 106, 084, 041, 083, 039, 079,...
031, 062, 125, 122, 116, 104, 080, 033,...
067, 007, 014, 028, 056, 113, 098, 068,...
009, 018, 036, 073, 019, 038, 077, 027,...
054, 109, 090, 053, 107, 086, 045, 091,...
055, 111, 094, 061, 123, 118, 108, 088,...
049, 099, 070, 013, 026, 052, 105, 082,...
037, 075, 023, 046, 093, 059, 119, 110,...
092, 057, 115, 102, 076, 025, 050, 101,...
074, 021, 042, 085, 043, 087, 047, 095,...
063, 127, 126, 124, 120, 112, 096, 064];
for ii = 1:128
    sram(addr(ii)+1) = clk(ii);
```

end

## B Vulcan Encrypt Simulation

```
%--------------------------------------------------------------------
% Vulcan encryption simulation
% 2014-06-12 --- CJR
% inputs: 128-bit SRAM array contents "sram"
% 8-bit CV shift register contents "cv8"
% n-bit plaintext "pti"
% (31-bit ciphertext shift register contents "ctsr")
% outputs: n-bit ciphertext "cto"
% encrypts plaintext into ciphertext using Vulcan cipher
% requires "key schedule" to have been run prior to use
%--------------------------------------------------------------
% check user-supplied inputs for simple errors
if((exist('sram') == 0)||(length(sram) != 128))
    error("128-bit SRAM array not found -- run key schedule")
end
if((exist('cv8') == 0)||(length(cv8) != 8))
    error("8-bit CV shift register not found -- run key schedule")
end
if(exist('pti') == 0)
    error("plaintext input not found -- create some plaintext")
end
if((exist('ctsr') == 0)||(length(ctsr) != 31))
    ctsr = 1*(rand(1,31)>0.5); % create random initialization vector
end
% set up required variables
last = 0; % last plaintext output bit for differential encoder
cto = zeros(1,length(pti)); % pre-allocate ciphertext output
% process all bits of plaintext input
for ii = 1:length(pti)
    % compute the XOR tree results
    ct0 = pti(ii);
    ct0 = bitxor(ct0,bitand(cv8(1),ctsr(22)));
    ct0 = bitxor(ct0,bitand(cv8(2),ctsr(23)));
    ct0 = bitxor(ct0,bitand(cv8(3),ctsr(24)));
    ct0 = bitxor(ct0,bitand(cv8(4),ctsr(25)));
    ct0 = bitxor(ct0,bitand(cv8(5),ctsr(26)));
    ct0 = bitxor(ct0,bitand(cv8(6),ctsr(27)));
    ct0 = bitxor(ct0,bitand(cv8(7),ctsr(28)));
    ct0 = bitxor(ct0,bitand(cv8(8),ctsr(29)));
    ct0 = bitxor(ct0,ctsr(30));
    % differentially encode and save the output ciphertext bit
    cto(ii) = last;
    last = bitxor(ct0,last);
    % compute the current SRAM address
    addr = ctsr(1:7);
    addr = addr*[[1 2 4 4 8 16 32 64 []';
    addr = addr+1;
    % combine SRAM bit with CT21
    keyb = sram(addr);
    m0 = bitxor(ctsr(21),keyb);
    % clock the ciphertext shift-register
    ctsr = [ct0 ctsr(1:20) m0 ctsr(22:30)];
end
```


## C Vulcan Decrypt Simulation

```
%-------------------------------------------------------------------
% Vulcan decryption simulation
% 2014-06-04 --- CJR
% inputs: 128-bit SRAM array contents "sram"
% 8-bit CV shift register contents "cv8"
% n-bit ciphertext "cti"
% outputs: n-bit plaintext "pto"
% > first 31 bits of plaintext will be gibberish
% decrypts ciphertext into plaintext using Vulcan cipher
% requires "key schedule" to have been run prior to use
%----------------------------------------------------------------
% check user-supplied inputs for simple errors
if((exist('sram') == 0)||(length(sram) != 128))
    error("128-bit SRAM array not found -- run key schedule")
end
if((exist('cv8') == 0)||(length(cv8) != 8))
    error("8-bit CV shift register not found -- run key schedule")
end
if(exist('cti') == 0)
    error("ciphertext input not found -- create some ciphertext")
end
% set up required variables
last = 0; % last ciphertext input bit for differential decoder
ctsr = zeros(1,31); % 31-bit ciphertext shift-register
pto = zeros(1,length(cti)); % pre-allocate plaintext output
% process all bits of ciphertext input
for ii = 1:length(cti)
    % differentially decode the ciphertext input bit
    ctO = bitxor(cti(ii),last);
    last = cti(ii);
    % compute the current SRAM address
    addr = ctsr(1:7);
    addr = addr*[1 2 4 8 16 32 64]';
    addr = addr+1;
    % combine SRAM bit with CT21
    keyb = sram(addr);
    m0 = bitxor(ctsr(21),keyb);
    % compute the XOR tree results
    pt = ct0;
    pt = bitxor(pt,bitand(cv8(1),ctsr(22)));
    pt = bitxor(pt,bitand(cv8(2),ctsr(23)));
    pt = bitxor(pt,bitand(cv8(3),ctsr(24)));
    pt = bitxor(pt,bitand(cv8(4),ctsr(25)));
    pt = bitxor(pt,bitand(cv8(5),ctsr(26)));
    pt = bitxor(pt,bitand(cv8(6),ctsr(27)));
    pt = bitxor(pt,bitand(cv8(7),ctsr(28)));
    pt = bitxor(pt,bitand(cv8(8),ctsr(29)));
    pt = bitxor(pt,ctsr(30));
    % save decrypted plaintext
    pto(ii) = pt;
    % clock the ciphertext shift-register
    ctsr = [ct0 ctsr(1:20) m0 ctsr(22:30)];
end
```


## D Vulcan Cryptanalysis Simulation

```
%---------------------------------
% 2014-07-15 --- CJR
% input: n-bit ciphertext "ct"
% output: 71-bit cryptovariable "cv"
% recovers Vulcan CV from ciphertext assuming all-0 plaintext
%-
% check user-supplied inputs for simple errors
if(exist('ct') == 0)
    error("ciphertext not found -- create some ciphertext")
end
% table of 128-bit SRAM contents in terms of 64 CV bits (of 71-bit user CV)
% each row represents the contents of the corresponding SRAM address from 000 to 127
% the columns are user CV bits v70 v69 v68 v67 ... v10 v09 v08 v05
% (note that rows containing more than a single 1 represent parity bits)
% (note that user CV bits v07 v06 v04 v03 v02 v01 v00 are NOT stored in SRAM)
sr = [...
1,0,1,1,1,0,1,0,1,0,0,1,1,1,0,1,1,0,1,0,1,0,0,0,0,1,1,1,0,1,0,0,\ldots
1,1,1,0,1,1,1,0,1,1,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1,1,1,0,1,0;\ldots
0,1,0,1,0,1,0,1,1,0,1,1,1,0,0,1,0,1,0,1,1,0,0,1,1,1,0,0,1,0,0,1,\ldots
1,1,0,0,1,1,1,1,1,1,1,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,1,0,0,1,0,1;\ldots
0,0,1,1,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1,0,0,1,0,1,1,1,0,0,0,1,1,\ldots
1,1,1,0,0,0,0,1,0,0,1,1,1,0,1,1,0,0,0,1,1,0,0,1,1,1,1,1,0,0,0,0;\ldots
1,0,1,0,1,0,1,0,1,0,1,1,0,1,1,1,0,1,0,1,0,0,1,0,0,1,0,0,0,0,0,1,\ldots
0,0,1,0,1,0,0,0,0,0,1,1,0,1,1,0,1,1,1,1,1,1,0,0,0,1,0,0,0,0,0,1;\ldots
0,1,0,0,0,0,1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,0,0,1,1,0,0,0,1,1,1,\ldots
0,1,0,1,0,0,1,0,0,0,1,1,1,1,1,1,0,0,0,1,0,0,1,1,1,1,1,0,0,0,1,0;\ldots
0,0,1,0,1,0,1,0,1,0,0,1,0,1,1,1,0,0,0,1,0,0,1,1,1,1,1,1,1,1,1,0,\ldots
0,1,0,1,1,0,1,0,1,0,0,1,0,1,1,1,0,1,0,0,0,1,0,0,0,1,0,1,1,1,0,0;\ldots
1,1,1,0,1,1,1,1,1,1,1,0,0,0,0,1,0,0,0,0,0,1,0,0,1,1,1,1,0,1,1,0,\ldots
1,0,1,0,1,1,1,0,1,0,0,1,0,1,0,1,1,1,1,0,0,1,0,0,1,0,1,1,1,0,0,0;...
0,1,1,0,0,0,1,0,0,0,0,0,0,1,0,1,1,0,1,0,0,1,0,1,1,1,0,0,1,1,0,1,\ldots
0,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,1,0,1,0,0,1,1,1,1,0,0,0,0,1,0;...
1,0,0,0,0,0,0,1,0,1,0,1,0,1,1,1,1,1,0,1,0,0,0,1,1,0,1,0,1,1,1,0,\ldots
0,0,1,1,0,1,1,0,0,0,1,1,0,1,1,1,0,0,1,0,0,0,1,1,1,1,0,1,0,1,1,0;\ldots
1,0,1,0,0,0,1,0,1,0,0,1,1,1,1,0,1,0,0,0,1,0,0,1,0,0,0,0,0,0,1,1,\ldots
0,0,1,0,1,1,1,0,1,0,1,0,1,0,0,0,0,0,0,0,1,0,0,0,1,1,0,0,1,0,0,1;...
0,1,0,1,0,0,0,1,0,0,1,1,1,1,0,0,0,0,1,0,1,1,1,0,1,1,0,1,1,1,0,0,\ldots
1,0,1,1,0,1,1,1,0,1,1,0,0,1,1,0,1,0,1,0,1,0,0,0,0,0,1,0,1,0,1,1;\ldots
0,0,0,0,0,0,0,1,1,0,1,1,1,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,1,1,\ldots
1,1,0,0,0,1,1,0,1,1,1,0,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,1,0,0,0,1;...
1,1,0,1,1,1,1,1,0,1,0,1,0,0,1,0,0,0,0,0,1,0,0,0,1,1,1,0,1,1,0,1,\ldots
1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,0,1,1,1,0,1,1,0,1,0,1,1,0,0,0,0,1;\ldots
0,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,0,0,0,1,1,0,0,0,0,1,0,\ldots
0,1,1,0,1,0,0,1,1,1,0,0,0,0,0,1,0,1,1,1,1,1,0,1,0,0,0,0,0,0,0,0;\ldots
1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,0,\ldots
0,0,0,0,0,0,1,0,1,1,0,0,0,0,0,0,1,0,0,0,0,1,1,1,1,0,0,1,0,1,1,0;\ldots
1,1,0,1,1,0,0,1,0,1,0,1,0,1,1,1,0,0,0,1,1,1,0,0,1,1,1,0,0,1,1,1,\ldots
1,1,1,0,0,0,0,0,0,1,1,0,0,1,0,1,1,0,1,1,1,1,0,0,1,0,1,0,0,0,0,0;\ldots
0,0,0,0,0,1,1,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1,0,0,1,0,1,1,1,0,0,\ldots
0,1,1,1,1,1,0,0,0,0,1,0,0,1,1,1,0,1,1,0,0,0,1,1,0,0,1,1,1,1,1,0;\ldots
0,0,0,1,1,0,1,1,0,0,1,0,1,0,1,0,1,1,1,0,0,0,1,1,1,0,0,1,1,1,0,0,\ldots
1,1,1,1,1,1,0,0,0,0,0,0,1,1,0,0,1,0,1,1,0,1,1,1,1,0,0,1,0,1,0,0;\ldots
1,1,1,1,1,1,1,1,1,0,1,0,0,0,0,0,1,1,1,1,1,0,1,0,0,1,1,1,0,1,1,0,\ldots
1,0,1,1,0,0,1,1,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,1,0,0,1,1,1,0,0,1;\ldots
0,1,1,1,0,0,0,0,1,0,0,1,0,1,1,1,1,1,0,1,1,0,1,0,0,0,0,0,1,0,1,0,\ldots
```


#### Abstract

$0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,0,0,0,0,0,0,0,1,1,0,1 ; \ldots$ $0,0,1,1,1,0,0,0,0,1,1,1,0,1,0,1,1,1,1,1,1,1,0,1,1,1,0,0,1,0,0,1, \ldots$ $0,0,0,0,0,0,1,0,0,1,1,1,1,1,0,1,0,1,1,0,1,1,0,0,0,1,1,1,1,1,1,1 ; \ldots$ $0,1,0,1,0,1,0,1,1,1,1,1,0,1,0,0,0,1,1,0,1,0,1,1,1,0,0,0,1,1,0,1, \ldots$ $1,0,0,0,1,1,0,1,1,1,0,0,1,0,0,0,1,1,1,1,0,1,0,1,1,0,0,0,1,0,0,0 ; \ldots$ $1,0,1,1,1,0,0,1,1,1,1,1,1,0,0,0,1,1,1,1,0,1,0,0,0,0,0,0,0,1,1,1, \ldots$ $0,1,1,1,0,0,0,1,0,1,1,1,0,0,0,1,1,0,0,0,1,1,1,0,1,1,0,0,1,0,0,0 ; \ldots$ $0,0,0,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0, \ldots$ $1,0,0,0,1,0,1,0,0,1,1,1,1,0,0,1,0,1,0,1,1,0,1,1,0,1,0,1,0,0,0,0 ; \ldots$ $0,0,1,0,0,1,0,0,0,0,1,0,1,0,1,1,1,0,1,1,1,0,0,0,1,0,1,0,1,0,1,1, \ldots$ $1,1,0,0,0,1,0,0,0,0,1,1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,1,1,1,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,0,1,1,0,1,1,0,0,0,1,0,0,0,0,0,0, \ldots$ $0,0,1,1,0,0,0,0,1,0,0,1,1,1,1,0,0,1,1,1,0,0,1,0,0,1,0,1,0,1,1,1 ; \ldots$ $1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,0,0, \ldots$ $0,1,0,1,0,0,1,1,1,1,0,0,1,0,1,0,1,1,0,1,1,0,1,0,1,0,0,0,0,0,0,0 ; \ldots$ $0,1,1,0,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,1,0,0,0,1,0,0,1,1,1,0,1, \ldots$ $1,0,0,0,1,1,0,1,1,0,1,0,0,1,0,1,0,1,1,0,1,0,0,0,0,1,0,1,1,1,1,0 ; \ldots$ $1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,0,1,1,1,1,0,0,1,0,1,1,1,0,0, \ldots$ $1,0,0,1,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,1,1,1,1,0,0 ; \ldots$ $0,1,0,0,0,0,1,0,1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,1,0,1,0,0,1,0,0,1, \ldots$ $0,0,1,0,0,0,0,1,1,1,0,0,1,1,1,1,0,0,1,0,0,0,0,1,0,0,1,1,1,1,0,1 ; \ldots$ $1,0,0,1,0,1,1,0,1,0,1,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,1,1,1,1, \ldots$ $0,1,0,1,0,0,1,0,1,0,0,0,0,0,1,1,0,1,1,1,1,1,0,1,1,1,0,0,0,0,0,0 ; \ldots$ $0,0,1,1,0,1,1,0,0,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1,0,0,1,1,1,0,1,1, \ldots$ $0,0,1,1,0,1,0,0,1,1,0,1,0,1,1,0,1,1,0,1,0,1,1,0,1,0,1,1,1,0,1,1 ; \ldots$ $0,0,0,0,1,0,0,0,0,1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,0,0,1,1,0,0,0, \ldots$ $1,1,1,0,1,0,1,0,0,1,0,0,0,1,1,1,1,1,1,0,0,0,1,0,0,1,1,1,1,1,0,1 ; \ldots$ $0,1,1,1,1,1,1,0,0,0,0,0,0,1,0,1,0,0,1,0,0,1,1,0,0,1,0,1,1,1,0,0, \ldots$ $1,1,0,1,0,0,1,0,0,1,0,1,1,0,0,0,0,1,1,1,0,1,1,1,1,0,1,1,0,1,0,1 ; \ldots$ $0,0,0,1,0,0,1,0,1,1,0,1,0,1,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,1, \ldots$ $1,1,1,0,1,0,1,0,0,1,0,1,0,0,0,0,0,1,1,0,1,1,1,1,1,0,1,1,1,0,0,0 ; \ldots$ $0,1,1,0,1,1,1,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,1,1,1,1,0,0,0,1, \ldots$ $1,0,1,1,1,0,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,1,0,0,0,0,0,1,0,0,0,0 ; \ldots$ $0,1,1,0,0,1,0,1,0,1,0,0,1,1,0,0,0,0,0,1,1,1,0,1,1,0,1,1,1,1,0,1, \ldots$ $0,0,0,1,1,0,1,1,1,1,1,1,1,0,1,1,0,0,0,0,1,0,1,0,0,1,0,1,1,0,1,1 ; \ldots$ $0,0,0,0,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,0,0,0,1,1,0,0,0, \ldots$ $0,1,0,0,1,1,0,1,0,0,1,1,1,0,0,0,0,0,1,0,1,1,1,1,1,0,1,0,0,0,0,0 ; \ldots$ $0,1,1,1,1,0,1,1,0,0,1,1,0,0,0,0,0,0,0,1,0,1,0,0,0,1,1,0,0,1,0,0, \ldots$ $1,1,1,1,1,0,0,0,1,0,1,0,1,1,1,0,0,0,1,0,1,0,0,0,0,0,1,1,0,0,1,1 ; \ldots$ $1,1,0,1,0,1,1,0,0,0,0,0,1,0,0,1,0,1,1,0,1,0,1,1,1,0,1,1,0,1,1,1, \ldots$ $1,1,1,0,0,1,0,0,0,0,0,1,0,0,1,1,1,1,1,1,0,1,0,1,0,0,0,0,1,0,0,1 ; \ldots$ $1,1,1,0,1,0,1,0,0,1,1,1,0,1,1,0,0,0,0,1,0,0,1,1,1,1,0,0,0,0,1,0, \ldots$ $1,1,1,0,1,0,1,0,0,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,1,0,1,0,1,0,1 ; \ldots$ $1,1,1,0,1,0,0,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,0,1,1,1,0,1,1,0,0,1, \ldots$ $1,0,1,1,0,0,1,0,0,0,1,0,1,1,0,1,1,1,1,0,1,0,0,0,1,0,0,0,1,1,0,0 ; \ldots$ $1,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1,0,0,1,0,1,1,1,0,0,0,1,1,1,1,1, \ldots$ $0,0,0,0,1,0,0,1,1,1,0,1,1,0,0,0,1,1,0,0,1,1,1,1,1,0,0,0,0,0,0,1 ; \ldots$ $1,0,1,0,1,1,0,1,1,1,0,0,1,0,1,0,1,1,0,0,1,1,1,0,0,1,0,0,1,1,1,0, \ldots$ $0,1,1,1,1,1,1,1,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,1,0,0,1,0,1,0,1,0 ; \ldots$ $0,1,0,1,0,1,1,1,1,1,1,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,1,0,1,0,1,0, \ldots$ $0,1,1,1,0,0,1,0,1,0,1,0,1,0,1,1,0,0,0,1,1,1,0,1,0,0,0,1,0,0,1,1 ; \ldots$ $1,1,1,0,0,0,1,0,0,0,1,0,1,0,0,1,0,1,0,0,0,0,0,0,1,0,1,0,1,1,1,0, \ldots$ $1,0,0,1,0,0,0,0,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,1,0,1,0,1,0,0 ; \ldots$ $0,0,0,0,0,0,1,0,1,0,0,1,1,0,1,1,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1, \ldots$ $1,0,0,0,0,1,0,0,1,1,1,1,0,0,1,1,1,0,0,1,0,0,1,0,1,0,1,1,0,0,1,0 ; \ldots$ $0,1,0,1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,0,1,0,1,0,0,1,0,1,1,1,0,0,0, \ldots$ $0,1,1,1,1,1,1,0,0,1,0,1,1,0,0,0,0,0,0,0,1,1,1,0,0,1,0,1,0,1,0,1 ; \ldots$


#### Abstract

$0,1,0,0,1,0,0,0,0,1,0,0,0,1,0,1,0,1,1,1,1,0,0,1,0,1,0,1,0,1,1,1, \ldots$ $0,0,0,1,1,0,1,0,0,0,1,1,0,0,1,1,1,0,1,0,1,1,0,0,0,1,1,1,0,0,1,1 ; \ldots$ $1,0,1,0,0,1,1,0,1,1,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1, \ldots$ $0,0,1,1,1,1,0,0,1,1,1,0,0,1,0,0,1,0,1,0,1,1,0,0,1,0,0,0,0,0,0,0 ; \ldots$ $1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,0,0,1,1,0,0,0,1,1,1,0,1,0,1,0,0, \ldots$ $1,0,0,0,1,1,1,1,1,1,0,0,0,1,0,0,1,1,1,1,1,0,0,0,1,0,0,0,0,1,0,0 ; \ldots$ $0,0,1,1,0,0,0,1,0,1,0,0,1,0,1,0,0,0,1,0,0,0,1,1,0,0,1,0,0,1,0,0, \ldots$ $1,0,0,1,1,0,0,1,1,1,1,0,0,1,1,1,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1,0 ; \ldots$ $0,0,0,1,0,1,0,0,1,1,0,1,1,0,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1,1,0,0, \ldots$ $0,0,1,0,0,1,1,1,1,0,0,1,1,1,0,0,1,0,0,1,0,1,0,1,1,0,0,1,0,0,0,0 ; \ldots$ $0,0,1,1,1,1,1,0,0,0,0,1,1,0,1,1,0,1,0,1,0,0,0,0,0,1,1,0,1,1,1,1, \ldots$ $0,1,1,1,1,1,1,0,0,0,1,0,1,1,1,0,1,1,0,1,1,1,0,0,1,1,1,1,0,1,1,1 ; \ldots$ $1,1,0,1,0,0,0,0,0,1,0,0,1,0,1,1,0,1,0,1,1,0,0,0,0,0,0,1,1,1,1,1, \ldots$ $1,0,0,0,1,0,0,1,0,1,0,0,1,0,1,1,1,1,1,1,0,0,0,0,0,0,1,0,1,1,1,1 ; \ldots$ $1,1,0,0,0,1,0,0,0,1,0,1,0,0,0,0,1,1,0,0,1,0,0,0,0,1,0,1,1,1,0,1, \ldots$ $0,0,1,1,0,0,0,1,0,0,0,0,0,0,1,1,0,1,1,0,1,1,1,1,0,0,1,0,1,0,1,0 ; \ldots$ $0,0,0,0,1,1,0,0,0,1,0,0,0,0,0,0,1,0,1,1,0,1,0,0,1,0,1,1,1,0,0,1, \ldots$ $1,0,1,0,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,1,0,1,0,0,1,1,1,1,0,0,1 ; \ldots$ $0,0,0,0,0,0,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0, \ldots$ $0,0,0,1,0,0,0,1,0,1,0,0,1,1,1,1,0,0,1,0,1,0,1,1,0,1,1,0,1,0,1,0 ; \ldots$ $0,0,1,1,1,0,1,1,1,0,0,0,0,0,0,1,0,0,1,0,1,1,1,0,1,1,0,0,0,1,1,0, \ldots$ $1,0,1,0,1,1,1,1,0,1,1,0,1,1,1,0,0,1,1,1,1,0,1,0,0,1,0,1,0,0,1,1 ; \ldots$ $0,0,0,0,0,0,0,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,1,0,0,0,0,0,0,1,1, \ldots$ $0,0,0,0,1,0,0,1,1,0,1,0,0,1,1,1,0,0,0,0,0,1,0,1,1,1,1,1,0,1,0,0 ; \ldots$ $0,0,0,0,1,1,0,1,1,1,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,1,1,1,1,0, \ldots$ $0,0,1,1,0,1,1,1,0,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,1,0,0,0,0,0,1,0 ; \ldots$ $1,0,0,0,1,1,0,0,0,0,1,0,0,0,1,1,1,0,0,1,0,0,0,0,1,0,0,1,1,0,1,0, \ldots$ $1,1,1,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,0,1,1,0,1,0,0,1,0,1,0,1,0,1 ; \ldots$ $1,1,1,1,0,0,1,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,1,1,0,0,0,0,0,1,1,0, \ldots$ $0,0,0,1,0,1,1,1,0,1,0,1,0,1,0,0,1,0,1,1,0,1,0,1,1,1,0,0,1,1,1,1 ; \ldots$ $1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, \ldots$ $1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1 ; \ldots$ $0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$ $0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ; \ldots$ $0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, \ldots$


```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,...
0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0;\ldots
0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,...
0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots.
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0;\ldots
0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0;...
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\ldots.
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,\ldots.
];
% table of all 128 possible values of 8-bit CV shift register
cs = [...
255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 015,..
239, 238, 237, 236, 235, 234, 233, 023, 231, 230, 229, 027, 227, 029, 030, 031,\ldots..
223, 222, 221, 220, 219, 218, 217, 039, 215, 214, 213, 043, 211, 045, 046, 047,...
207, 206, 205, 051, 203, 053, 054, 055, 199, 057, 058, 059, 060, 061, 062, 063,\ldots..
191, 190, 189, 188, 187, 186, 185, 071, 183, 182, 181, 075, 179, 077, 078, 079,...
175, 174, 173, 083, 171, 085, 086, 087, 167, 089, 090, 091, 092, 093, 094, 095,\ldots..
159, 158, 157, 099, 155, 101, 102, 103, 151, 105, 106, 107, 108, 109, 110, 111,...
143, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127,\ldots
```

];
\% differentially decode the ciphertext prior to processing
ctd = abs(diff(ct));
\% set up initial conditions
solved $=0 ; \%$ no solution
outernotdone $=1$; \% loop flag
ii = 1; \% CV shift register table index
while(outernotdone)
\% get 8-bit CV shift register candidate from table \& convert to binary
$\mathrm{v}=\operatorname{dec} 2 \mathrm{bin}(\mathrm{cs}(\mathrm{ii}), 8)$;
cv8 = toascii(v)-48;
\% set up new blank solution
qqq $=$ zeros $(64,65)$; \% augmented solution matrix
rrr $=$ zeros $(64,1)$; \% vector of matrix rows that have been filled
innernotdone $=1$;
$\mathrm{nn}=32$; \% index of ciphertext bit
while(innernotdone)
\% form SRAM addresses
a1 $=\operatorname{ctd}(n n-8: n n-2) *\left[\begin{array}{lllll}64 & 32 & 16 & 4 & 2\end{array}\right]$ ';
$\mathrm{a} 2=\operatorname{ctd}(\mathrm{nn}-9: \mathrm{nn}-3) *\left[\begin{array}{lllll}64 & 32 & 16842\end{array}\right]$ ';
a3 $=\operatorname{ctd}(n n-10: n n-4) *\left[\begin{array}{llllll}64 & 32 & 16 & 4 & 2 & 1\end{array}\right]$;
$\mathrm{a} 4=\operatorname{ctd}(\mathrm{nn}-11: \mathrm{nn}-5) *\left[\begin{array}{llllll}64 & 32 & 16 & 8 & 4 & 2\end{array}\right]$ ';
a5 $=\operatorname{ctd}(n n-12: n n-6) *\left[\begin{array}{llllll}64 & 32 & 16 & 4 & 2\end{array}\right]$ ';
a6 $=\operatorname{ctd}(n n-13: n n-7) *\left[\begin{array}{llllll}64 & 32 & 16 & 8 & 4 & 2\end{array}\right]$ ';
a7 $=\operatorname{ctd}(n n-14: n n-8) *\left[\begin{array}{llllll}64 & 32 & 16 & 8 & 2 & 1\end{array}\right]$;
$\mathrm{a} 8=\operatorname{ctd}(\mathrm{nn}-15: n n-9) *\left[\begin{array}{llllll}64 & 32 & 16 & 8 & 4 & 2\end{array}\right]$ ';
a9 $=\operatorname{ctd}(n n-16: n n-10) *\left[\begin{array}{lll}64 & 16 & 8 \\ 4 & 2 & 1\end{array}\right]$;
\% get SRAM content vectors
$\mathrm{v} 1=\operatorname{sr}(\mathrm{a} 1+1,:$ );
$\mathrm{v} 2=\mathrm{sr}(\mathrm{a} 2+1,: \mathrm{s})$;
$\mathrm{v} 3=\operatorname{sr}(\mathrm{a} 3+1,:$ );
$\mathrm{v} 4=\operatorname{sr}(\mathrm{a} 4+1,:$ );
v5 = sr(a5+1,:);
$\mathrm{v} 6=\operatorname{sr}(\mathrm{a} 6+1,:$ );
v7 $=\operatorname{sr}(\mathrm{a} 7+1,:$ );
$\mathrm{v} 8=\mathrm{sr}(\mathrm{a} 8+1,:$ );
v9 = sr(a9+1,:);
\% build left side of linear equation describing this ciphertext bit
vv = zeros (1,64);
$\mathrm{vv}=$ bitxor(vv,bitand(cv8(1),v1));
vv = bitxor(vv,bitand(cv8(2),v2));
$\mathrm{vv}=$ bitxor(vv,bitand(cv8(3),v3));
vv = bitxor(vv,bitand(cv8(4),v4));
$\mathrm{vv}=$ bitxor(vv,bitand(cv8(5),v5));
vv = bitxor(vv,bitand(cv8(6),v6));
vv = bitxor(vv,bitand(cv8(7),v7));
vv = bitxor(vv,bitand(cv8(8),v8));
vv = bitxor(vv,v9);
\% build right side of linear equation describing this ciphertext bit
\% i.e. calculate resulting keystream bit (assuming plaintext was 0 )
$\mathrm{c}=\operatorname{ctd}(\mathrm{nn})$; \% (if plaintext was NOT 0 , XOR the known plaintext bit here)
$\mathrm{c}=\mathrm{bitxor}(\mathrm{c}, \mathrm{bitand}(\mathrm{cv} 8(1), \mathrm{ctd}(\mathrm{nn}-22)))$;
$\mathrm{c}=\mathrm{bitxor}(\mathrm{c}, \mathrm{bitand}(\mathrm{cv} 8(2), \operatorname{ctd}(\mathrm{nn}-23)))$;
c = bitxor(c,bitand(cv8(3),ctd(nn-24)));
$\mathrm{c}=\mathrm{bitxor}(\mathrm{c}, \mathrm{bitand}(\mathrm{cv} 8(4), \mathrm{ctd}(\mathrm{nn}-25)))$;
c = bitxor(c,bitand(cv8(5),ctd(nn-26)));
$\mathrm{c}=\mathrm{bitxor}(\mathrm{c}, \mathrm{bitand}(\mathrm{cv} 8(6), \mathrm{ctd}(\mathrm{nn}-27)))$;

```
    c = bitxor(c,bitand(cv8(7),ctd(nn-28)));
    c = bitxor(c,bitand(cv8(8),ctd(nn-29)));
    c = bitxor(c,ctd(nn-30));
    % build the overall equation
    qq = [vv,c];
    notdone = 1;
    % put the linear equation into the augmented solution matrix
    while(notdone)
        cc = find(qq==1,1); % column of leftmost 1 in candidate
        if(cc==65) % this is an inconsistent solution
            ii = ii + 1; % try next CV shift register value
            % break out of inner loop
            innernotdone = 0;
            break;
        end
        if(!isempty(cc)) % vector was not all Os
            if(rrr(cc)) % this row already exists in the matrix
                qq = bitxor(qq, qqq(cc,:));
            else % this row does not exist yet
                qqq(cc,:) = qq; % save it in the matrix
                rrr(cc) = 1; % show this row now full
                notdone = 0; % break out of vector loop
            end
            else % vector was all 0s and is thus of no use to us
                notdone = 0; % break out of vector loop
            end
        end
        if(sum(rrr)==64) % have we filled the entire solution matrix
            innernotdone = 0; % we are done
            outernotdone = 0;
            solved = 1; % we have a solution
        end
        nn = nn + 1; % move on to next ciphertext bit
        if(nn>length(ctd)) % have we run out of ciphertext
            innernotdone = 0; % we are done
            outernotdone = 0;
        end
    end
end
if(solved) % we have a valid & complete solution matrix
    % put the solution matrix in reduced row echelon form
    for ii = 64:-1:2 % work up from bottom row to top row
        qq = qqq(ii,:); % get row
        for jj = ii-1:-1:1 % check all rows above this one
            if(qqq(jj,ii)==1) % target row has a 1 in our column
                    qqq(jj,:) = bitxor(qqq(jj,:),qq); % XOR it out of there
            end
        end
    end
    % form & print the final answer
    % right-most column of solution matrix holds CV bits 70.69.68.67...10.09.08.05
    % cv8 holds CV bits X.01.00.04.03.02.07.06, bits inverted if bin2dec(cv8) > 127
    tmp = qqq(:,65)';
    if(cv8*[128 64 32 16 8 4 2 1]'>127) % invert cv8 if necessary
        cv8 = bitxor(cv8,ones(1,8));
    end
    cv = [tmp(1:63) cv8(7:8) tmp(64) cv8(4:6) cv8(2:3)]; % recovered CV
```

printf("recovery complete\n");
printf("CV = ");
printf("\%d", cv);
printf("\n");
else
printf("full solution not found -- try longer ciphertext input\n");
end

## E DVP Key Loader Firmware Disassembly



| F030 : 44 | "D" | lsra |  |  |
| :---: | :---: | :---: | :---: | :---: |
| F031 : 7C 0066 | " \\| f" | inc | \$0066 |  |
| F034 : D6 66 | " f" | ldab | \$0066 |  |
| F036 : C1 10 | " " | cmpb | \#\$10 |  |
| F038 : 26 EB | "\& " | bne | \$F025 |  |
| F03A : 16 | " " | tab |  |  |
| F03B : 48 | "H" | asla |  |  |
| F03C : 54 | "T" | lsrb |  |  |
| F03D : 54 | "T" | lsrb |  |  |
| F03E : 84 C0 | " | anda | \#\$C0 |  |
| F040 : C4 06 | " | andb | \#\$06 |  |
| F042 : 9767 | " g" | staa | \$0067 |  |
| F044 : D7 74 | " t" | stab | \$0074 |  |
| F046 : C6 07 | " | 1 dab | \#\$07 |  |
| F048 : D7 71 | " q" | stab | \$0071 |  |
| F04A : 9667 | " g" | ldaa | \$0067 |  |
| F04C : 2B 06 | "+ " | bmi | \$F054 |  |
| F04E : 8619 |  | ldaa | \#\$19 |  |
| F050 : C6 08 |  | 1 dab | \#\$08 |  |
| F052 : 2010 | " " | bra | \$F064 |  |
| F054 : C6 0F | " | 1 dab | \#\$0F |  |
| F056 : D7 71 | " q" | stab | \$0071 |  |
| F058 : C6 06 | " " | 1 dab | \#\$06 |  |
| F05A : 8440 | " @" | anda | \#\$40 |  |
| F05C : 2704 | "' | beq | \$F062 |  |
| F05E : 86 0F |  | ldaa | \#\$0F |  |
| F060 : 2002 |  | bra | \$F064 |  |
| F062 : 8611 |  | ldaa | \#\$11 |  |
| F064 : $97{ }^{\circ}$ 6F | " o" | staa ${ }^{\text {b }}$ | \$006F |  |
| F066 : D7 70 | " p" | stab | \$0070 |  |
| F068 : BD FE 25 | " \%" | jsr | \$FE25 |  |
| F06B : F6 01 E2 | " " | ldab | \$01E2 |  |
| F06E : 58 | "X" | aslb |  |  |
| F06F : 58 | "X" | aslb |  |  |
| F070 : 58 | "X" | aslb |  |  |
| F071 : 58 | "X" | aslb |  |  |
| F072 : D7 75 | " u" | stab | \$0075 |  |
| F074 : F6 01 E1 | " " | 1 dab | \$01E1 |  |
| F077 : C4 0F |  | andb | \#\$0F |  |
| F079 : DA 75 | " u" | orab | \$0075 |  |
| F07B : 10 | " " | sba |  |  |
| F07C : 26 3A | "\&:" | bne | \$F0B8 |  |
| F07E : B6 01 FD | " | ldaa | \$01FD |  |
| F081 : 840 F |  | anda | \#\$0F |  |
| F083 : 81 OA |  | cmpa | \#\$0A |  |
| F085 : 2631 | "\&1" | bne | \$F0B8 |  |
| F087 : C6 81 |  | ldab | \#\$81 |  |
| F089 : D7 68 | " h" | stab | \$0068 |  |
| F08B : BD FE 1C | " " | jsr | \$FE1C |  |
| F08E : 9680 |  | ldaa | PA |  |
| F090 : 8580 |  | bita | \#\$80 |  |
| F092 : 261 F | "\& " | bne | \$F0B3 |  |
| F094 : 7C 00 6B | " \\| k" | inc | \$006B |  |
| F097 : CE FF 8E | " " | 1 dx | \#\$FF8E | All Erased |




| $\begin{aligned} & \text { F166 : 5C } \\ & \text { F167 : } 20 \text { F9 } \end{aligned}$ | "\" | incb <br> bra | \$F162 |  |
| :---: | :---: | :---: | :---: | :---: |
| F169 : D6 66 | " f" | 1 dab | \$0066 |  |
| F16B : C1 80 | " | cmpb | \#\$80 |  |
| F16D : 26 E7 | "\& " | bne | \$F156 |  |
| F16F : 39 | "9" | rts |  |  |
| F170 : CE 0051 | " Q" | 1 dx | \#\$0051 | write buffer 51.59 to display |
| F173 : 8667 | " g" | ldaa | \#\$67 |  |
| F175 : 9780 |  | staa | PA | enable display |
| F177 : C6 FD |  | 1 dab | \#\$FD |  |
| F179 : D7 82 | " " | stab | PB |  |
| F17B : C6 47 | " G" | ldab | \#\$47 |  |
| F17D : D7 80 |  | stab | PA | drop dsp write |
| F17F : 9780 |  | staa | PA | raise dsp write |
| . . . . . | - | - | - . |  |
| F181 : A6 00 | " | ldaa | \$00, x | get byte from table |
| F183 : 8407 |  | anda | \#\$07 |  |
| F185 : 8A 20 |  | oraa | \#\$20 |  |
| F187 : 9780 |  | staa | PA | save to display |
| F189 : E6 00 | " " | 1 dab | \$00, x | get same byte |
| F18B : 54 | "T" | 1 srb |  | move as required |
| F18C : C4 FD |  | andb | \#\$FD |  |
| F18E : CA 01 |  | orab | \#\$01 |  |
| F190 : D7 82 |  | stab | PB | save to display |
| F192 : 84 CF |  | anda | \#\$CF |  |
| F194 : 9780 | " | staa | PA |  |
| F196 : 8A 20 | " | oraa | \#\$20 | display write |
| F198 : 9780 | " " | staa | PA |  |
| F19A : 08 | " " | inx |  |  |
| F19B : 8C 0059 | " Y" | cpx | \#\$0059 | done with 8 chars? |
| F19E : 26 E 1 | "\& " | bne | \$F181 |  |
| F1A0 : D7 4C | " L" | stab | \$004C | 4C <- last char |
| F1A2 : 86 E3 | " " | ldaa | \#\$E3 | a <- 227 |
| F1A4 : D6 67 | " g" | 1 dab | \$0067 | b <- (67) |
| F1A6 : 2A 02 | "* " | bpl | \$F1AA |  |
| F1A8 : 84 FD | " " | anda | \#\$FD |  |
| . | - $\cdot$ | - | - . . |  |
| F1AA : 974 B | " K" | staa | \$004B |  |
| F1AC : 9780 | " | staa | PA |  |
| F1AE : 39 | "9" | rts |  |  |
| F1AF : CE FF 7E | " ~" | 1 dx | \#\$FF7E | numbers table |
| F1B2 : 4D | "M" | tsta |  | a is octal digit |
| F1B3 : 2704 | "' | beq | \$F1B9 |  |
| F1B5 : 08 | " " | inx |  | inc table index until digit found |
| F1B6 : 4A | "J" | deca |  |  |
| F1B7 : 20 F9 | " | bra | \$F1B2 |  |
| F1B9 : A6 00 | " | 1 daa | \$00, x | we have correct index of digit char |
| F1BB : DE 63 | " c" | 1 dx | \$0063 |  |
| F1BD : A7 00 | " " | staa | \$00, x | save digit in display buffer |
| F1BF : 39 | "9" | rts |  |  |
| F1C0 : 9F 63 | " c" | sts | \$0063 | load characters from table to 51.59 |


| F1C2 : 35 | "5" | txs |  |  |
| :---: | :---: | :---: | :---: | :---: |
| F1C3 : CE 0051 | " Q" | 1 dx | \#\$0051 |  |
| . . . . . | . |  | . |  |
| F1C6 : 33 | "3" | pulb |  |  |
| F1C7 : E7 00 | " | stab | \$00, x |  |
| F1C9 : 08 | " | inx |  |  |
| F1CA : 8C 0059 | " Y" | cpx | \#\$0059 |  |
| F1CD : $26 \mathrm{F7}$ | "\& " | bne | \$F1C6 |  |
| F1CF : 9E 63 | " c" | lds | \$0063 |  |
| F1D1 : 39 | "9" | rts |  |  |
| F1D2 : B6 01 C8 | " | ldaa | \$01C8 | a <- key\# |
| F1D5 : 8408 | " | anda | \#\$08 |  |
| F1D7 : 2712 | "' " | beq | \$F1EB |  |
| F1D9 : CE FF B3 | " " | 1 dx | \#\$FFB3 | Loc? |
| F1DC : DF 61 | " a" | stx | \$0061 |  |
| F1DE : BD F1 C0 | " | jsr | \$F1C0 | put table chars in buffer |
| F1E1 : 8680 |  | ldaa | \#\$80 |  |
| F1E3 : 9756 | " V" | staa | \$0056 | $56<-128$ |
| F1E5 : 8601 |  | ldaa | \#\$01 |  |
| F1E7 : 9772 | " r" | staa | \$0072 | $72<-1$ |
| F1E9 : 2023 | " \#" | bra | \$F20E |  |
| F1EB : BD FE 51 | " Q" | jsr | \$FE51 | $\mathrm{x}<-1 \mathrm{c} 0.1 \mathrm{c} 7$ based on key\# |
| F1EE : A6 00 |  | ldaa | \$00, x |  |
| F1F0 : 2705 | "' " | beq | \$F1F7 |  |
| F1F2 : CE FF 96 | " " | $1 d x$ | \#\$FF96 | ready |
| F1F5 : 2003 | " " | bra | \$F1FA |  |
| F1F7 : CE FF 8E | " | 1 dx | \#\$FF8E | All Erased |
| F1FA : BD F1 C0 | " | jsr | \$F1C0 | put table chars in buffer |
| F1FD : 8680 | " " | ldaa | \#\$80 |  |
| F1FF : 9751 | " Q" | staa | \$0051 |  |
| F201 : B6 01 FE | " | ldaa | \$01FE | get a byte from RAM |
| F204 : 8407 |  | anda | \#\$07 | keep only octal digit |
| F206 : CE 0052 | " R" | $1 d x$ | \#\$0052 | point to display buffer |
| F209 : DF 63 | " c" | stx | \$0063 |  |
| F20B : BD F1 AF | " | jsr | \$F1AF | convert oct to char |
| - . . $\cdot$ |  |  |  |  |
| F20E : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| F211 : 39 | "9" | rts |  |  |
| F212 : 9668 | " h" | ldaa | \$0068 |  |
| F214 : 2B 34 | "+4" | bmi | \$F24A |  |
| F216 : D6 6B | " k" | ldab | \$006B |  |
| F218 : C1 04 |  | cmpb | \#\$04 |  |
| F21A : 27 2A | "'*" | beq | \$F246 |  |
| F21C : F6 01 C8 | " " | ldab | \$01C8 |  |
| F21F : C4 08 |  | andb | \#\$08 |  |
| F221 : 2623 | "\&\#" | bne | \$F246 |  |
| F223 : D6 6B | " k" | ldab | \$006B |  |
| F225 : 2713 |  | beq | \$F23A |  |
| F227 : C1 03 |  | cmpb | \#\$03 |  |
| F229 : 27 4A | "'J" | beq | \$F275 |  |
| F22B : 39 | "9" | rts |  |  |


| F22C : CE FF 9D | " | 1 dx | \#\$FF9D | Load _ |
| :---: | :---: | :---: | :---: | :---: |
| F22F : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| F232 : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| F235 : C6 07 |  | ldab | \#\$07 |  |
| F237 : D7 69 | " i" | stab | \$0069 |  |
| F239 : 39 | "9" | rts |  |  |
| F23A : 8508 | " " | bita | \#\$08 |  |
| F23C : 2701 | "' | beq | \$F23F |  |
| F23E : 39 | "9" | rts |  |  |
| F23F : B7 01 FE | " | staa | \$01FE |  |
| F242 : BD F1 D2 |  | jsr | \$F1D2 |  |
| F245 : 39 | "9" | rts |  |  |
| F246 : BD FD AF |  | jsr | \$FDAF |  |
| F249 : 39 | "9" | rts |  |  |
| F24A : CE F2 5D | " ] " | 1 dx | \#\$F25D |  |
| F24D : 847 F | " | anda | \#\$7F |  |
| - . . . | - • | - | - • |  |
| F24F : 2706 | "' | beq | \$F257 |  |
| F251 : 08 | " " | inx |  |  |
| F252 : 08 | " " | inx |  |  |
| F253 : 08 | " " | inx |  |  |
| F254 : 4A | "J" | deca |  |  |
| F255 : 20 F8 | " " | bra | \$F24F |  |
| F257 : AD 00 | " " | jsr | \$00, x |  |
| F259 : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| F25C : 39 | "9" | rts |  |  |
| F25D : 7E F3 B5 | "~ | jmp | \$F3B5 |  |
| F260 : 7E F4 65 | "~ e" | jmp | \$F465 |  |
| F263 : 7E F4 C5 | "~ | jmp | \$F4C5 |  |
| F266 : 7E F3 57 | "~ W" | jmp | \$F357 |  |
| F269 : 7E F5 7B | "~ \{" | jmp | \$F57B |  |
| F26C : 7E F5 9E | "~ | jmp | \$F59E | !keyload! |
| F26F : 7E F3 68 | "~ h" | jmp | \$F368 |  |
| F272 : 7E F3 7C | "~ \\| | jmp | \$F37C |  |
| F275 : 96 6E | " n" | ldaa | \$006E |  |
| F277 : 261 A | "\& " | bne | \$F293 |  |
| F279 : 9668 | " h" | ldaa | \$0068 |  |
| F27B : 8508 |  | bita | \#\$08 |  |
| F27D : 2701 | "' | beq | \$F280 |  |
| F27F : 39 | "9" | rts |  |  |
| F280 : 7F 0000 | " " | clr | \$0000 |  |
| F283 : 7F 0001 | " " | clr | \$0001 |  |




| F35A : 8408 | " | anda | \#\$08 |
| :---: | :---: | :---: | :---: |
| F35C : 9A 6B | " k" | oraa | \$006B |
| F35E : 2607 | "\& " | bne | \$F367 |
| F360 : 8603 | " " | ldaa | \#\$03 |
| F362 : 976 B | " k" | staa | \$006B |
| F364 : BD F2 2C | " , " | jsr | \$F22C |
| . . . . . |  |  |  |
| F367 : 39 | "9" | rts |  |
| F368 : 4F | "0" | clra |  |
| F369 : 974 F | " 0" | staa | \$004F |
| F36B : 9750 | " P" | staa | \$0050 |
| F36D : 9668 | " h" | ldaa | \$0068 |
| F36F : 8187 |  | cmpa | \#\$87 |
| F371 : 2708 | "' | beq | \$F37B |
| F373 : B6 01 C8 | " " | ldaa | \$01C8 |
| F376 : 84 FD |  | anda | \#\$FD |
| F378 : B7 01 C8 | " " | staa | \$01C8 |
| . . . . . | - • | . |  |
| F37B : 39 | "9" | rts |  |
| F37C : B6 01 C8 | " | ldaa | \$01C8 |
| F37F : 8408 | " | anda | \#\$08 |
| F381 : 9A 6B | " k" | oraa | \$006B |
| F383 : 2701 | "' | beq | \$F386 |
| F385 : 39 | "9" | rts |  |
| F386 : B6 01 C8 |  | 1 daa | \$01C8 |
| F389 : 8404 |  | anda | \#\$04 |
| F38B : 2709 |  | beq | \$F396 |
| F38D : 8606 |  | ldaa | \#\$06 |
| F38F : B7 01 C8 | " " | staa | \$01C8 |
| F392 : BD F1 D2 | " " | jsr | \$F1D2 |
| F395 : 39 | "9" | rts |  |
| F396 : 4F | "0" | clra |  |
| F397 : CE 01 C0 | " " | 1 dx | \#\$01C0 |
| . . . . . |  |  |  |
| F39A : AA 00 |  | oraa | \$00,x |
| F39C : 840 O |  | anda | \#\$0F |
| F39E : 8C 01 C8 | " " | cpx | \#\$01C8 |
| F3A1 : 2703 | "' | beq | \$F3A6 |
| F3A3 : 08 | " | inx |  |
| F3A4 : 20 F4 | " " | bra | \$F39A |
| F3A6 : 4D | "M" | tsta |  |
| F3A7 : 2701 | "' | beq | \$F3AA |
| F3A9 : 39 | "9" | rts |  |
| F3AA : 7F 0072 | " r" | clr | \$0072 |
| F3AD : 8604 |  | ldaa | \#\$04 |
| F3AF : 976 B | " k" | staa | \$006B |
| F3B1 : BD FD AF | " " | jsr | \$FDAF |
| F3B4 : 39 | "9" | rts |  |
| F3B5 : 96 6B | " k" | ldaa | \$006B |
| F3B7 : 27 2B | "'+" | beq | \$F3E4 |


| F3B9 : 8104 | " " | cmpa | \#\$04 |  |
| :---: | :---: | :---: | :---: | :---: |
| F3BB : 274 E | "'N" | beq | \$F40B |  |
| F3BD : 8103 | " " | cmpa | \#\$03 |  |
| F3BF : 2705 |  | beq | \$F3C6 |  |
| F3C1 : 8101 | " | cmpa | \#\$01 |  |
| F3C3 : 2722 | ")"" | beq | \$F3E7 |  |
| F3C5 : 39 | "9" | rts |  |  |
| F3C6 : 96 6E | " n" | Idaa | \$006E |  |
| F3C8 : 2601 | "\& " | bne | \$F3CB |  |
| F3CA : 39 | "9" | rts |  |  |
| F3CB : 916 F | " ○" | cmpa | \$006F |  |
| F3CD : 271 F | "' " | beq | \$F3EE |  |
| F3CF : BD FE 51 | " Q" | jsr | \$FE51 | $\mathrm{x}<-1 \mathrm{c} 0.1 \mathrm{c} 7$ based on key\# |
| F3D2 : 6F 00 | "○ " | clr | \$00, x |  |
| F3D4 : BD FE E7 | " | jsr | \$FEE7 | 63.64 <- ram addr of 24-digit key |
| F3D7 : DE 63 | " c" | 1 dx | \$0063 |  |
| F3D9 : 4F | "0" | clra |  |  |
| . . . . | - | . | . |  |
| F3DA : 6F 00 | "0 " | clr | \$00, x |  |
| F3DC : 4C | "L" | inca |  |  |
| F3DD : 08 | " " | inx |  |  |
| F3DE : 8118 | " " | cmpa | \#\$18 |  |
| F3E0 : 26 F8 | "\& " | bne | \$F3DA |  |
| F3E2 : 2020 | " " | bra | \$F404 |  |
| F3E4 : 7E F4 30 | "~ 0 " | jmp | \$F430 |  |
| F3E7 : BD FF 4B | " K" | jsr | \$FF4B |  |
| F3EA : 7F 00 6B | " k" | clr | \$006B |  |
| F3ED : 39 | "9" | rts |  |  |
| F3EE : BD FE E7 | " | jsr | \$FEE7 | 63.64 <- ram addr of 24-digit key |
| F3F1 : 8618 | " " | ldaa | \#\$18 |  |
| F3F3 : CE 0002 | " | ldx | \#\$0002 |  |
| F3F6 : DF 61 | " a" | stx | \$0061 |  |
| F3F8 : DE 63 | " c" | ldx | \$0063 |  |
| F3FA : BD FF 3C | " < " | jsr | \$FF3C |  |
| F3FD : BD FE 51 | " Q" | jsr | \$FE51 | $\mathrm{x}<-1 \mathrm{c} 0.1 \mathrm{c} 7$ based on key\# |
| F400 : 6F 00 | "○ " | clr | \$00, x |  |
| F402 : 6C 00 | "1 " | inc | \$00, x |  |
| . | - • | - | . |  |
| F404 : BD FF 34 | " 4" | jsr | \$FF34 |  |
| F407 : BD F1 D2 | " | jsr | \$F1D2 |  |
| F40A : 39 | "9" | rts |  |  |
| F40B : 9672 | " r" | Idaa | \$0072 |  |
| F40D : 8407 | " " | anda | \#\$07 |  |
| F40F : 8101 | " " | cmpa | \#\$01 |  |
| F411 : 2201 | "" | bhi | \$F414 |  |
| F413 : 39 | "9" | rts |  |  |
| F414 : 8606 | " " | Idaa | \#\$06 |  |
| F416 : CE 0002 | " | ldx | \#\$0002 |  |
| F419 : DF 61 | " a" | stx | \$0061 |  |
| F41B : CE 01 C9 | " " | ldx | \#\$01C9 |  |



| F48B : 39 | "9" | rts |  |  |
| :---: | :---: | :---: | :---: | :---: |
| F48C : 96 6E | " n" | ldaa | \$006E |  |
| F48E : 2602 | "\& " | bne | \$F492 |  |
| F490 : 20 E6 | " " | bra | \$F478 |  |
| F492 : 9669 | " i" | ldaa | \$0069 |  |
| F494 : 8102 |  | cmpa | \#\$02 |  |
| F496 : 2607 | "\& " | bne | \$F49F |  |
| F498 : 7F 00 6E | " n" | clr | \$006E |  |
| F49B : BD F2 2C | " , " | jsr | \$F22C |  |
| F49E : 39 | "9" | rts |  |  |
| F49F : D6 69 | " i" | 1 dab | \$0069 |  |
| F4A1 : C0 02 | " " | subb | \#\$02 |  |
| F4A3 : 96 6E | " n" | ldaa | \$006E |  |
| F4A5 : 10 | " " | sba |  |  |
| F4A6 : 976 E | " n" | staa | \$006E |  |
| F4A8 : BD FF 65 | " e" | jsr | \$FF65 |  |
| F4AB : CE 0053 | " S" | 1 dx | \#\$0053 |  |
| F4AE : 8680 | " " | ldaa | \#\$80 |  |
| . . . . . | - | - | . . |  |
| F4B0 : A7 00 | " " | staa | \$00, x |  |
| F4B2 : 8C 0058 | " X" | cpx | \#\$0058 |  |
| F4B5 : 2703 | "' | beq | \$F4BA |  |
| F4B7 : 08 | " " | inx |  |  |
| F4B8 : 20 F6 |  | bra | \$F4B0 |  |
| F4BA : 4C | "L" | inca |  |  |
| F4BB : 9753 | " S" | staa | \$0053 |  |
| F4BD : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| F4C0 : 8602 | " " | ldaa | \#\$02 |  |
| F4C2 : 9769 | " i" | staa | \$0069 |  |
| F4C4 : 39 | "9" | rts |  |  |
| F4C5 : B6 01 C8 | " | 1 daa | \$01C8 |  |
| F4C8 : 8408 | " " | anda | \#\$08 |  |
| F4CA : 2701 | "' " | beq | \$F4CD |  |
| F4CC : 39 | "9" | rts |  |  |
| F4CD : 966 B | " k" | 1 daa | \$006B |  |
| F4CF : 8104 | " | cmpa | \#\$04 |  |
| F4D1 : 2601 | "\& " | bne | \$F4D4 |  |
| F4D3 : 39 | "9" | rts |  |  |
| F4D4 : 8103 | " " | cmpa | \#\$03 |  |
| F4D6 : 2712 | "' | beq | \$F4EA |  |
| F4D8 : 8101 |  | cmpa | \#\$01 |  |
| F4DA : 2601 | "\& " | bne | \$F4DD |  |
| F4DC : 39 | "9" | rts |  |  |
| F4DD : B6 01 FE | " | ldaa | \$01FE |  |
| F4E0 : 4C | "L" | inca |  |  |
| F4E1 : 8407 |  | anda | \#\$07 |  |
| F4E3 : B7 01 FE | " " | staa | \$01FE |  |
| F4E6 : BD F1 D2 | " " | jsr | \$F1D2 |  |
| F4E9 : 39 | "9" | rts |  |  |



| F553 : 39 | "9" | rts |  |  |
| :---: | :---: | :---: | :---: | :---: |
| F554 : D6 67 | " g" | ldab | \$0067 |  |
| F556 : C4 C0 | " " | andb | \#\$C0 |  |
| F558 : C1 C0 | " " | cmpb | \#\$C0 |  |
| F55A : 26 EE | "\& " | bne | \$F54A |  |
| F55C : 8606 | " | ldaa | \#\$06 |  |
| F55E : 9769 | " i" | staa | \$0069 |  |
| F560 : D6 6A | " j" | ldab | \$006A |  |
| F562 : C1 04 | " " | cmpb | \#\$04 |  |
| F564 : 26 E4 | "\& " | bne | \$F54A |  |
| F566 : 8604 | " " | ldaa | \#\$04 |  |
| F568 : 9769 | " i" | staa | \$0069 |  |
| F56A : C6 80 | " " | ldab | \#\$80 |  |
| F56C : D7 55 | " U" | stab | \$0055 |  |
| F56E : D7 56 | " V" | stab | \$0056 |  |
| F570 : DE 4D | " M" | 1 dx | \$004D |  |
| F572 : 09 | " | dex |  |  |
| F573 : 09 | " " | dex |  |  |
| F574 : DF 4D | " M" | stx | \$004D |  |
| F576 : CE 0054 | " T" | 1 dx | \#\$0054 |  |
| F579 : 20 CF | " " | bra | \$F54A |  |
| F57B : B6 01 C8 | " " | ldaa | \$01C8 |  |
| F57E : 8502 | " | bita | \#\$02 |  |
| F580 : 2705 | "' | beq | \$F587 |  |
| F582 : 86 OC |  | ldaa | \#\$0C |  |
| F584 : B7 01 C8 | " " | staa | \$01C8 |  |
| . . . . . | . $\cdot$ | - | . |  |
| F587 : B6 01 DF |  | ldaa | \$01DF |  |
| F58A : 840 F | " | anda | \#\$0F |  |
| F58C : 81 OF | " " | cmpa | \#\$0F |  |
| F58E : 2703 | "' | beq | \$F593 |  |
| F590 : 7F 01 DF | " " | clr | \$01DF |  |
| - . . . | - $\cdot$ | - $\cdot$ | - |  |
| F593 : BD FE 25 | " \%" | jsr | \$FE25 |  |
| F596 : BD FE 46 | " F" | jsr | \$FE46 |  |
| F599 : 8637 | " 7" | ldaa | \#\$37 |  |
| F59B : 9783 |  | staa | CRB |  |
| F59D : 39 | "9" | rts |  |  |
| F59E : B6 01 C8 | " | ldaa | \$01C8 | !start of keyload! get key\# |
| F5A1 : 8408 |  | anda | \#\$08 |  |
| F5A3 : 9A 6B | " k" | oraa | \$006B |  |
| F5A5 : 2701 | "' " | beq | \$F5A8 | check for valid key |
| F5A7 : 39 | "9" | rts |  |  |
| F5A8 : BD FE 51 | " Q" | jsr | \$FE51 | $\mathrm{x}<-1 \mathrm{CO} .1 \mathrm{C} 7$ based on key\# |
| F5AB : A6 00 |  | ldaa | \$00, x |  |
| F5AD : 2601 | "\& " | bne | \$F5B0 | ?another valid key check? |
| F5AF : 39 | "9" | rts |  |  |
| F5B0 : BD F1 D2 | " " | jsr | \$F1D2 | display something |
| F5B3 : BD FE E7 | " " | jsr | \$FEE7 | 63.64 <- ram addr of 24-digit key |
| F5B6 : 9667 | " g" | ldaa | \$0067 |  |
| F5B8 : 2A 03 | "* " | bpl | \$F5BD | another check of some sort |
| F5BA : 7E F9 1A | "~ " | jmp | \$F91A |  |


| F5BD : 7F 0000 | " " | clr | \$0000 | clear parity |
| :---: | :---: | :---: | :---: | :---: |
| F5C0 : 8615 |  | ldaa | \#\$15 |  |
| F5C2 : 9766 | " $\mathrm{f}^{\prime}$ | staa | \$0066 | 66 <- 21 |
| F5C4 : DE 63 | " c" | 1 dx | \$0063 | 63.64 holds key address 24 oct digits |
| F5C6 : BD F7 0A | " | jsr | \$F70A | get 1 digit of key \& put in buffer |
| F5C9 : 08 | " | inx |  |  |
| F5CA : 7A 0066 | "z f" | dec | \$0066 |  |
| F5CD : 26 F7 | "\& " | bne | \$F5C6 | put 1st 21 key digs (63 bits) in buffer |
| F5CF : C6 01 |  | ldab | \#\$01 |  |
| F5D1 : BD F7 OC | " " | jsr | \$F70C | put 64th key bit in 51.58 (1b) |
| F5D4 : BD F7 1E | " " | jsr | \$F71E | put next 2 bits of 22nd dig in 65 (2b) |
| F5D7 : BD F7 1E | " " | jsr | \$F71E |  |
| F5DA : BD F7 1B | " " | jsr | \$F71B | put 23rd oct key digit in 65 (3b) |
| F5DD : BD F7 1E | " " | jsr | \$F71E |  |
| F5E0 : BD F7 1E | " " | jsr | \$F71E |  |
| F5E3 : BD F7 1B | " " | jsr | \$F71B | put 24th oct key digit in 65 (2b) |
| F5E6 : BD F7 1E | " " | jsr | \$F71E | 65 now has 7 last bits of key |
| F5E9 : 740065 | "t e" | lsr | \$0065 | 0 fill at msb / b has 1s count 0.7 |
| F5EC : C1 03 |  | cmpb | \#\$03 | more than 3 of 71 s ? |
| F5EE : 2E 03 | ". " | bgt | \$F5F3 |  |
| F5F0 : 730065 | "s e" | com | \$0065 | fewer than 3 1s, invert all 8 bits |
| F5F3 : C6 40 | " @" | ldab | \#\$40 | b <- 64 to clear out 59.60 |
| F5F5 : BD F7 3F | " ? " | jsr | \$F73F | put 0 bit in 59.60 (clear parity reg) |
| F5F8 : 5A | "Z" | decb |  |  |
| F5F9 : 26 FA | "\& " | bne | \$F5F5 |  |
| F5FB : 9F 61 | " a" | sts | \$0061 | save present sp |
| F5FD : C6 40 | " @" | ldab | \#\$40 | count 64 bits |
| F5FF : 9E 61 | " a" | lds | \$0061 | get saved sp |
| F601 : 9658 | " X" | ldaa | \$0058 | get oldest bit from 51.58 |
| F603 : 46 | "F" | rora |  | save as newest (rotate entire reg) |
| F604 : BD F7 26 | " \& " | jsr | \$F726 | put key bit in 51.58 |
| F607 : 49 | "I" | rola |  | key bit in lsb of a |
| F608 : BD F7 3F | " ?" | jsr | \$F73F | put 0 in 59.60 \& get oldest bit out |
| F60B : 8900 |  | adca | \#\$00 | we are encoding key bits here |
| F60D : 46 | "F" | rora |  | c <- k+p |
| F60E : 2411 | "\$ " | bcc | \$F621 | $\mathrm{k}+\mathrm{p}$ was 0 so shift only |
| F610 : 8E F9 07 | " " | lds | \#\$F907 | tab:52009BF1ED40EAC2 (bit was 1) |
| F613 : CE 0059 | " Y" | 1 dx | \#\$0059 |  |
| . . . . . . | - $\cdot$ |  | - . |  |
| F616 : 32 | "2" | pula |  | get byte from table |
| F617 : A8 00 |  | eora | \$00, x | xor into 59.60 |
| F619 : A7 00 |  | staa | \$00, x | put taps from table into parity reg |
| F61B : 08 | " " | inx |  | next byte |
| F61C : 8C 0061 | " a" | cpx | \#\$0061 | do 8 bytes (59.60) |
| F61F : 26 F5 | "\& " | bne | \$F616 |  |
| F621 : 5A | "Z" | decb |  | dec bit cnt |
| F622 : 26 DB | "\& " | bne | \$F5FF | encode all 64 bits of key |
| F624 : D6 00 |  | ldab | \$0000 | get parity bit |
| F626 : C4 80 |  | andb | \#\$80 |  |
| F628 : DA 59 | " Y" | orab | \$0059 |  |
| F62A : D7 59 | " Y" | stab | \$0059 | save parity bit @ msb of 59.60 |



| F696 : 86 E2 | " " | ldaa | \#\$E2 |  |
| :---: | :---: | :---: | :---: | :---: |
| F698 : 9780 | " " | staa | PA |  |
| F69A : 8601 | " " | ldaa | \#\$01 |  |
| F69C : 9773 | " s" | staa | \$0073 | 73 <- 01 (key) |
| F69E : 8640 | " @" | ldaa | \#\$40 |  |
| F6AO : 9775 | " u" | staa | \$0075 | $75<-40$ (cti) |
| F6A2 : CE 0051 | " Q" | 1 dx | \#\$0051 |  |
| F6A5 : DF 00 | " " | stx | \$0000 | $0<-0051$ (end+1) |
| F6A7 : CE 0000 | " " | ldx | \#\$0000 | x <- 0000 (start) |
| F6AA : BD F7 8B | " " | jsr | \$F78B | send key \& cti to hybrid (81 bits) |
| F6AD : 8602 | " " | ldaa | \#\$02 | we sent 00.50cti/16.66key |
| F6AF : 9773 | " s" | staa | \$0073 | 73 <- 02 (key) |
| F6B1 : 8680 | " " | ldaa | \#\$80 |  |
| F6B3 : 9775 | " u" | staa | \$0075 | $75<-80$ (cti) |
| F6B5 : CE 00 3A | " : " | 1 dx | \#\$003A |  |
| F6B8 : DF 00 | " " | stx | \$0000 | $0<-003 \mathrm{~A}$ (end+1) |
| F6BA : CE 0002 | " " | ldx | \#\$0002 | x <- 0002 (start) |
| F6BD : BD F7 8B | " " | jsr | \$F78B | send key \& cti to hybrid (56 bits) |
| F6C0 : A6 00 | " " | ldaa | \$00, x | (003A) we sent 02.39cti/18.4Fkey |
| F6C2 : 8480 | " " | anda | \#\$80 |  |
| F6C4 : 2702 |  | beq | \$F6C8 |  |
| F6C6 : 8640 | " @" | ldaa | \#\$40 | cti |
| F6C8 : E6 16 | " " | ldab | \$16, ${ }^{\text {x }}$ | (0050) |
| F6CA : C4 02 | " " | andb | \#\$02 |  |
| F6CC : 2702 | "' " | beq | \$F6D0 |  |
| F6CE : 8A 20 | " " | oraa | \#\$20 | key |
| . . . . . | - • | - | - . |  |
| F6D0 : 8A 08 | " | oraa | \#\$08 | we- |
| F6D2 : 9782 | " " | staa | PB | send key \& cti to hybrid (1 bit) |
| F6D4 : 01 | " " | nop |  | we have now sent 138 bits total (1st 2 were dummy) |
| F6D5 : 01 | " " | nop |  |  |
| F6D6 : 8804 | " " | eora | \#\$04 | toggle clk |
| F6D8 : 9782 | " " | staa | PB |  |
| F6DA : $86 \mathrm{F0}$ | " " | ldaa | \#\$F0 |  |
| F6DC : 9780 | " " | staa | PA |  |
| F6DE : CE 03 E8 | " " | ldx | \#\$03E8 | x <- 1000 |
| F6E1 : BD FB F0 | " " | jsr | \$FBF0 |  |
| F6E4 : 8608 | " " | ldaa | \#\$08 | we- |
| F6E6 : 9782 | " " | staa | PB |  |
| F6E8 : BD F7 A7 | " " | jsr | \$F7A7 |  |
| F6EB : 8804 | " " | eora | \#\$04 |  |
| F6ED : 9782 | " " | staa | PB |  |
| F6EF : C6 F1 | " " | ldab | \#\$F1 |  |
| F6F1 : D7 80 | " | stab | PA |  |
| F6F3 : 08 | " " | inx |  |  |
| F6F4 : 09 | " " | dex |  |  |
| F6F5 : 01 | " " | nop |  |  |
| F6F6 : D6 74 | " t" | ldab | \$0074 |  |
| F6F8 : 2703 | "' | beq | \$F6FD |  |
| F6FA : BD FB C6 | " " | jsr | \$FBC6 |  |
| - . . . . . |  | . $\cdot$ | - . |  |
| F6FD : 8804 | " " | eora | \#\$04 |  |
| F6FF : 9782 | " | staa | PB |  |
| F701 : 01 |  | nop |  |  |
| F702 : D6 74 | " t" | ldab | \$0074 |  |
| F704 : BD FC D2 | " " | jsr | \$FCD2 |  |


| F707 : 7E F9 10 | "~ | jmp | \$F910 | display "beep?" |
| :---: | :---: | :---: | :---: | :---: |
| F70A : C6 03 | " | ldab | \#\$03 | get 1 digit of key \& put in buffer |
| . . . . . | - |  | - . |  |
| F70C : A6 00 |  | ldaa | \$00, x | get oct key digit |
| - . . . . | - . | - | - |  |
| F70E : 46 | "F" | rora |  | we shift digit right (lsb 1st) |
| F70F : 2403 | "\$ " | bcc | \$F714 |  |
| F711 : 730000 | "s | com | \$0000 | toggle parity if bit is 1 |
| - . | - $\cdot$ | - | - . |  |
| F714 : BD F7 26 | " \& " | jsr | \$F726 | put key bit into 51.58 |
| F717 : 5A | "Z" | decb |  |  |
| F718 : 26 F4 | "\& " | bne | \$F70E |  |
| F71A : 39 | "9" | rts |  |  |
| F71B : 08 | " " | inx |  | get leftover oct key digits |
| F71C : A6 00 |  | ldaa | \$00, x |  |
| . | - • | - | - . . |  |
| F71E : 46 | "F" | rora |  |  |
| F71F : 2401 | "\$ " | bcc | \$F722 |  |
| F721 : 5C | "\" | incb |  | count 1s in b for parity |
| - . . |  | - | - . |  |
| F722 : 760065 | "v e" | ror | \$0065 | put them in 65 |
| F725 : 39 | "9" | rts |  |  |
| F726 : 760051 | "v Q" | ror | \$0051 | >> bit into 64-bit fifo 51.58 |
| F729 : 760052 | "v R" | ror | \$0052 |  |
| F72C : 760053 | "v S" | ror | \$0053 |  |
| F72F : 760054 | "v T" | ror | \$0054 |  |
| F732 : 760055 | "v U" | ror | \$0055 |  |
| F735 : 760056 | "v V" | ror | \$0056 |  |
| F738 : 760057 | "v W" | ror | \$0057 |  |
| F73B : 760058 | "v X" | ror | \$0058 |  |
| F73E : 39 | "9" | rts |  | oldest bit in c |
| F73F : 740059 | "t Y" | 1 sr | \$0059 | >> 0 into 64-bit fifo 59.60 |
| F742 : 76005 A | "v Z" | ror | \$005A |  |
| F745 : 7600 5B | "v [" | ror | \$005B |  |
| F748 : 76005 C | "v \" | ror | \$005C |  |
| F74B : 7600 5D | "v ] " | ror | \$005D |  |
| F74E : 76005 E | "v ^" | ror | \$005E |  |
| F751 : 76005 F | "v _ " | ror | \$005F |  |
| F754 : 760060 | "v '" | ror | \$0060 |  |
| F757 : 39 | "9" | rts |  | oldest bit in c |
| F758 : 9F 61 | " a" | sts | \$0061 | interleave key/par into data bytes |
| F75A : 9E 63 | " c" | lds | \$0063 | load table addr |
| - . . | . |  | - . |  |
| F75C : 8608 |  | ldaa | \#\$08 | bit count |
| F75E : 9766 | " $\mathrm{f}^{\prime}$ | staa | \$0066 | 66 <- 8 |
| F760 : 32 | "2" | pula |  | get table byte |
| F761 : 9F 63 | " c" | sts | \$0063 | save table ptr sp |
| F763 : 9E 61 | " a" | lds | \$0061 | restore original sp |
| F765 : 46 | "F" | rora ${ }^{\text {b }}$ |  | ror table byte |
| F766 : 2505 | "\% " | bcs | \$F76D | determines where we get bits |
| F768 : BD F7 26 | " \& " | jsr | \$F726 | if bit 0, get from 51.58 (key) |


| F76B : 2003 | " " | bra | \$F770 |  |
| :---: | :---: | :---: | :---: | :---: |
| F76D : BD F7 3F | " ?" | jsr | \$F73F | if bit 1, get from 59.60 (par) |
|  | - • |  |  |  |
| F770 : 2406 | "\$ " | bcc | \$F778 | key/par bit was 0 |
| F772 : E6 00 | " | ldab | \$00, x | get existing data |
| F774 : DA 73 | " s" | orab | \$0073 | or in key/par bit per 73 (mask) |
| F776 : E7 00 | " | stab | \$00, x | save as new data |
| . | . $\cdot$ | - | . . |  |
| F778 : 08 | " " | inx |  | next data byte |
| F779 : 9C 00 | " " | cpx | \$0000 | see if done (end) |
| F77B : 27 OB |  | beq | \$F788 | done |
| F77D : 7A 0066 | "z f" | dec | \$0066 | bit count (was 8) |
| F780 : 26 E3 | "\& " | bne | \$F765 | get more bits |
| F782 : 9F 61 | " a" | sts | \$0061 |  |
| F784 : 9E 63 | " c" | lds | \$0063 | next table byte |
| F786 : 20 D4 | " | bra | \$F75C |  |
| F788 : 9E 61 | " a" | lds | \$0061 |  |
| F78A : 39 | "9" | rts |  |  |
| F78B : A6 00 |  | Idaa | \$00, x | clock out some key \& cti |
| F78D : 9475 | " u" | anda | \$0075 | cti from $\mathrm{x}+0$ |
| F78F : 2702 | "' " | beq | \$F793 |  |
| F791 : 8640 | " @" | ldaa | \#\$40 | set cti6 per mask in 75 (7/6) |
| . | . | - | - . |  |
| F793 : E6 16 | " " | ldab | \$16, x | key from $\mathrm{x}+16$ |
| F795 : D4 73 | " s" | andb | \$0073 |  |
| F797 : 2702 | "' | beq | \$F79B |  |
| F799 : 8A 20 | " " | oraa | \#\$20 | set key1 per mask in 73 (1/0) |
| . . . . | - | - | - - |  |
| F79B : 9782 |  | staa | PB |  |
| F79D : 08 | " " | inx |  |  |
| F79E : 8804 |  | eora | \#\$04 | toggle clk (10cyc width) |
| F7AO : 9782 |  | staa | PB |  |
| F7A2 : 9C 00 | " " | cpx | \$0000 |  |
| F7A4 : 26 E5 | "\& " | bne | \$F78B |  |
| F7A6 : 39 | "9" | rts |  |  |
| F7A7 : CE 0002 |  | 1 dx | \#\$0002 |  |
| F7AA : 86 F3 |  | ldaa | \#\$F3 |  |
| F7AC : D6 74 | " t" | ldab | \$0074 |  |
| F7AE : 2703 | "' | beq | \$F7B3 |  |
| F7B0 : BD FB C6 | " | jsr | \$FBC6 |  |
| . . . . . | - • | - | - $\cdot$ |  |
| F7B3 : C6 0C |  | 1 lab | \#\$0C |  |
| F7B5 : D7 82 |  | stab | PB |  |
| F7B7 : 9780 |  | staa | PA |  |
| F7B9 : D6 74 | " t" | ldab | \$0074 |  |
| F7BB : 2665 | "\&e" | bne | \$F822 |  |
| . . . . . | . . | - | - . |  |
| F7BD : A6 00 |  | ldaa | \$00, x |  |
| F7BF : 9782 |  | staa | PB |  |
| F7C1 : 8804 | " " | eora | \#\$04 |  |
| F7C3 : 6600 | "f " | ror | \$00, x |  |
| F7C5 : 8C 0015 | " " | cpx | \#\$0015 |  |
| F7C8 : 2707 | "' " | beq | \$F7D1 |  |




| F87F : 08 | " | inx |  |
| :---: | :---: | :---: | :---: |
| F880 : D6 74 | " t" | ldab | \$0074 |
| . . . . | . |  | . |
| F882 : C0 02 | " " | subb | \#\$02 |
| F884 : 01 | " " | nop |  |
| F885 : 01 | " " | nop |  |
| F886 : 26 FA | "\& " | bne | \$F882 |
| F888 : 20 D9 |  | bra | \$F863 |
| F88A : 9782 |  | staa | PB |
| F88C : D6 74 | " t" | ldab | \$0074 |
| F88E : D6 74 | " t" | ldab | \$0074 |
| . . . . . | - | - | . |
| F890 : C0 02 | " | subb | \#\$02 |
| F892 : 01 | " " | nop |  |
| F893 : 01 | " " | nop |  |
| F894 : 26 FA | "\& " | bne | \$F890 |
| F896 : CE 0002 | " " | 1 dx | \#\$0002 |
| , | - | . | . . . |
| F899 : A6 00 |  | ldaa | \$00, x |
| F89B : 84 FB |  | anda | \#\$FB |
| F89D : 9782 |  | staa | PB |
| F89F : 8804 | " | eora | \#\$04 |
| F8A1 : D6 74 | " t" | ldab | \$0074 |
| F8A3 : 08 | " " | inx |  |
| - . . . | - $\cdot$ |  | - • |
| F8A4 : C0 02 |  | subb | \#\$02 |
| F8A6 : 01 | " " | nop |  |
| F8A7 : 01 | " " | nop |  |
| F8A8 : 26 FA | "\& " | bne | \$F8A4 |
| F8AA : 8C 0050 | " P' | cpx | \#\$0050 |
| F8AD : 27 OD | "' | beq | \$F8BC |
| F8AF : 9782 |  | staa | PB |
| F8B1 : D6 74 | " t" | 1 dab | \$0074 |
| - | - | - | - . |
| F8B3 : C0 02 |  | subb | \#\$02 |
| F8B5 : 01 | " " | nop |  |
| F8B6 : 01 | " " | nop |  |
| F8B7 : 26 FA | "\& " | bne | \$F8B3 |
| F8B9 : 01 | " " | nop |  |
| F8BA : 20 DD |  | bra | \$F899 |
| F8BC : 9782 |  | staa | PB |
| F8BE : 01 | " " | nop |  |
| F8BF : 01 | " " | nop |  |
| F8C0 : D6 74 | " t" | ldab | \$0074 |
| F8C2 : C0 02 |  | subb | \#\$02 |
| F8C4 : 2706 | "' | beq | \$F8CC |
| . | . |  | - . |
| F8C6 : C0 02 |  | subb | \#\$02 |
| F8C8 : 01 | " " | nop |  |
| F8C9 : 01 | " " | nop |  |
| F8CA : 26 FA | "\& " | bne | \$F8C6 |
| F8CC : ${ }^{\text {C6 }}$ 3E | " >" | ldab | \#\$3E |
| F8CE : D7 81 | " " | stab | CRA |
| F8D0 : A6 00 |  | ldaa | \$00, x |



| F947 : 9782 | " " | staa | PB |
| :---: | :---: | :---: | :---: |
| F949 : 8804 | " " | eora | \#\$04 |
| F94B : 08 | " " | inx |  |
| F94C : 09 | " " | dex |  |
| F94D : 08 | " " | inx |  |
| F94E : 09 | " " | dex |  |
| F94F : 9782 | " " | staa | PB |
| F951 : 08 | " " | inx |  |
| F952 : 09 | " " | dex |  |
| F953 : 09 | " " | dex |  |
| F954 : 26 EF | "\& " | bne | \$F945 |
| F956 : 9F 61 | " a" | sts | \$0061 |
| F958 : CE FF CD | " | 1 dx | \#\$FFCD |
| - . . . . . | - | - | . |
| F95B : C6 08 |  | ldab | \#\$08 |
| - . . . . . | . $\cdot$ | - | - |
| F95D : 8628 | " (" | ldaa | \#\$28 |
| F95F : 9782 |  | staa | PB |
| F961 : 8804 | " " | eora | \#\$04 |
| F963 : 9782 | " " | staa | PB |
| F965 : 9680 | " " | ldaa | PA |
| F967 : 46 | "F" | rora |  |
| F968 : 46 | "F" | rora |  |
| F969 : 46 | "F" | rora |  |
| F96A : 46 | "F" | rora |  |
| F96B : 790046 | "y F" | rol | \$0046 |
| F96E : 5A | "Z" | decb |  |
| F96F : 26 EC | "\& " | bne | \$F95D |
| F971 : E6 00 | " " | 1 dab | \$00, x |
| F973 : D1 46 | " F" | cmpb | \$0046 |
| F975 : 2707 | "' | beq | \$F97E |
| F977 : C6 00 |  | ldab | \#\$00 |
| F979 : D7 69 | " i" | stab | \$0069 |
| F97B : 7E FB 87 | "~ " | jmp | \$FB87 |
| F97E : 08 | " " | inx |  |
| F97F : 8C FF D5 | " | cpx | \#\$FFD5 |
| F982 : 26 D7 | "\& " | bne | \$F95B |
| F984 : 9F 61 | " a" | sts | \$0061 |
| F986 : 8E 01 CE | " " | 1 ds | \#\$01CE |
| F989 : CE 0002 | " " | 1 dx | \#\$0002 |
| - | - . | - | - |
| F98C : 8604 |  | ldaa | \#\$04 |
| F98E : 9765 | " e" | staa | \$0065 |
| F990 : 32 | "2" | pula |  |
| - . . . . . | - $\cdot$ |  | - . |
| F991 : 46 | "F" | rora |  |
| F992 : C4 00 | " | andb | \#\$00 |
| F994 : 2402 | "\$ " | bcc | \$F998 |
| F996 : C6 40 | " @" | ldab | \#\$40 |
| . . . . . | . $\cdot$ | . ${ }^{\text {b }}$ | - . . |
| F998 : E7 00 |  | stab | \$00, x |
| F99A : 08 | " " | inx |  |
| F99B : 7A 0065 | "z e" | dec | \$0065 |
| F99E : 26 F1 | "\& " | bne | \$F991 |
| F9A0 : 8C 0042 | " B" | cpx | \#\$0042 |
| F9A3 : 26 E7 | "\& " | bne | \$F98C |



| FA05 : 5A | "Z" | decb |  |
| :---: | :---: | :---: | :---: |
| FA06 : 26 F4 | "\& " | bne | \$F9FC |
| FA08 : C6 40 | " @" | ldab | \#\$40 |
| FAOA : 8E 0001 | " " | lds | \#\$0001 |
| . . . . . | - • |  |  |
| FAOD : 32 | "2" | pula |  |
| FAOE : 8420 | " " | anda | \#\$20 |
| FA10 : 2702 |  | beq | \$FA14 |
| FA12 : 8601 | " " | ldaa | \#\$01 |
| $\text { FA14 : 8A } 08$ | " " |  | \#\$08 |
| FA16 : 36 | "6" | psha |  |
| FA17 : 31 | "1" | ins |  |
| FA18 : 5A | "Z" | decb |  |
| FA19 : 26 F2 | "\& " | bne | \$FAOD |
| FA1B : 9E 61 | " a" | lds | \$0061 |
| FA1D : CE 00 FA | " " | 1 dx | \#\$00FA |
| FA20 : BD FB F0 | " " | jsr | \$FBFO |
| FA23 : 86 F0 | " " | ldaa | \#\$F0 |
| FA25 : 9780 | " " | staa | PA |
| FA27 : CE 0258 | " X" | $1 d x$ | \#\$0258 |
| FA2A : BD FB F0 | " " | jsr | \$FBF0 |
| FA2D : 86 B9 | " " | ldaa | \#\$B9 |
| FA2F : 9780 | " " | staa | PA |
| FA31 : 9782 | " " | staa | PB |
| FA33 : CE 0190 | " " | $1 d x$ | \#\$0190 |
| FA36 : D6 74 | " t" | ldab | \$0074 |
| FA38 : 2703 |  | beq | \$FA3D |
| FA3A : BD FB C6 | " " | jsr | \$FBC6 |
| - • . . . | - • | - |  |
| FA3D : BD FB F0 | " " | jsr | \$FBF0 |
| FA40 : 08 | " " | inx |  |
| FA41 : 01 | " " | nop |  |
| FA42 : CE 0008 | " " | 1 dx | \#\$0008 |
| FA45 : 8804 | " " | eora | \#\$04 |
| FA47 : 9782 | " " | staa | PB |
| FA49 : C6 36 | " 6" | ldab | \#\$36 |
| FA4B : D7 81 |  | stab | CRA |
| FA4D : F6 0074 | " t" | 1 dab | \$0074 |
| FA50 : 2703 | "' | beq | \$FA55 |
| FA52 : BD FB C6 |  | jsr | \$FBC6 |
| . . . . . |  |  |  |
| FA55 : 8804 |  | eora | \#\$04 |
| FA57 : 9782 |  | staa | PB |
| FA59 : 01 | " " | nop |  |
| FA5A : 01 | " " | nop |  |
| FA5B : BD FC OF |  | jsr | \$FCOF |
| FA5E : 2705 | "' | beq | \$FA65 |
| FA60 : C6 01 |  | ldab | \#\$01 |
| FA62 : 7E FB 6E | "~n" | jmp | \$FB6E |
| FA65 : C6 3E | " >" | 1 dab | \#\$3E |
| FA67 : D7 81 |  | stab | CRA |
| FA69 : F6 0074 | " t" | ldab | \$0074 |
| FA6C : 2703 | "' | beq | \$FA71 |
| FA6E : BD FB C6 |  | jsr | \$FBC6 |
| . . . . . | * | . | - |



| FAE4 : 2703 | "' | beq | \$FAE9 |
| :---: | :---: | :---: | :---: |
| FAE6 : BD FB C6 | " " | jsr | \$FBC6 |
| - . . . . | - |  |  |
| FAE9 : 8804 |  | eora | \#\$04 |
| FAEB : 9782 | " " | staa | PB |
| FAED : BD FC 47 | " G" | jsr | \$FC47 |
| FAFO : 08 | " " | inx |  |
| FAF1 : CE 0097 | " " | ldx | \#\$0097 |
| FAF4 : 8804 | " " | eora | \#\$04 |
| FAF6 : 9782 | " " | staa | PB |
| FAF8 : D6 74 | " t" | ldab | \$0074 |
| FAFA : 2703 | "' | beq | \$FAFF |
| FAFC : BD FB C6 | " " | jsr | \$FBC6 |
| - . . . . . | - . | . | . $\cdot$ |
| FAFF : BD FC 09 | " " | jsr | \$FC09 |
| FB02 : 2705 | "' | beq | \$FB09 |
| - . . | - | . | . . . |
| FB04 : C6 05 |  | ldab | \#\$05 |
| FB06 : 7E FB 6E | "~n" | jmp | \$FB6E |
| FB09 : CE 0044 | " D" | ldx | \#\$0044 |
| $\stackrel{\text { FBOC : }}{ } 01$ | " | nop |  |
| FBOD : 01 | " " | nop |  |
| FB0E : D6 74 | " t" | ldab | \$0074 |
| FB10 : 2703 | ", " | beq | \$FB15 |
| FB12 : BD FB C6 | " " | jsr | \$FBC6 |
| - . . . . . | . $\cdot$ | . | . |
| FB15 : 8804 |  | eora | \#\$04 |
| FB17 : 9782 | " " | staa | PB |
| FB19 : 8804 | " " | eora | \#\$04 |
| FB1B : 9782 | " " | staa | PB |
| FB1D : D6 82 | " " | ldab | PB |
| FB1F : C4 01 | " " | andb | \#\$01 |
| FB21 : 26 E1 | "\& " | bne | \$FB04 |
| FB23 : 09 | " " | dex |  |
| FB24 : 26 E6 | "\& " | bne | \$FB0C |
| FB26 : CE 0001 | " " | 1 dx | \#\$0001 |
| FB29 : F6 0074 | " t" | ldab | \$0074 |
| FB2C : 2703 |  | beq | \$FB31 |
| FB2E : BD FB C6 | " | jsr | \$FBC6 |
| . . . . . . | - $\cdot$ |  | . $\cdot$ |
| FB31 : 8804 |  | eora | \#\$04 |
| FB33 : 9782 |  | staa | PB |
| FB35 : 01 | " " | nop |  |
| FB36 : 01 | " " | nop |  |
| FB37 : BD FC 0F | " " | jsr | \$FCOF |
| FB3A : 2605 | "\& " | bne | \$FB41 |
| FB3C : C6 06 |  | ldab | \#\$06 |
| FB3E : 7E FB 6E | "~n" | jmp | \$FB6E |
| FB41 : 6D 00 | "m " | tst | \$00, x |
| FB43 : D6 74 | " t" | ldab | \$0074 |
| FB45 : 2703 |  | beq | \$FB4A |
| FB47 : BD FB C6 | " | jsr | \$FBC6 |
| . . . . | - |  | - |
| FB4A : 8804 | " " | eora | \#\$04 |


| FB4C : 9782 | " | staa | PB |  |
| :---: | :---: | :---: | :---: | :---: |
| FB4E : BD FC 47 | " G" | jsr | \$FC47 |  |
| FB51 : BD FC D2 | " | jsr | \$FCD2 |  |
| FB54 : BD FB D0 | " " | jsr | \$FBD0 |  |
| FB57 : CE FF BF | " " | 1 dx | \#\$FFBF | Pass |
| FB5A : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| FB5D : B6 01 FE | " " | ldaa | \$01FE |  |
| FB60 : 840 F | " " | anda | \#\$0F |  |
| FB62 : CE 0052 | " R" | 1 dx | \#\$0052 |  |
| FB65 : DF 63 | " c" | stx | \$0063 |  |
| FB67 : BD F1 AF | " " | jsr | \$F1AF | convert oct to char |
| FB6A : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| FB6D : 39 | "9" | rts |  |  |
| FB6E : D7 69 | " i" | stab | \$0069 |  |
| FB70 : BD FC 47 | " G" | jsr | \$FC47 |  |
| FB73 : 86 3E | " >" | ldaa | \#\$3E |  |
| FB75 : 9781 | " | staa | CRA |  |
| FB77 : C6 16 | " " | ldab | \#\$16 |  |
| FB79 : BD FB C6 | " " | jsr | \$FBC6 |  |
| FB7C : C6 36 | " 6" | ldab | \#\$36 |  |
| FB7E : D7 81 |  | stab | CRA |  |
| FB80 : C6 64 | " d" | ldab | \#\$64 |  |
| FB82 : BD FB C6 | " " | jsr | \$FBC6 |  |
| FB85 : 9781 | " | staa | CRA |  |
| - . . . . . | . | - • | - . |  |
| FB87 : CE FF B9 | " | 1 dx | \#\$FFB9 | Fail |
| FB8A : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| FB8D : B6 01 FE | " | ldaa | \$01FE |  |
| FB90 : 8407 |  | anda | \#\$07 |  |
| FB92 : CE 0052 | " R" | 1 dx | \#\$0052 |  |
| FB95 : DF 63 | " c" | stx | \$0063 |  |
| FB97 : BD F1 AF | " | jsr | \$F1AF | convert oct to char |
| FB9A : BD FE 60 |  | jsr | \$FE60 |  |
| FB9D : A6 00 | " " | ldaa | \$00, x |  |
| FB9F : 847 F |  | anda | \#\$7F |  |
| FBA1 : A7 00 |  | staa | \$00, x |  |
| FBA3 : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| FBA6 : BD FB AA | " " | jsr | \$FBAA |  |
| FBA9 : 39 | "9" | rts |  |  |
| FBAA : 7F 0065 | " e" | clr | \$0065 |  |
| . . . | . | . $\cdot$ | . . . |  |
| FBAD : BD F1 53 | " S" | jsr | \$F153 |  |
| FBBO : CE 0000 | " " | 1 dx | \#\$0000 |  |
| FBB3 : 08 |  | inx | - . |  |
| FBB4 : BD FF 04 |  | jsr | \$FF04 |  |
| FBB7 : 8C 05 A0 | " | cpx | \#\$05A0 |  |
| FBBA : 26 F7 | "\& " | bne | \$FBB3 |  |
| FBBC : 7C 0065 | " \\| e" | inc | \$0065 |  |
| FBBF : 9665 | " e" | ldaa | \$0065 |  |
| FBC1 : 810 A |  | cmpa | \#\$0A |  |
| FBC3 : 26 E8 | "\& " | bne | \$FBAD |  |
| FBC5 : 39 | "9" | rts |  |  |
| FBC6 : C0 02 | " " | subb | \#\$02 |  |




| FC80 : 01 | " " | nop |  |
| :---: | :---: | :---: | :---: |
| FC81 : D6 74 | " t" | 1 dab | \$0074 |
| FC83 : 2608 | "\& " | bne | \$FC8D |
| FC85 : 8804 | " | eora | \#\$04 |
| FC87 : 9782 | " | staa | PB |
| FC89 : 01 | " " | nop |  |
| FC8A : 7E FC 9D | "~ | jmp | \$FC9D |
| FC8D : BD FB C6 | " | jsr | \$FBC6 |
| FC90 : 8804 | " | eora | \#\$04 |
| FC92 : 9782 | " | staa | PB |
| FC94 : 01 | " " | nop |  |
| FC95 : D6 74 | " t" | ldab | \$0074 |
| . . . . . | - | - | - . |
| FC97 : C0 02 | " " | subb | \#\$02 |
| FC99 : 01 | " | nop |  |
| FC9A : 01 | " " | nop |  |
| FC9B : 26 FA | "\& " | bne | \$FC97 |
| . | - | . | - . |
| FC9D : C6 36 | " 6" | 1 dab | \#\$36 |
| FC9F : D7 81 |  | stab | CRA |
| FCA1 : ${ }^{\text {a }}$ ( 00 | " " | ldaa | - $000, \mathrm{x}$ |
| FCA3 : 9782 | " | staa | PB |
| FCA5 : D6 74 | " t" | ldab | \$0074 |
| FCA7 : 27 OC | "' " | beq | \$FCB5 |
| FCA9 : 01 | " " | nop |  |
| FCAA : 01 | " " | nop |  |
| FCAB : C0 02 | " | subb | \#\$02 |
| FCAD : 2706 | "' | beq | \$FCB5 |
| - . $\cdot$ | . $\cdot$ | . ${ }^{\text {b }}$ | - . |
| FCAF : C0 02 |  | subb | \#\$02 |
| FCB1 : 01 | " " | nop |  |
| FCB2 : 01 | " " | nop |  |
| FCB3 : 26 FA | "\& " | bne | \$FCAF |
| - . . . . | - | - | - . |
| FCB5 : 8804 | " | eora | \#\$04 |
| FCB7 : 8C 0041 | " A" | cpx | \#\$0041 |
| FCBA : 2713 | "' | beq | \$FCCF |
| FCBC : 9782 |  | staa | PB |
| FCBE : 08 | " | inx |  |
| FCBF : D6 74 | " t" | ldab | \$0074 |
| FCC1 : 27 DE | "' | beq | \$FCA1 |
| FCC3 : C0 02 |  | subb | \#\$02 |
| FCC5 : 2706 | "' | beq | \$FCCD |
| . . . . . . | - . | - $\cdot$ | - . |
| FCC7 : C0 02 | " " | subb | \#\$02 |
| FCC9 : 01 | " " | nop |  |
| FCCA : 01 | " " | nop |  |
| FCCB : 26 FA | "\& " | bne | \$FCC7 |
| . . . | - | - $\cdot$ | . |
| FCCD : 20 D2 |  | bra | \$FCA1 |
| FCCF : 9782 | " " | staa | PB |
| FCD1 : 39 | "9" | rts |  |
| FCD2 : CE 0258 | " X" | 1 dx | \#\$0258 |


| FCD5 : 7E FC DA | "~ | jmp | \$FCDA |
| :---: | :---: | :---: | :---: |
| FCD8 : 08 | " " | inx |  |
| FCD9 : 09 | " " | dex |  |
| . . . . | - | - | - . |
| FCDA : 8844 | " D" | eora | \#\$44 |
| FCDC : 9782 | " " | staa | PB |
| FCDE : F6 0074 | " t" | ldab | \$0074 |
| FCE1 : 2703 | "' | beq | \$FCE6 |
| FCE3 : BD FB C6 | " " | jsr | \$FBC6 |
| . . . . . | - |  | - |
| FCE6 : 8804 |  | eora | \#\$04 |
| FCE8 : 9782 | " " | staa | PB |
| FCEA : 01 | " " | nop |  |
| FCEB : 01 | " " | nop |  |
| FCEC : 09 | " " | dex |  |
| FCED : 26 E9 | "\& " | bne | \$FCD8 |
| FCEF : 8804 |  | eora | \#\$04 |
| FCF1 : 9782 |  | staa | PB |
| FCF3 : 01 | " " | nop |  |
| FCF4 : 01 | " " | nop |  |
| FCF5 : D6 74 | " t" | 1 dab | \$0074 |
| FCF7 : 2624 | "\&\$" | bne | \$FD1D |
| FCF9 : CE 5D C0 | " ] " | 1 dx | \#\$5DC0 |
| FCFC : C6 08 | " | ldab | \#\$08 |
| FCFE : 86 2C | " , " | ldaa | \#\$2C |
| $\text { FDOO : } 9782$ | " " | staa | ${ }^{\text {PB }}{ }^{\text {a }}$ |
| FD02 : 5A | "Z" | decb |  |
| FD03 : 2704 | "' | beq | \$FD09 |
| FD05 : 8804 | " " | eora | \#\$04 |
| FD07 : 2005 | " " | bra | \$FD0E |
| FD09 : 8844 | " D" | eora | \#\$44 |
| FDOB : C6 08 | " " | ldab | \#\$08 |
| FDOD : 01 | " " | nop |  |
| - . . . . . | . $\cdot$ |  | - $\cdot$ |
| FDOE : 9782 | " | staa | PB |
| FD10 : 8804 | " " | eora | \#\$04 |
| FD12 : 08 | " " | inx |  |
| FD13 : 09 | " " | dex |  |
| FD14 : 01 | " " | nop |  |
| FD15 : 09 | " " | dex |  |
| FD16 : 26 E8 | "\& " | bne | \$FD00 |
| FD18 : 9782 | " " | staa | PB |
| FD1A : 7E FD 67 | "~ g" | jmp | \$FD67 |
| FD1D : C1 02 |  | cmpb | \#\$02 |
| FD1F : 2607 | "\& " | bne | \$FD28 |
| FD21 : CE 3E 80 | " > " | $1 d x$ | \#\$3E80 |
| FD24 : C6 05 |  | ldab | \#\$05 |
| FD26 : 2016 | " " | bra | \$FD3E |
| FD28 : C1 04 |  | cmpb | \#\$04 |
| FD2A : 2607 | "\& " | bne | \$FD33 |
| FD2C : CE 2E E0 | " . " | $1 d x$ | \#\$2EE0 |
| FD2F : C6 04 |  | 1 dab | \#\$04 |


| FD31 : 2008 | " " | bra | \$FD3B |
| :---: | :---: | :---: | :---: |
| FD33 : CE 2580 | " \% " | 1 dx | \#\$2580 |
| FD36 : C6 03 | " " | ldab | \#\$03 |
| FD38 : BD FF 07 | " " | jsr | \$FF07 |
| FD3B : BD FF OC | " " | jsr | \$FFOC |
| - . . . . |  |  |  |
| FD3E : D7 75 | " u" | stab | \$0075 |
| FD40 : D7 65 | " e" | stab | \$0065 |
| FD42 : 01 |  | nop |  |
| FD43 : 86 2C | " , " | ldaa | \#\$2C |
| $\text { FD45 : } 9782$ | " " | staa | PB |
| FD47 : D6 65 | " e" | 1 dab | \$0065 |
| FD49 : 5A | "Z" | decb |  |
| FD4A : 2704 | ", | beq | \$FD50 |
| FD4C : 8804 |  | eora | \#\$04 |
| FD4E : 2005 | " " | bra | \$FD55 |
| FD50 : 8844 | " D" | eora | \#\$44 |
| FD52 : F6 0075 | " u" | ldab | \$0075 |
| FD55 : 9782 | " | staa | PB |
| FD57 : 8804 | " " | eora | \#\$04 |
| FD59 : D7 65 | " e" | stab | \$0065 |
| FD5B : D6 74 | " t" | ldab | \$0074 |
| FD5D : BD FB C6 | " " | jsr | \$FBC6 |
| FD60 : 09 | " " | dex |  |
| FD61 : 26 E2 | "\& " | bne | \$FD45 |
| FD63 : 9782 |  | staa | PB |
| FD65 : D6 75 | " u" | ldab | \$0075 |
| FD67 : BD FF 0C | " " | jsr | \$FFOC |
| FD6A : CE 0002 | " " | 1 dx | \#\$0002 |
| $\text { FD6D : } 8804$ | " | eora | \#\$04 |
| FD6F : 9782 |  | staa | PB |
| FD71 : F6 0074 | " t" | ldab | \$0074 |
| FD74 : 2703 | "' | beq | \$FD79 |
| FD76 : BD FB C6 | " " | jsr | \$FBC6 |
| - . . . . | . $\cdot$ |  |  |
| FD79 : 8804 |  | eora | \#\$04 |
| FD7B : 9782 |  | staa | PB |
| FD7D : D6 80 |  | ldab | PA |
| FD7F : E7 00 |  | stab | \$00, x |
| FD81 : 08 | " " | inx |  |
| FD82 : 8C 0042 | " B" | cpx | \#\$0042 |
| FD85 : 26 E6 | "\& " | bne | \$FD6D |
| FD87 : 8804 |  | eora | \#\$04 |
| FD89 : 9782 |  | staa | PB |
| FD8B : F6 0074 | " t" | 1 dab | \$0074 |
| FD8E : 2703 | "' | beq | \$FD93 |
| FD90 : BD FB C6 |  | jsr | \$FBC6 |
| FD93 : 8804 |  | eora | \#\$04 |
| FD95 : 9782 | " " | staa | PB |


| FD97 : 8804 | " | eora | \#\$04 |  |
| :---: | :---: | :---: | :---: | :---: |
| FD99 : 9782 | " " | staa | PB |  |
| FD9B : CE 03 E 8 | " " | 1 dx | \#\$03E8 |  |
| FD9E : C6 F0 | " | ldab | \#\$F0 |  |
| FDAO : D7 80 | " | stab | PA |  |
| FDA2 : D6 74 | " t" | ldab | \$0074 |  |
| FDA4 : D6 74 | " t" | ldab | \$0074 |  |
| FDA6 : 2703 | "' | beq | \$FDAB |  |
| FDA8 : BD FB C6 | " " | jsr | \$FBC6 |  |
| . . . . | - . | . $\cdot$ | - . |  |
| FDAB : BD FB FO | " " | jsr | \$FBFO |  |
| FDAE : 39 | "9" | rts |  |  |
| FDAF : 9672 | " r" | ldaa | \$0072 |  |
| FDB1 : 2709 | "' " | beq | \$FDBC |  |
| FDB3 : 8101 |  | cmpa | \#\$01 |  |
| FDB5 : 2717 | "' " | beq | \$FDCE |  |
| FDB7 : 8107 |  | cmpa | \#\$07 |  |
| FDB9 : 2632 | "\&2" | bne | \$FDED |  |
| FDBB : 39 | "9" | rts |  |  |
| FDBC : CE FF B3 | " " | 1 dx | \#\$FFB3 | Loc? |
| FDBF : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| FDC2 : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| FDC5 : 8605 | " " | ldaa | \#\$05 |  |
| FDC7 : 9769 | " i" | staa | \$0069 |  |
| FDC9 : 7C 0072 | " \\| r" | inc | \$0072 |  |
| FDCC : 20 3B | " ; | bra | \$FE09 |  |
| FDCE : CE FF AD |  | 1 dx | \#\$FFAD | (6 spaces) |
| FDD1 : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| FDD4 : 868 B |  | ldaa | \#\$8B |  |
| FDD6 : 9751 | " Q" | staa | \$0051 |  |
| FDD8 : 864 B | " K" | ldaa | \#\$4B |  |
| FDDA : 9752 | " R" | staa | \$0052 |  |
| FDDC : 8602 |  | ldaa | \#\$02 |  |
| FDDE : 9769 | " i" | staa | \$0069 |  |
| FDE0 : CE 0002 | " " | $1 d x$ | \#\$0002 |  |
| FDE3 : DF 4D | " M" | stx | \$004D |  |
| . . . . . | . | . | . $\cdot$ |  |
| FDE5 : 6F 00 | "0 " | clr | \$00, x |  |
| FDE7 : 08 | " " | inx |  |  |
| FDE8 : 8C 0008 | " " | cpx | \#\$0008 |  |
| FDEB : 26 F 8 | "\& " | bne | \$FDE5 |  |
| . . . . . |  | . | - . |  |
| FDED : 9668 | " h" | ldaa | \$0068 |  |
| FDEF : DE 4D | " M" | 1 dx | \$004D |  |
| FDF1 : A7 00 |  | staa | \$00, x |  |
| FDF3 : BD FE 60 | " '" | jsr | \$FE60 |  |
| FDF6 : DF 63 | " c" | stx | \$0063 |  |
| FDF8 : BD F1 AF | " | jsr | \$F1AF | convert oct to char |
| FDFB : 7C 0072 | " \\| r" | inc | \$0072 |  |
| FDFE : BD FF OD | " | jsr | \$FFOD |  |
| FE01 : 7F 006 E | " n " | clr | \$006E |  |
| FE04 : DE 4D | " M" | ldx | \$004D |  |
| FE06 : 08 |  | inx |  |  |
| FE07 : DF 4D | " M" | stx | \$004D |  |



| FE57 : C4 07 | " | andb | \#\$07 |
| :---: | :---: | :---: | :---: |
| . . . . . | - | - |  |
| FE59 : 2704 | "' | beq | \$FE5F |
| FE5B : 08 | " " | inx |  |
| FE5C : 5A | "Z" | decb |  |
| FE5D : 20 FA | " " | bra | \$FE59 |
| FE5F : 39 | "9" | rts |  |
| FE60 : CE 0051 | " Q" | 1 dx | \#\$0051 |
| FE63 : D6 69 | " i" | ldab | \$0069 |
| - . . . | - | - | . |
| FE65 : 2704 | "' | beq | \$FE6B |
| FE67 : 08 | " " | inx |  |
| FE68 : 5A | "Z" | decb |  |
| FE69 : 20 FA | " " | bra | \$FE65 |
| FE6B : 39 | "9" | rts |  |
| FE6C : 5F | "_" | clrb |  |
| FE6D : 8601 | " | ldaa | \#\$01 |
| FE6F : 9B 4F | " 0" | adda | \$004F |
| FE71 : 97 4F | " 0" | staa | \$004F |
| FE73 : D9 50 | " P" | adcb | \$0050 |
| FE75 : D7 50 | " P" | stab | \$0050 |
| FE77 : 96 4F | " O" | ldaa | \$004F |
| FE79 : 2701 | ", " | beq | \$FE7C |
| FE7B : 39 | "9" | rts |  |
| FE7C : 96 6B | " k" | ldaa | \$006B |
| FE7E : 2622 | "\&" | bne | \$FEA2 |
| FE80 : 9672 | " r" | ldaa | \$0072 |
| FE82 : 8101 | " | cmpa | \#\$01 |
| FE84 : 22 1C | "" | bhi | \$FEA2 |
| FE86 : C1 25 | " \%" | cmpb | \#\$25 |
| FE88 : 2201 | "" | bhi | \$FE8B |
| FE8A : 39 | "9" | rts |  |
| FE8B : BD F1 D2 | " " | jsr | \$F1D2 |
| FE8E : 9650 | " P" | ldaa | \$0050 |
| FE90 : 8140 | " @" | cmpa | \#\$40 |
| FE92 : 2603 | "\& " | bne | \$FE97 |
| FE94 : BD F1 53 | " S" | jsr | \$F153 |
| - |  |  | - 0 |
| FE97 : 9650 | " P" | ldaa | \$0050 |
| FE99 : 8150 | " P" | cmpa | \#\$50 |
| FE9B : 2201 | "" | bhi | \$FE9E |
| FE9D : 39 | "9" | rts |  |
| FE9E : BD F5 87 | " " | jsr | \$F587 |
| FEA1 : 39 | "9" | rts |  |
| FEA2 : BD FE 60 | " '" | jsr | \$FE60 |
| FEA5 : E6 00 | " " | ldab | \$00, x |
| FEA7 : D7 4A | " J" | stab | \$004A |
| FEA9 : 9650 | " P" | ldaa | \$0050 |
| FEAB : 8401 | " " | anda | \#\$01 |


| FEAD : 2704 | "' | beq | \$FEB3 |  |
| :---: | :---: | :---: | :---: | :---: |
| FEAF : 8680 | " | ldaa | \#\$80 |  |
| FEB1 : A7 00 | " " | staa | \$00, x |  |
| . | - | . | - . |  |
| FEB3 : BD F1 70 | " p" | jsr | \$F170 | write buffer to display |
| FEB6 : BD FE 60 | " ${ }^{\text {c }}$ | jsr | \$FE60 |  |
| FEB9 : 964 A | " J" | ldaa | \$004A |  |
| FEBB : A7 00 | " | staa | \$00, x |  |
| . . . . . | . | . | . . |  |
| FEBD : 9650 | " P" | ldaa | \$0050 |  |
| FEBF : 81 E0 |  | cmpa | \#\$E0 |  |
| FEC1 : 2603 | "\& " | bne | \$FEC6 |  |
| FEC3 : BD F1 53 | " S" | jsr | \$F153 |  |
| . . . | - . | - | - . |  |
| FEC6 : 9650 | " P" | ldaa | \$0050 |  |
| FEC8 : 81 E9 | " " | cmpa | \#\$E9 |  |
| FECA : 22 D2 | "" " | bhi | \$FE9E |  |
| FECC : 39 | "9" | rts |  |  |
| FECD : CE FF AD |  | 1 dx | \#\$FFAD | (6 spaces) |
| FEDO : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| FED3 : B6 01 FE | " " | ldaa | \$01FE |  |
| FED6 : 8407 |  | anda | \#\$07 |  |
| FED8 : CE 0051 | " Q" | 1 dx | \#\$0051 |  |
| FEDB : DF 63 | " c" | stx | \$0063 |  |
| FEDD : BD F1 AF | " " | jsr | \$F1AF | convert oct to char |
| FEEO : 9651 | " Q" | ldaa | \$0051 |  |
| FEE2 : 847 F | " " | anda | \#\$7F |  |
| FEE4 : 9751 | " Q" | staa | \$0051 |  |
| FEE6 : 39 | "9" | rts |  |  |
| FEE7 : CE 0100 | " | 1 dx | \#\$0100 |  |
| FEEA : DF 63 | " c" | stx | \$0063 | 63.64 <- start of key ram |
| FEEC : F6 01 FE | " " | ldab | \$01FE | get key\# |
| FEEF : C4 07 | " " | andb | \#\$07 |  |
| FEF1 : $27{ }^{\circ} 10^{\circ}$ | "; " | beq | ${ }_{\text {\$FF03 }}{ }^{\text {a }}$ | quit when addr is correct |
| FEF3 : 5A | "Z" | decb |  |  |
| FEF4 : 9664 | " d" | ldaa | \$0064 | add 24 to 63.64 for each key\# |
| FEF6 : 8B 18 | " | adda | \#\$18 |  |
| FEF8 : 9764 | " d" | staa | \$0064 |  |
| FEFA : 9663 | " c" | ldaa | \$0063 |  |
| FEFC : 8900 |  | adca | \#\$00 |  |
| FEFE : 9763 | " c" | staa | \$0063 |  |
| FF00 : 5D | "] " | tstb |  |  |
| FF01 : 20 EE | " " | bra | \$FEF1 |  |
| FF03 : 39 | "9" | rts |  |  |
| FF04 : 01 | " " | nop |  |  |
| FF05 : 01 | " | nop |  |  |
| FF06 : 01 | " " | nop |  |  |
| - | - . | . . | - . |  |
| FF07 : 01 | " | nop |  |  |
| FF08 : 01 |  | nop |  |  |
| FF09 : 01 |  | nop |  |  |
| FFOA : 01 | " " | nop |  |  |




FFD5 : 00000000
FFD9 : 0000

| FFDB : OF | " " | sei |  | low battery interrupt |
| :---: | :---: | :---: | :---: | :---: |
| FFDC : 9679 | " y" | ldaa | \$0079 |  |
| FFDE : 8A 10 | " " | oraa | \#\$10 |  |
| FFEO : 9779 | y" | staa | \$0079 |  |
| FFE2 : D6 67 | g" | ldab | \$0067 |  |
| FFE4 : CA 20 | " " | orab | \#\$20 |  |
| FFE6 : D7 67 | g" | stab | \$0067 |  |
| FFE8 : CE FF A5 | " " | 1 dx | \#\$FFA5 | LOCharge |
| FFEB : BD F1 C0 | " " | jsr | \$F1C0 | put table chars in buffer |
| FFEE : BD F1 70 | p" | jsr | \$F170 | write buffer to display |
| FFF1 : BD FF 34 | 4" | jsr | \$FF34 |  |
| FFF4 : BD F3 68 | " h" | jsr | \$F368 |  |

