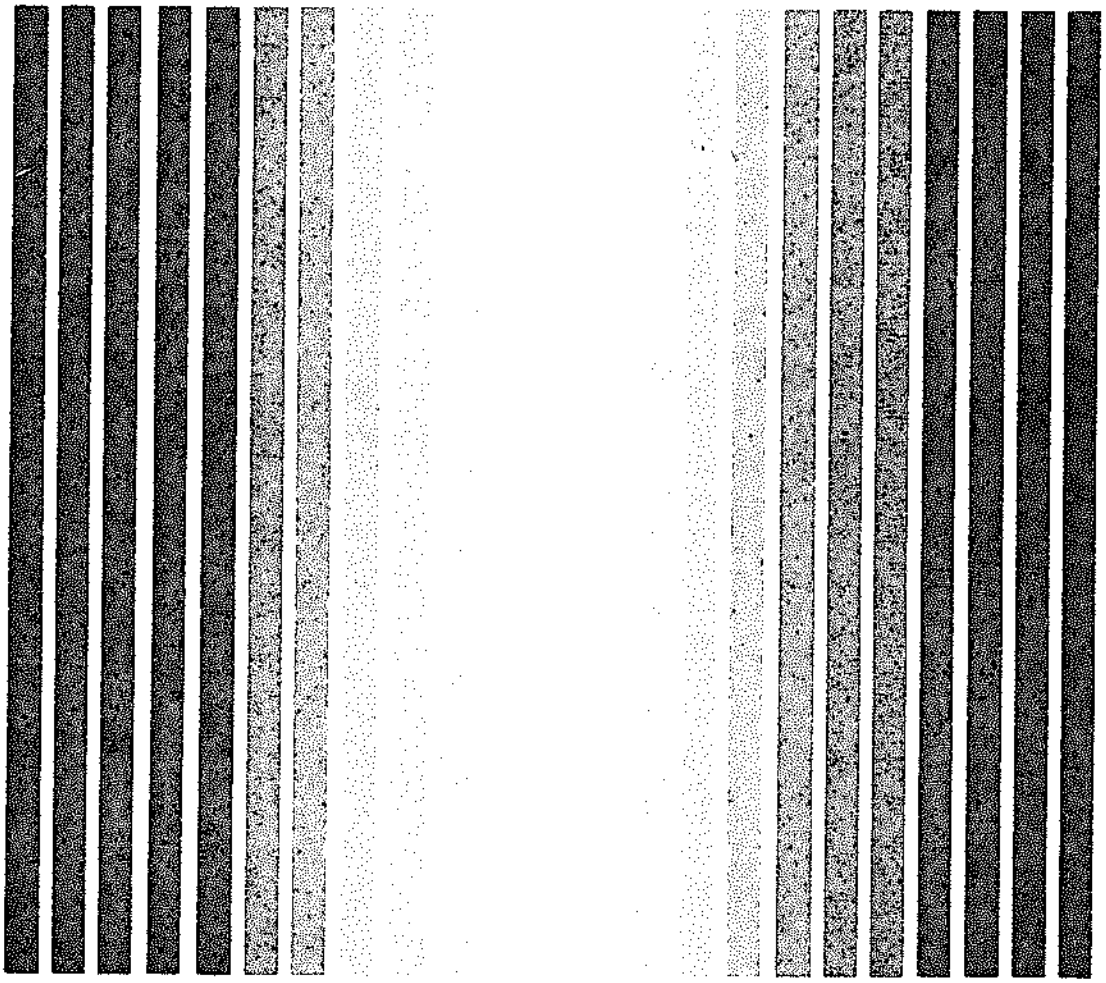
APX ATARI® PROGRAM EXCHANGE



Chris Crawford

SOURCE CODE FOR EASTERN FRONT (1941)

A behind-the-scenes look at creating a complex war game (for advanced programmers)

Diskette: 40K (APX-20095)

User-Written Software for ATARI Home Computers

Distributed By

The ATARI Program Exchange
P.O. Box 3705
Santa Clara, CA 95055

To request an APX Product Catalog, write to the address above, or call toll-free:

800/538-1862 (outside California)
800/672-1850 (within California)

Or call our Sales number, 408/727-5603

Trademarks of Atari

The following are trademarks of Atari, Inc.

ATARI®
ATARI 400™ Home Computer
ATARI 800™ Home Computer
ATARI 410™ Program Recorder
ATARI 810™ Disk Drive
ATARI 820™ 40-Column Printer
ATARI 822™ Thermal Printer
ATARI 825™ 80-Column Printer
ATARI 830™ Acoustic Modem
ATARI 850™ Interface Module

Printed in U.S.A.

SOURCE CODE FOR EASTERN FRONT (1941)

by

Chris Crawford

EASTERN FRONT DOCUMENTATION PACKAGE

This package contains material of value to any programmer attempting to study the program EASTERN FRONT (1941). My purpose in making these materials available is to provide programmers with an instructive lesson in designing and programming a major game. This program demonstrates many aspects of the game designer's art: high-level design concepts, algorithms for wargames, programming structure and technique, and specific applications of the special capabilities of the ATARI Home Computer™. I cannot claim that the program is of textbook clarity; indeed, it is fraught with clumsy inanities. I made no efforts to conceal or correct the mistakes in the program. I believe that most programmers live by a double standard. They expect all code to be clean, tight, and elegant, yet they are seldom able to achieve this goal. I wanted to show this program "warts and all". I am not proud of the warts; I simply won't deny their existence. Furthermore, they are themselves instructive. By studying them, the programmer can see how mistakes are made and can better avoid them.

My hope is that people will study these materials to become better programmers with the ATARI Home Computer. There will also be smaller-minded individuals who see them not as instructional materials but as sources of profit. I'm sure some yokel will perform some trivial modifications to the code and start selling WESTERN FRONT 1944 or some similar rip-off. Modifying an existing program is a useful exercise for the beginning programmer. Selling such a program without proper authorization is not legally secure, economically realistic, or professionally respectable. If you are seriously interested in modifying EASTERN FRONT 1941 for commercial reasons, then contact me before you begin work. I will entertain proposals for extensions which do not sully the original product.

This is a very complex program; to explain completely all aspects of the program would take far too much space. I have tried to include in this package all the key items that a programmer would need to understand the program. I assume that the user of this package is already a competent programmer who is familiar with assembly language and the structure of the ATARI Home Computer. I also assume that you have played the game and understand its functions. This makes my task shorter. If you are a beginning programmer, you will not be able to understand what is in here. Even the competent programmer will find some of the quirks of this program mystifying. A few of these strange quirks are brilliant strokes of programming genius; the majority are simple mistakes.

Chris Crawford

The following items are included in this package:

EASTERN FRONT STRUCTURE

EASTERN FRONT 1941 is divided into six modules. The program was developed with the Atari Assembler/Editor cartridge, which has no linking facility. Therefore, the modules were linked by hand. This makes the program more difficult to understand and modify.

The six modules and their functions are as follows:

FONTS.DAT	a data module containing character fonts for the map
EFT18D.ASM	Data module: display list, map and troop data
EFT18I.ASM	Interrupt routines: joystick, scrolling, orders
EFT18M.ASM	Mainline: initialization, movement, seasons
EFT18C.ASM	Combat: combat and logistics routines
EFT18T.ASM	Thinking: artificial intelligence routines

The sequence above is the historical sequence in which the modules were developed. The later modules are structurally higher than the earlier ones. They frequently make use of subroutines and tables in the earlier ones while the reverse is rare.

The program was designed to run in a 16K machine with a cassette only. To achieve this goal I had to scrunch the program very tightly. The lack of good linking facilities made scrunching a difficult task. I was forced to take some subroutines and data tables out of one module and insert them into another module. Many times the positioning of a subroutine or table was decided not by logic or structure but rather by the fortuitous discovery of a chunk of space in one module that was precisely the right size to accommodate the homeless code.

Virtually all of the memory space available to the 16K system is used. There are a few unused chunks of RAM, but they are rather small. I did preserve the 1K region used by the Operating System for its Mode 0 display list and display data. This RAM could be stolen by a desperate programmer, but the Mode 0 display shown while loading the program would go wild, possibly frightening the user into unfortunate recourse to the SYSTEM RESET key. The programmer should study the global memory map on page 54 very closely before appropriating any memory. You should also refer to the appropriate source code listing. I repeat, there is very little available memory.

DATA MODULE

This is the simplest of the modules. It is nothing more than a collection of data bytes. Many inexperienced programmers think of a program in terms of the executable code. The code is only one portion of the entire program. The data is the other major component. Both components are necessary, but many programmers neglect the data. Don't make this mistake. The data needs as much attention as the code.

MILITARY STATE VARIABLES

The first data tables are the values for the military units. These are presented in a more orderly fashion in the Unit Characteristics Chart on pages 56-58. There are 159 different units recognized in this game. Of these, 54 are German and 105 are Russian. These numbers are critical; you will see them often in the code in one form or another.

The first two data tables are CORPSX and CORPSY (lines 30-330). These tables specify the initial map coordinates for the military units, corps for the Germans and armies for the Russians. The coordinate system is the same one used for the map; see the map reproduced on page 55.

The next two data tables are MSTRNG and CSTRNG (lines 340-570). These tables store the muster and combat strength of the units. The combat strength is initialized at the beginning of the game to equal the muster strength.

Next comes the SWAP table (lines 580-790). This table serves two purposes. It contains the character type of the unit (infantry or armor) for use when the unit is put onto the map. The same table also acts as a buffer to store the terrain underneath the unit. The unit's image is swapped with the terrain image, hence the label.

The table called ARRIVE (lines 800-1000) tells the turn on which each unit first arrives on the map. It is a reinforcement schedule. Note that some units are set to arrive on turn 255. As in the real world, it is sometimes more convenient to postpone beyond reasonable limits some commitment that we cannot actually refuse but no longer wish to honor. This table is frequently used to determine if a unit is on the map. Many sections of code begin with LDA ARRIVE, X/CMP #$FF/BEQ NEXT to weed out units that are either already dead or not yet on the map.

CORPT (lines 1180-1380) specifies the type of unit. There are many different types of units in this game, but only three types are recognized in the mechanics of the game: infantry, armor, and militia. I do recognize different types of units for identification purposes. There are panzergrenadier, mountain, paratroop, and SS units for the Germans and Guards, tank and shock armies for the Russians, among others. There are also the different nationalities. All these factors are encoded in the single CORPT constant.

CORPNO (lines 1390-1590) specifies the military unit number, as in the 48th Panzer Corps. This is another quantity that has no significance to the

operation of the game but is included for the unit description when a unit is examined. Such nonfunctional elements in a game are referred to as "color". I call them "dirt". My bad manners are exceeded only by my hypocrisy, for I still use such elements in my own games. Oink.

These eight parameters completely determine the state of a military unit. They were the first items I defined when I set about designing the game. By defining them at the outset, I fixed what the game would and would not be able to do. This lent focus to the design. Before doing any simulation, you must declare precisely what you know before you attempt to do anything with it.

WORDS TABLE

Another chunk of this module is devoted to the WORDS table (lines 1010-1170), which gives the text strings used in the text windows. I decided to use a fixed field size of eight characters rather than a variable field size. There are only a few cases where the words I need are too long to fit: SEPTEMBR, HUNGARAN, PARATRP, PZRGRNDR. The decision to use eight characters per field was a good one. The code to put text on the screen is fast and simple, and the data tables required are short.

CONVERTING BYTES TO DIGITS

Line 1600 begins one of the strangest ideas I have ever implemented in a program. It is also one of the stupidest. I was worried about the conversion of hexadecimal byte values in my tables into numeral strings on the screen. Whenever the player presses the button to raise a unit in the cursor, the interrupt routine must put a considerable amount of information into the text window. It must first find out which unit is in the cursor, then look up the unit's CORPNO, CORPT, MSTRNG, and CSTRNG. It must then translate all these quantities into readable text and place that text onto the text window. Furthermore, the entire operation must be completed during the 2000 machine cycles available in the vertical blank interrupt routine. These requirements impose severe time constraints on any code.

My solution was pretty ruthless. I created three tables in memory, one for the hundreds digit of a byte, one for the tens digit, and one for the ones digit. With these tables the task of hexadecimal to decimal text conversion became simple. I put the byte to be converted into the X register and LDA HDIGIT,X. That one instruction produces the hundreds digit. Similar operations with TDIGIT and ODIGIT give the other digits. The total time for conversion is 12 cycles. That's extremely fast! Unfortunately, it is also extremely RAM-expensive. Those three tables require 768 bytes.

The alternative is to calculate the conversion value rather than look it up. The following routine is a standard way to solve the problem:

```
;start with byte to be converted in accumulator
        LDX  #$FF
        SEC
LOOP1   INX
        SBC  #$64
        BCS  LOOP1
        STX  HDIGIT
        ADC  #$64
        LDX  #$FF
        SEC
LOOP2   INX
        SBC  #$0A
        BCS  LOOP2
        STX  TDIGIT
        ADC  #$0A
        STA  ODIGIT
```

This code will require at most 108 cycles to execute. Now, 108 cycles is not much machine time, but the conversion has to be done three times during vertical blank interrupt. Thus the method I chose to use saves me nearly 300 machine cycles out of 2000 available. That is why I chose a memory-wasteful algorithm.

Did I make the right decision? It is very difficult to calculate how many cycles my routine needs. I know that it consumes at least 1700 cycles in the worst case. Without a logic analyzer it is very difficult to say anything more. I might have gotten away with the standard algorithm. This discussion illustrates the nature of the guesswork that a designer must use. When you are in the early stages of writing a program, you have no way of knowing how big or how slow your code will be. You must rely on hunches. My hunch told me to trade memory for time. Such conservatism is very important in the early stages of the programming phase. Once a problem is built into a program, it is extremely difficult to expunge. Problems should be prevented before you have exhausted your reserves of memory and execution time.

MORE MISCELLANEOUS TABLES

The next table in the data module is called TXTTBL (lines 2450-2500). It is a table of long text messages. I chose a fixed field length of 32 bytes for these messages. There are only three messages here.

MONLEN (lines 2510-2520) is a table giving the lengths in days of the months. MONLEN is 13 bytes long. More astute readers may recall that this does not quite correspond with the number of months in a year. This is an example of lazy coding. I chose to number my months from 1 rather than zero. It made more sense to me. I was unwilling to hassle with the

redefinition problem arising from my inappropriate numbering system. Rather than think the problem through I decided on a brazen solution. "What the hell!", I cried, "Let's waste a byte! I've got plenty to spare!" I'm a devil-may-care rascal.

The next two tables, HMORDS and WHORDS (lines 2530-2540), keep track of the orders given to the units during the course of the game. They are initialized to zero at the beginning of the game. HMORDS tells how many valid orders are in storage, and WHORDS tells what the orders are. This game uses a rectangular grid, so each unit can move in any of four directions. It takes two bits to specify one of four orders. Thus, the two bytes of WHORDS allocated for each unit can store up to eight orders.

There is an interesting bug in EASTERN FRONT 1941 associated with these two tables. Under certain conditions HMORDS can get a value greater than eight. When this happens the arrow showing the future path of the unit keeps moving right off the edge of the map. I have never found the cause of the bug. The bug is rare and nondestructive, so I never bothered expending the time to track it down.

BEEPTB (line 2550) is a table of frequencies used to give feedback when the joystick is used to give orders.

ERRMSG (lines 2560-2630) is a table of error messages. Like the other text messages, I use a fixed field length of 32 bytes. Only four error messages are supported, yet together they consume 128 bytes of RAM. This demonstrates why textual error messages are so rare in personal computers.

The number and type of error messages are a revealing indication of the quality of human engineering in the program. The ideal program has no error messages, because it would make errors inconceivable. The four errors generated by this program could have been avoided with sufficient effort on my part. All four concern the entry of orders. The "only eight orders allowed" error could have been prevented by the simple expedient of using more bytes for storing orders. Of course, there has to be some kind of limit, and I think eight is a reasonable limit, so I can rest easy with this one. The "please wait for Maltakreuze" error was purely a matter of programmer convenience; I had problems implementing the code necessary to allow orders to be entered immediatly, so I hid behind the excuse that the user should wait to see what he has already entered before he adds new orders. Again, this is a reasonable defense. I now think that I should have sped up the arrow so that it moves faster. This would have made the error less common. The error "That is a Russian unit" could have been dispensed with. It might have been better to ignore orders given to Russian units. I don't know about this one. The last error, "no diagonal moves allowed", bothers me greatly. I could have allowed diagonal moves, simply interpreting a diagonal move as a combination of horizontal and vertical moves. However, the resolution on the joystick is so poor that many people can mistakenly enter a diagonal move when they intended to enter only a horizontal or vertical move. I am torn between protecting my user and accommodating him.

The tables in lines 2640-2680 are used for logical manipulation of the joystick entries and for unit motion.

TRTAB (lines 2690-2700) is a table of monthly colors for trees. It is the table that allows me to change the color of the trees as the seasons go by. It is only 13 bytes long. The extra byte can be attributed to my wanton disregard for the requirements of tight coding.

MLTKRZ (line 2710) is a bit map of the maltese cross.

The RAM from $6000-$63FF is reserved for the two graphics character sets. They are contained in file FONTS.DAT.

The display list comes next (lines 2780-2830). It is rather long because I reload the memory scan counter on each ANTIC mode 7 line. This is necessary for proper fine scrolling. Note also the blank lines inserted into the display list.

ARRTAB (line 2840) is a bit map of the arrows used to display existing orders. One shape is used for each of the four cardinal directions.

The screen data for the text window comes next. An interesting oddity of the text window arises from the history of the program. I originally put the date window in the main text window at the bottom of the screen. Later on I decided for aesthetic reasons to move the date window to the top of the screen. This was accomplished with a simple change in the display list. The upshot of this is that the screen data area for the date window comes after the screen data area for the text window at the bottom of the screen.

Lines 2950-5400 contain the map data. This huge chunk contains all of the terrain. It acts both as display data and as terrain behavior data. I had no need to keep separate images of the map, one for display and one for computations. The same 2K chunk fills both needs. The numbers stored here are the character codes for the ANTIC mode 7 display. The 127 code is a border character used to indicate the edge of the map. For a fuller understanding of how the map works, consult the map image figure and the character set definition.

Line 5410 gives a table called STKTAB. This table is used in decoding joystick values. You may have noticed that I use tables rather heavily. In general, table-driven solutions to programming problems are frequently more desirable than solutions implemented directly in code. They offer far greater flexibility and are normally simpler to program. Furthermore, table-driven routines normally execute faster than code-intensive routines. This point is discussed further in the comments on the interrupt module.

The TRNTAB (lines 5440-5490) specifies the number of subturns expended to enter a given type of square under given weather conditions. A wargamer would call it a movement point costs chart. An entry of 128 indicates that the square in question can never be entered. The operation of this table is a little messy. There are ten terrain types supported, with different values for each of three seasons and two unit types. Thus, there are sixty

entries in this table. Ten entries for infantry alternate with ten entries for armor. Twenty entries for summer are followed by twenty for mud and twenty for snow. The SSNCOD table on line 5430 gives an index into TRNTAB as a function of month. The terrain table is on page 63.

The four following tables (BHX1 through BHY2---lines 5500-5570) specify blocked movement paths. One of the worst problems I encountered in designing the movement algorithms of this game involved blocked movement. It is a simple matter to determine whether motion into a particular type of square, say an ocean square or a border square, is forbidden. Just look at the terrain type and you know that no unit can enter the square. However, there are unfortunate circumstances in which two legitimate squares can be inaccessible to each other. For example, consider the coastline squares of southern Finland and northern Estonia. These squares are adjacent to each other and are all land squares, so a simple-minded program would allow units to move freely from one square to the other. The only problem with this is that the Gulf of Bothnia lies between the two coastlines. Armies cannot walk on water. How can the program detect this condition?

I wrestled with a number of possible algorithms. Most of my early attempts focused on devising an intelligence that would perceive the nature of the situation and act accordingly. I tried all sorts of clever algorithms. All were big and slow. None worked reliably. The scheme I finally chose is remarkably stupid. I found only 11 pairs of squares on the map that caused this problem. I created a table of these square pairs. During movement, the program tests if the unit is attempting to move between a forbidden pair. If so, movement is denied. The table labels stand for Bad Hex X coordinate 1, Bad Hex Y coordinate 1, etc. I'm an old time wargamer and I still think in terms of hexes even though the game uses squares.

This case is an excellent example of the usefulness of table-driven solutions. Logic-driven solutions did not work acceptably, yet the table-driven solution was simple and easy to implement.

The last chunk of RAM reserved by the module is EXEC. This table holds the execution times of the units. The number stored here specifies the subturn in which the unit's next order will be executed.

INTERRUPT MODULE

This module handles all of the I/O for the game. It consists of two routines: a vertical blank interrupt routine which is executed at the beginning of each frame, and a display list interrupt routine which is executed several times during each frame. It is not possible for these two routines to operate together, or for one routine to interrupt the other. The vertical blank interrupt routine reads and responds to the joystick. It performs the scrolling, picks up units and displays the unit data, accepts orders inputs, and displays existing orders. The entire vertical blank interrupt routine must operate under tight timing requirements, as there are only 2000 machine cycles available during vertical blank.

COORDINATE SYSTEMS

The coordinate systems used by this module will drive you nuts. I must admit that I didn't quite know what I was doing as I wrote this module, so whenever I encountered a problem I simply spawned a new coordinate system to deal with it. The result is a maddening plethora of systems and units of measurement. To some extent I can blame the problems on the complexity of handling a constant space that must be addressed in several different ways and can also scroll across the screen. When player-missile graphics, with their independent coordinate system, are thrown in, the situation gets messier.

The first coordinate system keeps track of the cursor against the background of the map. This coordinate system is measured in units of color clocks and pairs of scan lines. Its basic unit is the smallest visual increment on the screen. This coordinate system sees the map as a gridwork 304 pixels high and 360 pixels wide. The position of the cursor in this system is recorded in zero-page addresses CURSXL, CURSXH, CURSYL, and CURSYH. This system is used for managing the scrolling functions.

The second coordinate system is a character-level version of the first system. This system measures the map as a gridwork 38 characters high and 45 characters wide. This system is useful for ascertaining the unit or terrain that the cursor is over. It is maintained with the zero-page variables CHUNKX and CHUNKY.

The third coordinate system maintains player-missile screen coordinates. It uses SHPOSP0 (shadow of horizontal position of player 0) and SCY (shadow of cursor Y-coordinate). This coordinate system is critical for all player-missile manipulations, for it is the only link between the scrolling map and the player coordinates.

The fourth and final coordinate system identifies the position of the map relative to the screen. It is useful for calculations involving the relationship between the map as a whole and the subset that the user sees. It uses the variables XPOSL, YPOSL, and YPOSH.

DATABASE

There are three primary database regions used by the interrupt service routines. The first is the data area on page zero in locations $B0-$BF. I allocated a good portion of my available page zero space for the interrupt routines because they are so time-critical. Most of the values stored here are coordinates. The second database region is the variable storage area on page six. This is used for single-byte variables (not tables) that have lower priority. Most of these values are also coordinates for the various graphics critters that run around on the screen. There are also a variety of counters and miscellaneous variables. The third database area for these routines is the database established by the data module. This consists of tables.

PERSONAL PROGRAMMING STYLE AND CONVENTIONS

A word on my personal programming practices is in order. Every programmer has little conventions about writing code and assigning labels. My conventions are simple. Labelled points that are merely the destinations of branches that skip over code are given meaningless labels. These points are typically not significant entry or exit points, but rather simple highway markers. I have found that trying to cook up descriptive labels for every destination point taxed my limited creative powers too heavily. I therefore adopted the simple expedient of labelling them in sequence X1, X2, X3, etc. up to X99. When I ran out of X's I went to Y, then Z, then A. This does not mean that I used 400 such labels. I wrote many sections of code that I later discarded; I discarded old labels along with old code.

Looping points were always assigned the label LOOPXX, where XX is a two-digit number. When I reached LOOP99, I went to LOOPA, LOOPB, etc. Only major entry points or truly significant program points received meaningful labels.

Variables are usually assigned meaningful names, although sometimes the references are obscure. I prefer to use defining suffixes rather than prefixes. Thus, coordinates will have an X or a Y suffixed to indicate their dimension. The suffixes LO and HI indicate the low order and high order bytes of a 16 bit number such as an address. CNT or NDX normally indicate some type of counter or index. FLG indicates a flag which is set to indicate a condition being met and cleared to indicate the condition not being met.

I always set aside several temporary variables called TEMPthis or TEMPthat. My rule for such variables is absolute: such a variable is always usable for very short-term storage and may never be used for storage exceeding one-half page of source code. I have a short memory.

VERTICAL BLANK INTERRUPT CODE

The VBI routine begins at $7400. It begins with a now-defunct break routine that I used for debugging purposes. This is a valuable tool for any serious programmer. It is prudent to build diagnostic tools into the software to facilitate debugging. This tool is keyed to the joystick in controller jack #2. You can jump out of the program and back into the Assembler/Editor cartridge by plugging a joystick into controller jack #2 and pressing the trigger button. I masked out the code by brute force in the final version. I believe so strongly in the importance of good debugging tools that I did not mask out the routine until the very last minute.

The next section of code handles the handicap option. It reads the console to see if the OPTION key is pressed. If so, it sets the handicap flag and changes the color in the text window. The change is effected by a rather sorry example of self-modifying code. I'm getting finicky about self-modifying code. To be worthwhile it really should do something surprising. This particular application accomplished nothing more than to save me a few minutes of programmer time and destroy any last shreds of respectability the program may have had.

The code beginning at line 2000 determines the state of the button and responds to it. It is tricked by the variable BUTMSK, a button mask set or cleared by the mainline routine to prevent the vertical blank interrupt routine from responding to the button. There are actually two conditions that must be tested. The first condition is the current state of the button, and the second is the state of the button in the immediately preceding VBI. The previous state of the button is recorded in BUTFLG. If both are false (neither the button is down nor was it down earlier) then we immediately proceed to test the joystick. Recall that the button-down condition is signalled by the critical bit being zero. If the button was down but isn't now down, then it was just released, and we must clear the text window and clear any flags and sounds that had been set. We must also unswap any unit in the cursor (more on this later). Finally, we clear out the maltakreuze and the arrow in case they were being displayed.

If the button was down and is still down, (BUTHLD) we must test the joystick for orders. First we check for a space bar being pressed; this would cause the orders to be cleared. Then we move the arrow (lines 2660-3330) until it reaches the maltakreuze. The task of moving the arrow is involved. The unit's orders must be retrieved and the relevant order must be stripped out of the byte. The arrow must be positioned and moved according to the order stored. Furthermore, the display is not done in a single pass of the vertical blank interrupt but in several. The speed of the arrow is set with the operand of the instruction in line 2630. The display of the maltakreuze is a somewhat simpler task (lines 3370-3590). The critical values for this routine are (BASEX, BASEY) which give the player-missile coordinates of the displayed unit, and (STEPX, STEPY) which give the player-missile coordinates of the arrow along its path.

The next button response routine is called FBUTPS and is the response to the first pushing of the button. This one does a lot of work. First it calculates (CHUNKX, CHUNKY) from the cursor coordinates. Then it attempts to find the unit (if any) underneath the cursor. This search alone can consume

1700 machine cycles. If it fails to find a match, the routine terminates. If it finds a unit under the cursor, then it must display the information on the unit.

The display routine is long (lines 4430-5350) but straightforward. The Y-register acts as an index into the text window for all display computations. As the characters are put into the text window, Y is incremented. The important coordinates BASEX, BASEY are computed in lines 5070-5240. These coordinates are exressed in the player-missile coordinate system. They are computed from the cursor coordinates SHPOS0 and SCY. Unlike the cursor, which can straddle map gridlines, they must be properly registered in the map gridwork. The computations in these lines center BASEX, BASEY on the unit.

The HMORDS and WHORDS values are shadowed out of their tables and into special locations on page six (HOWMNY and ORD1, ORD2). This is done in lines 5280-5340; its purpose is to make the orders processing simpler.

The orders input routine follows (lines 5390-6570). It is only entered when the button is held down and has been held down for at least one previous VBI. There are several error conditions which are tested before orders are entered (lines 5410-5660). These include giving orders to Russian units, exceeding eight orders, failure to wait for the maltakreuze, and entering diagonal orders. All errors result in a jump to SQUAWK, the nasty noisemaker routine which displays an error message.

This code also includes a debounce test. Simple switches bounce when first opened or closed, generating a sequence of on-off pulses spanning several milliseconds. A sufficiently rapid polling routine would read this sequence as many switch presses, and would enter multiple presses where the user had only pressed once. A common solution is to set a debounce timer that delays response to the entry for a period of time exceeding the bounce time. Such debouncing is automatically provided by the VBI routine's 16 millisecond polling period, but I inserted a very long debounce (160 milliseconds) anyway. I did this partly out of conservatism (never trust the machine to work properly) and partly to provide some protection against minor mistakes with the joystick. The delay of 1/6th second is not readily noticeable and gives some extra protection against errors.

The next chunk of code (lines 5700-5750) generate a feedback beep in response to the order. Next the new order must be folded into the existing orders. The task is to insert the two-bit order code specified by the joystick into the current orders byte. This requires some bit-twiddling. First we determine which of four bit pairs in the byte to use; the bit pair number is put into the Y-register and saved in TEMP1 (lines 5810-5870). Next we determine which of the two orders bytes should be twiddled. This byte index is either a 1 or a 0 and is put into the X-register (lines 5880-5930). Next, we shift the joystick entry bit pair upward in the byte to correspond to its desired position in the orders byte (lines 5940-6000). Lastly we fold our new order into the orders byte with a fiendishly clever bit of code that I learned from the fellows at Coin-Op (lines 6010-6050). Thanks, Mike and Ed.

The next routine repositions the maltakreuze (lines 6140-6360). This routine is a trivial memory move which moves bytes from the bit map table into the player RAM. It is of little interest.

The scrolling routine comes next. This routine is an adaptation of the routine I first distributed as SCRL19.ASM. If you are interested in the scrolling function of the game, I suggest that you purchase the Graphics/Sound Demo diskette containing SCRL19.ASM from the Atari Program Exchange, for it presents a far more general and better commented program for scrolling than this one. This scrolling routine differs from SCRL19.ASM in several ways. First, scrolling does not occur until the cursor bumps into an invisible wall near the edge of the screen. This is accomplished with some rather simple ad hoc tests in lines 7110, 7440, 7740, and 8220. The values tested were derived by trial and error. Second, the cursor motion is not uniform; it accelerates in the first second of motion. The purpose of the acceleration is to allow fine positioning without sacrificing speed. The acceleration feature is achieved with a very simple bit of code using variables called TIMSCL (time to scroll) and DELAY (delay between scrolls). By comparing TIMSCL with RTCLKL (real-time clock, low byte), the routine can determine when to move the cursor.

Fine scrolling is implemented by storing numbers directly into the fine scrolling registers. Coarse scrolling is implemented by accumulating a value called (OFFLO, OFFHI) and adding it to the LMS operands in the display list. This is done in lines 8650-8770. The final operation of the VBI routine is the preparation for the DLI routine. More on this later.

TABLES AND SUBROUTINES

The table JSTP is used by the artificial intelligence routine. DEFNC is used by the combat routine to figure the defensive value of a terrain type. DWORDS displays a fixed text message pointed to by an index in the accumulator.

SWITCH is an important subroutine. Its inputs are the coordinates of a square CHUNKX, CHUNKY and the identity of a unit CORPS. The subroutine then looks up the character code in the map and switches it with the value stored in the buffer table SWAP. This switches the unit character with a terrain character. The subroutine is used to bring units onto the map. At the beginning of the game there are no units on the map. Each one is brought in by subroutine SWITCH. Whenever the button is pressed and a unit is picked up, the subroutine is called to replace the unit with the terrain character. When the button is released, SWITCH is called again to put the unit back. SWITCH is also used to move units; they are switched off the map, their coordinates are changed, and they are switched back onto the map. SWITCH does not distinguish whether it is placing or removing a unit. A single call switches the unit character with the contents of its SWAP buffer; two calls in a row switch it twice.

The internal operation of SWITCH is simple enough. It computes an

indirect pointer (MAPLO, MAPHI) that points to the beginning of the map row containing the square. The Y-register provides the index to select the proper map byte. The computation of MAPLO, MAPHI is made simple by the fact that there are 48 bytes per map row. Multiplication by 48 is easy: four left shifts, a store, another left shift, and an add.

Subroutines CLRP1, CLRP2, and ERRCLR (lines 9900-10310) are uninteresting routines which merely clear out a player or an error condition and the text window. Nothing very fancy. BITTAB is used to select pairs of bits in a byte. ROTARR is a table used by the artificial intelligence routines to rotate an array. OBJX is a data table used by the artificial intelligence routine.

DISPLAY LIST INTERRUPT SERVICE ROUTINES

The display list interrupt routines are in lines 10450-11340. They are short, but very important. They are a curious mixture of cleverness and stupidity. The stupidity lies in the bucket brigade structure of the DLI execution. There are seven different DLIs serviced by this routine; the proper way to handle this many DLIs is to have each DLI rewrite the DLI vector to point to the next DLI service routine. The technique is described in Section 5 of DE RE ATARI. Instead, I used a DLI counter which is tested, bucket brigade fashion, until control finally reaches the proper DLI service routine. The time wasted by the technique is shameful.

The clever aspect of the code is the way that a DLI is applied to the map, even though the map is scrolled through the screen area. There are two character sets for the map. The switch from the northern character set to the southern one is made at CHUNKY=15. Unfortunately, we cannot simply set a DLI to hit on a specific mode line of the display, for there is no way of knowing if the map will be lined up with the screen properly. Indeed, with vertical scrolling taking place, the point where the transition should take place can be above, below, or on the screen. Obviously some cleverness is required.

The solution I used was to calculate during vertical blank the mode line on which the transition should take place. This value is calculated in lines 8790-8990 and is called CNT1. DLIs are set to hit on each and every mode line in the scrolling window. The DLI code will not be executed until the value of CNT1 indicates that the proper time has arrived. An alternative solution would have been to rewrite the display list every time a scroll is executed. There would then be at most one DLI bit set in the map window. The technique would have saved a great deal of execution time, and so it was the first technique I considered. As it happened, I encountered some difficult problems making the code work properly, so I gave it up and went to the present scheme using multiple DLI's only one of which does the work. The former method should be practicable; I don't know why I couldn't get it working. There's a lesson here: don't hold out for the elegant solution which eludes your grasp when an inelegant but workable solution is accessible. Readers of this document, a few score strong, will know what a klutz I am, but the thousands of happy users are none the wiser.

Another clever trick about these routines is in the timing. You may notice that they do not appear to be in a logical order. They have been carefully ordered to ensure that the most time-critical routines are at the front of the bucket-brigade, and the less critical routines are at the back of the bucket brigade. There is also a careful distribution of labor in the DLIs. Some graphics changes are made several lines before their effects are visible on the screen. This is one way of dealing with the shortage of execution time during a DLI. I make full use of the blank scan lines to perform some DLI chores. Blank lines are ideal for DLI's because no ANTIC DMA occurs during a blank line display; this leaves a full 55 machine cycles for Phase One DLI execution.

Finally, there is a strange example of tight timing in the last service routine. This routine is reached so late that it has almost no time before horizontal blank. I found that the STA WSYNC instruction sometimes produced skipped lines. This indicates that the instruction was being executed just as horizontal blank occurred. Rather than try to force horizontal blank synchrony, I decided to wait it out with a few time-killing instructions. It works.

On page 59 is a diagram depicting the sequence of changes made by the display list interrupts.

FINAL SUBROUTINES AND TABLES

Subroutine DNUMBR (lines 11390-11590) displays a number. It uses the table-driven method described in the notes on the data module. You can see that the code is certainly very clean and fast. Note that I was too lazy to properly encode the screen values properly, so I must perform a CLC/ADC #$10 which should have been done in the data itself. This waste of time in a very time-critical routine is not very consistent with my motivations which led to the use of this method.

NDX is a table used by the artificial intelligence routines to access bytes in an array.

XINC and YINC are tables used for motion. They tell how much to add to the X- or Y-coordinate given a step in any of four directions. I pulled an interesting stunt with YINC that shows how desperate I became for space. YINC is really a table 4 bytes long. The last 3 bytes of YINC just happen to be identical to the first 3 bytes of XINC. So I simply put the two together and cut out three bytes. This is a very dangerous way to save three bytes. If for some reason the two are separated, the program will malfunction in ways almost impossible to debug. Somewhere in the innards of my computer is an ugly green bug chuckling to himself. Someday he'll get me with that one.

OFFNC is a table of values used by the combat routines to evaluate attacks.

MAINLINE MODULE

This module handles the initialization of the game and game turn logic. It brings in reinforcements, figures the dates, seasons, and movement. The combat and thinking modules are subroutines called by this module.

I went through the module stripping out unnecessary equates to make the module somewhat smaller. This was necessary to make all of the source code fit onto a single diskette. You may wonder why I had so many unneccesary equates in the module in the first place. The five modules in this program must communicate with each other, and they do so through the variables in the database. This is impossible if the variables have not been declared in one of the modules. Furthermore, you can waste a great deal of time on bad assemblies discovering that some critical variable has not been declared. The ATARI Assembler/Editor cartridge is slow, and the printer slows things down even more. I solved this problem with a simple scheme. I wrote the modules in sequence. First the data and interrupt modules, then the mainline module, then the combat, and finally the thinking module. Each time I started a new module I created it by taking the previous module and stripping away all the code, leaving only the equates. This insured that each module inherited the complete database equate file.

There were two problems with this technique. First, I had to make certain that changes in an early file such as the interrupt module were properly transferred to all the succeeding modules. Also, equates in later modules sometimes needed to be included in the earlier ones. This problem plagued me throughout the development of the program.

The second problem with the all-inclusive database equate file is that the equate file eventually gets too large. The original equate file for the mainline module was four pages long. By stripping out some (but not all) of the unnecessary equates I was able to reduce it to only two and one-half pages. As you can see, there was a lot of fat. So if you see unused equates in the module equate files, don't get excited.

INITIALIZATION

The mainline routine begins with the initialization routines. The beginning of the mainline routine ($6E00) is the address to which the machine jumps after the program is fully loaded. The mainline routine must first initialize all of the hardware registers and database values. The first segment of code shows a common way to handle initialization of database variables. A table of initial values is kept with the main program. These values are then moved into the database region at the outset of the program. There is one danger in this technique: if for some reason you come along later and rearrange any of the database variables, the initialization code will put the numbers into the wrong places. This code forces you to keep all of your variables that require initialization together. It's not a bad idea to keep all such variables together, but it can be painful when you forget and make changes.

There will always be miscellaneous initializations necessary; with these you have no choice but to write a long string of LDA this, STA there,

instructions. The code is simple but you can waste a lot of bytes this way. One trick for reducing the size of this type of code is to group common initial values together. This is done in lines 1410-1460. Five very different locations all needed to be initialized to zero. Load once and then store five times. A similar method is used for several tables in lines 1480-1570.

The initializations in lines 1620-2060 are all quite straightforward.

MAIN GAME TURN LOOP

The outermost program loop begins on line 2080. The variable TURN is a simple turn counter telling which turn we are on.

First come the calendar calculations. These are simple enough. I add seven to the day, compare with the length of the month to see if a new month has arrived, and correct if it has. There is even a provision for the leap year in 1944 provided in lines 2190-2250. (At the time I wrote this routine I was planning to have the game cover the entire campaign.) With just a little effort the routine could be generalized to handle any leap year.

The tree color trick is executed in lines 2340-2350. Only two lines of code (6 bytes) and 13 bytes of table are required to implement the trick. Color register indirection can be powerful indeed, no?

Lines 2370-2670 put the date information onto the screen. They are simple data move routines with no interesting techniques.

The code in lines 2710-3080 is certainly the most obscure and clumsy code I have written in a long time. The purpose of the code is to figure out what season is in effect and perform any necessary changes related to the season. Unfortunately, I did not take the time to think the problem through. Instead, I just bulled into it, making up code on the fly and patching it together until it worked. The result is a gory mess.

There are four different variables (SEASN1, SEASN2, SEASN3, and EARTH) to tell the state of the season. SEASN1 is used to set the color of rivers and swamps. It holds a $40 for unfrozen water and a $80 for frozen water. SEASN2 tells if we are in fall or spring. This indicates whether the ice-line should move to the south or to the north. It holds a $00 to indicate spring and a $FF to indicate fall. SEASN3 is logically identical to SEASN2 but contains a different value becuase it is used in a different way. It holds a $01 in spring and a $FF in fall. EARTH is the color of the ground, brown for summer, grey for mud, and white for winter.

The code in lines 3130-3700 freezes the rivers and swamps. The algorithm here is interesting and instructive. The critical variables are ICELAT and OLDLAT. ICELAT defines the ice-line, that is, the latitude north of which everything is frozen. OLDLAT is the last turn's value of ICELAT. Everything between the two must be frozen. During spring, everthing between the two must be thawed.

The routine begins by calculating the new value of ICELAT. Notice that there is a random element in the determination of ICELAT. This randomness is leavened by cutting down the size of the random number (AND #$07) and adding a constant (ADC #$07). The result is a number ranging between $07 and $0E.

The code now prepares for the main loop which begins at LOOP40. It initializes LAT and LONG, which are input parameters for subroutine TERR. Then the loop begins. There are actually two loops beginning at LOOP40. The fundamental function of the loop is to sweep through all the map squares in the zone between OLDLAT and ICELAT, checking if they contain water. If so, they are then frozen or thawed, depending on the season. A complication is introduced by the presence of military units. The program must pick up each unit and look underneath to see what terrain is there, modify the terrain if necessary, and put the unit back down. This is gonna get messy, so hang on.

We begin LOOP40 by JSRing to subroutine TERR, an important routine that tells what type of terrain is in a given square. We specify the square's coordinates in LONG and LAT, and it returns the contents of that square in the accumulator. We then examine the terrain type. If it is the wrong type of terrain (mountains, for instance), we skip ahead to NOTCH (as in "no toucha da moichendize, eh!"), which proceeds to the next square in the row. If the square is touchable, we freeze or thaw it with the single instruction ORA SEASN1. Actually, we had already thawed it with the AND #$3F instruction in line 3390; the ORA instruction will freeze or ignore the byte depending on the value of SEASN1. In line 3540 we store the results of our crime. MAPPTR just happens to point to the right place because it is set up by subroutine TERR. Convenient, no?

As I said before, NOTCH moves us on to the next square. This is done by the simple expedient of incrementing CHUNKX. Of course, we must test to see if we have run off the edge of the map. This is done in line 3580. If we have reached the west edge of the map, we must reset CHUNKX and LONG to point back to the east edge of the map. Then we must go one step to the north or south depending on the season. This is done by adding SEASN3, which is either +1 or -1, to the latitude LAT. If we have not reached the vertical edge of the ice region, we loop back to LOOP40; otherwise, we exit the routine.

This is a big, slow routine. You can tell how slow it is by watching the freezing process in the game. You can actually see the iceline moving southward in November. Note that the routine is general enough that it can operate through many different years.

The next routine (lines 3720-3960) brings in reinforcements---units that have not been on the map up to now. This would be a simple routine if it weren't for one small problem: what if the unit comes in on top of another unit? We can't have that, so before we place the unit we have to see if anybody else is already there. This is all done in lines 3760-3840. Lines 3850-3880 notify the player of the arrival of reinforcements. If a unit was not allowed entry onto the board, lines 3910-3940 make sure that he'll get another chance next turn by modifying his value of ARRIVE.

Logistics is handled in lines 3980-4030. It is a simple loop with a subroutine call. The subroutine is inside the combat module; it is discussed in the essay on that module.

POINTS CALCULATION

Lines 4070-4760 calculate the current point score of the player. The algorithm used is involved. There are three factors used in calculating points: 1) how many German muster strength points have been projected how far east, 2) how many Russian combat strength points have been projected how far west, and 3) how many special cities have been captured by the Germans. I feel that this routine is instructive as a good example of a fast, short, and simple routine that imposes reasonable and realistic demands on the player. The importance of the routine is the algorithm, not the coding. The algorithm is optimized for the strengths and weaknesses of the 8-bit processor. Let's look at the implementation closely.

The routine starts by zeroing ACCHI and ACCLO, as these together constitute the point counter, which is sixteen bits wide. It then enters a loop that calculates the points for moving German units east. The longitude of each German unit (CORPSX) is subtracted from a constant value of $30. This value is multiplied by MSTRNG/2 in lines 4190-4280. The multiplication is the stupidest kind: a simple repetitive addition. For single-byte quantities the technique is not too expensive in time. Unfortunately, I did not analyze the problem carefully and so I got the looping backwards. The value of MSTRNG/2 is the loop counter in Y and the value of $30-CORPSX is the added constant. The former value will almost always be larger than the latter, so I should have used the latter as the loop counter. It's always faster to add, say, 50 to itself 3 times than to add 3 to itself 50 times. Oops.

After German points are calculated I begin calculating the effect of Russian points. These will be subtracted from the points accumulated by the Germans in the first loop. For the Russian units, a slightly different algorithm is used. First, the combat strength, not the muster strength, is used. Why? I didn't want to penalize the Germans for moving to the east. Remember, during winter the Germans have a harder time getting supplies as they move further east. So I had to use their muster strength. I also wanted to reward the Germans for Russian units that were still on the board but out of supply. So I used combat strength for the Russians.

The sum of the Russian score is subtracted from the German score in lines 4550-4590. Lines 4600-4680 award point bonuses for capturing cities. A simple loop is used. Two tables drive this routine. One, MOSCOW, is a simple set of flags that tell if the cities have been captured. The other, MPTS, holds the point values for each of the cities. If MOSCOW is set, the number of points assigned for that city are added to the point score.

The final operation associated with point evaluation is to halve the total points if the handicap was used. The operation takes three lines

(4700-4720).

Once the points have been calculated, they must be displayed. This is done in lines 4730-4760 in an operation which by now should be familiar to the reader. Next comes a test for end of game. The termination is not particularly elegant. I simply put an endgame message onto the screen and hang the game up in a loop. I am sure a more elegant termination could have been arranged but I was too lazy to implement one.

Lines 4850-4930 deal with the artificial intelligence routine. They allow the player to use the joystick button (by clearing BUTMSK) and put a prompting message on the screen. Then they jump to the artificial intelligence routine. The program spends most of its time there. It does not return until the player presses the START button. Then the joystick button is masked out by setting BUTMSK and an appropriate message ("figuring move---no orders allowed") is put onto the screen.

MOVEMENT EXECUTION

Lines 4970-5030 prepare the way for movement execution. They initialize the subturn counter TICK and calculate the first execution time of each unit. As mentioned in the player's manual, each turn is broken into 32 subturns. The movement cost to enter a square is expressed in terms of the number of subturns necessary to wait before entering the square. Subroutine DINGO does this calculation. The name DINGO is absolutely meaningless. You should see some of the labels I have used in other programs. When I was an undergraduate doing physics programs I had a penchant for obscene labels. It made sessions with the consulting programmer (especially lady programmers) interesting. The only problem with the idea is that there are a limited number of four-letter words, and I was forced to recycle each word in many different incarnations. Later on I took to using names of animals, fruits, foods, anything. I can't stand acronymic gibberish. I prefer creative gibberish.

Lines 5050-6180 perform the movement. The outer loop beginning with LOOP33 sweeps through all of the subturns. The inner loop beginning with LOOP32 sweeps through all of the units. The inner loop begins by performing the combat strength recovery function. If the combat strength is less than the muster strength, it is incremented. If the difference between the two is large, the combat strength may be incremented again. This ensures that large units will recover combat strength faster than small units.

The most heavily used test is at lines 5180-5190. This determines if the execution time of the unit has arrived yet. If not, the loop proceeds to the next unit.

An interesting stunt is pulled here. Program flow goes through line 5500, which is merely a jump instruction. You may wonder why I didn't insert a jump at the original branch point. I did it to save a few bytes of memory. Any big loop will have a variety of tests that call for abortion of the main loop and immediate procession to the next iteration. If the loop is

considerably longer than 128 bytes, the 6502 branch instructions will not work. The standard response to this problem is to replace the branch with its logical inverse (e.g., BCS with BCC or BNE with BEQ) and follow it with a JMP instruction. This costs three extra bytes. The waste can be reduced by placing a JMP instruction halfway through the loop and having the local test points branch to it. It acts rather like a collecting station for loop abortions. Three bytes are saved for each abortion path.

The remainder of the movement code retrieves the unit's orders, examines the terrain in the destination square, and checks if it is occupied. If it is occupied by a friendly unit, the moving unit must wait two subturns (lines 5450-5490). If it is occupied by an enemy unit, combat occurs and is referred to the combat subroutine at $4ED8. If the unit is allowed to enter the square, either because it was victorious in combat or the square was unoccupied, the code at DOMOVE, lines 5550-6060, is executed.

One last test must be made before actual motion happens. Zones of control are tested in lines 5550-5740. If zones of control do not interfere, the unit is moved by SWITCHing it off the map, substituting the new coordinates as parameters for SWITCH, and SWITCHing it back onto the map. The now-executed order is deleted from the unit's orders queue (lines 5850-5920). A test is made to see if the unit has entered a victory city. If so, the flag for that city is set or cleared depending on the nationality of the moving unit. This is done in lines 5930-6060. Lastly, the execution time until the next order is calculated by DINGO. Then the loop goes to the next unit. When the last unit has had its chance to move, the subturn counter TICK is incremented; when TICK reaches 32 a new turn begins. With this the major loop terminates.

The remainder of the module is devoted to subroutines and tables. STALL is a delay loop that kills time to slow down the action during movement. The debugging routine that follows (lines 6420-6610) links with the debugging routine first mentioned in the interrupt discussion.

TERR is a major subroutine. It sets up a pointer to the map (MAPPTR) on page zero and retrieves the contents of the map at the coordinates LAT and LONG. If the square contains a military unit, it determines the identity of that unit as well as the terrain underneath the unit. Note that TERR returns a terrain code identifier in the accumulator and unit identity (UNITNO). It also returns the terrain identifying code in TRNTYP. It also returns the Z flag of the 6502 processor status register set if the square was indeed occupied. Many calls to TERR are immediately followed by a BEQ or BNE instruction; such calls are attempting to determine if a square is occupied.

TERR does contain an interesting tidbit. Lines 7220-7230 are strictly error flag lines. They put an asterisk onto the screen. If these lines are ever executed a program error has occurred. The error arises when TERR finds a unit character in a square but is unable to find a unit whose coordinates match those of the square. It turned out that this condition could arise from a large number of bugs created by other sections of code. I would never find out about the problem during testing until it was too late to track the bug down. So I put this code in to warn myself. As it happens, there is

another symptom of the bug that is more interesting. The program becomes confused and starts mixing terrain codes with unit codes. The next thing you know, trees, cities, and rivers are marching around the map, fighting battles, retreating, and carrying on in very unterrainlike ways. I tracked down this bug diligently; I believe that it is now quite dead.

Subroutine DINGO is the next subroutine in sequence. It looks up a unit's orders, finds out the terrain in the destination square, determines the delay imposed by that terrain, and stores the delay in the unit's EXEC storage.

Subroutine TERRTY determines the type of terrain in a square, given its character code (TRNCOD). There are several different character types for each terrain type, so some logical analysis of character types is necessary to determine terrain types. It is done with a simple bucket brigade of logical tests. Somehow I am sure that there is a neater way to do this.

ZPVAL is a table of initial values for page zero locations. PSXVAL is a similar table of initial values for page six locations. COLTAB is the table that specifies tree colors for each month of the year. MPTS gives the point scores allocated for each captured city. MOSCX and MOSCY give the coordinates of cities that earn points. TXTMSG is a very simple subroutine that puts a 32-byte text message onto the screen.

COMBAT MODULE

This module handles combat resolution and logistics for the mainline routines. It is nothing more than a set of subroutines called by the mainline routines as needed. Hence, its layout and structure are simple.

The fundamental design of the combat system is not obvious. All combat systems have as their inputs the strengths of the opposing units and the environmental conditions under which they fight. All such systems attempt to determine outcomes as functions of these input conditions. The normal outcomes are reductions in strength and retreats. This game has two types of strength to reduce, which adds some richness to the possibilities.

The unique aspect of this combat system lies in the iterative nature of the combat results system. Instead of trying to compute the outcome of the battle with a single formula, this routine breaks a week-long battle up into many tiny battles which are resolved by simple rules. Each mini-battle can kill only a small number of muster and combat strength points on each side. Thus it is the aggregate effect of many such battles that determines the overall outcome of the battle. The sensitivity and power of the combat results system arises from the statistical behavior of this ensemble of many small battles.

This raises a very important point in game or simulation design: many very advanced functions can be generated using iterative methods with very simple arithmetic. Many people claim that good simulations cannot be done on microcomputers because 8-bit arithmetic is not good enough. While it is certainly true that eight bits are hard to work with, we must remember that eight bits of resolution give better than one percent accuracy in stating a properly normalized number. With imaginative programming these machines can do a great deal of impressive simulation.

SOUND AND GRAPHICS EFFECTS

The module begins with the combat resolution routine at $4ED8. It first clears the flag VICTRY, which is used to tell the mainline routine if the attack was successful. If so, the attacking unit will be allowed to enter the square it attacked. It then checks the attacking unit (ARMY) to make sure that it is not a Finnish unit. Finnish units are not allowed to attack.

The next step (lines 1270-1400) is to create the combat graphic in which the defending unit flashes in solid color. This is done by replacing the unit's original representation on the map with a solid square of color. There must be some logic to determine the nationality of the defending unit (red for Russians, white for Germans). The character used is simply the solid character used for the borders of the map and the open seas. We'll replace the original character later on.

Now we must make the machine gun sound. This is done in lines 1410-1520. My original intention was to create a deep explosion sound, rather like artillery. The result was not at all what I expected, but I liked it so much I left it as it was. The loop in lines 1430-1520 changes the frequency and the volume of the sound produced. The sound is stretched

out with subroutine STALL from the mainline module.

A great deal of time is killed in this loop, deliberately so. When I first ran these routines with no delays the motion and combat happened so fast that I had no chance to observe what was happening. I pushed the START button and saw pieces flying all over the screen like banshees. It was all over in less than a second. I decided that the player would enjoy sweating his turn out, so I put in longer and longer delays until it seemed right.

In lines 1560-1590 I put the defending unit's piece back on the map. The rest of the routine will execute very quickly.

COMBAT RESOLUTION

In lines 1620-1760 I evaluate the factors affecting the defender's strength. There are three: the defender's combat strength (CSTRNG), the terrain that the defender lies in, and the motion of the defender. Terrain evaluation is simple. Terrain can halve, double, or not affect the defender's strength. Notice the test on lines 1690-1700. This protects against overflow. If a large number is doubled too much it can overflow and produce a small number---an unfortunate inaccuracy. I guard against this by monitoring the Carry bit and reloading an $FF if it strikes.

In lines 1740-1760 I implement a very simple rule: defenders who are moving at the time they are attacked have their defensive strength halved. The implementation is about as clean and simple as you can get. This makes an important point about designing with a microcomputer. Some things are trivially simple to do; this operation requires six bytes of code and nine cycles of execution time. Other operations, such as logistics evaluation, are painfully difficult to execute. A designer needs a feeling for what can be done easily and tries whenever possible to work with the grain of his machine rather than against it. Of course, if he/she is to produce anything interesting, he/she must eventually cut across the grain. Doing it well is the hallmark of brilliant design.

In lines 1800-1900 the defender gets to make a first strike against the attacker. The defender's adjusted combat strength in the accumulator is compared with a random number. If it is less than the random number, the defender's pre-emptive strike fails and the attacker makes his strike. If it is greater, the strike succeeds. The attacker suffers the standard loss: he loses one point of muster strength and five points of combat strength. A test is then made to see if the attacker dies or breaks. More on death and breakage later.

On line 1940 we begin the main point of the whole routine, indeed of the whole game. ("The decision by arms is for all operations in war what cash settlement is in trade"---Clausewitz). We figure the attack. The only adjustment made on the attacker's combat strength is the halving of attack strength if the attacker is on a river square. Then we compare the attacker's strength with a random number just as we did with the defender. If the attacker's adjusted combat strength is less than the random number,

the attack fails and the combat routine terminates. If it is greater, then the attack succeeds and many things must happen. First, the defender loses one muster strength point and five combat strength points. That's easy enough to execute (lines 2100-2140).

Next, we must check if the defender dies. If so, we jump to subroutine DEAD, which handles all the paperwork for killing units. This is surprisingly extensive. His combat strength, muster strength, and orders must be zeroed. His execution time and arrival times on the map must be set to nonsense values to preclude his reincarnation. Finally, the body must be removed from the map with subroutine SWITCH.

If the defender did not die, we then test for breakage. This is an important concept in the game. A unit will stand and fight up to a point. At some point morale will break and the unit will collapse and run. Research has shown that this most often happens when some fraction of the unit's strength is destroyed. I chose to measure the intensity of a unit's casualties by comparing the unit's combat strength with its muster strength. If the combat strength falls below some set fraction of the muster strength, the unit breaks. The fraction used depends on the nationality of the unit. German and Finnish units were fairly tough; they don't break until their combat strength falls below one-half of their muster strength. All other units break when their combat strength falls below seven-eighths of their muster strength. The calculations for this are carried out in subroutine BRKCHK, lines 4980-5200. Any unit that breaks forgets any orders that had been assigned to it. Your priorities change when you're on the run.

If the defender does not break, the combat routine terminates. If he does break, he must retreat. This is a complex procedure; it is executed in lines 2210-2750. The basic idea of this code is that the defender attempts to retreat in various directions, but can find his retreat path blocked by zones of control, enemy or friendly units, or open ocean. If any of these events occurs, the unit suffers a penalty and attempts another route. If no retreat path is available the unit suffers heavy losses and remains in place.

An important subroutine for this retreat process is RETRET (lines 2850-3410), which checks for the various conditions that block retreats and exacts the penalty for blocked retreat paths.

If the defender can retreat, the retreat is executed in lines 2500-2630. The victory flag is set to tell the mainline routine that the attacker may indeed move into the defender's square regardless of the presence of enemy zones of control.

The combat routine terminates by incrementing the execution time of the attacking unit.

LOGISTICS

The supply evaluation routine is the next major routine in the module. The basic idea of the routine is to start at the location of each unit and

trace a line from that unit to the appropriate edge of the map without encountering a blocking square. A blocking square is a square containing an enemy unit, a square in an enemy zone of control (unless occupied by a friendly unit), or an open sea square if the unit is Russian. If a blocking square is encountered, the routine must try to trace the line in another direction. It is very easy in such circumstances for a routine to hang up in an infinite loop bouncing between two blocked squares. I precluded this by the clumsy solution of counting the number of blocked squares encountered and declaring the line blocked when the count exceeded a critical value. This critical value depends on the nationality of the unit and the season. There are also seasonal effects on German units. During mud, they receive no supplies at all. During winter, the probability that a German unit will receive supplies depends on how far east the unit has gone. The further east, the smaller the probability. Let's see how all this is done.

The first thing to do is skip units which have not yet arrived on the map (lines 3450-3490). In line 3510 I determine the nationality of the unit. If it is Russian, I skip the weather determination section. Notice the redundant code on line 3530. I blew it. I determine the season in lines 3540-3550 by examining the color of the ground. That's the simplest way to find out the season. If it is mud, there is no supply, period. If it is winter, then I perform a rather odd calculation. I quadruple the unit's longitude and add $4A. This guarantees that the resulting number in the accumulator will lie between 74 and 254. This number becomes the probability (measured against 255) that the unit will receive supplies. Thus, Germans on the west edge of the map have about a 99 percent chance of getting supplies while Germans on the east edge of the map have only a 30 percent chance.

There are two major loops in the logistics routine. The inner loop, labelled LOOP90, attempts to choose a safe direction in which to move from the current square. The outer loop, LOOP91, performs the jump to the chosen square. The inner loop always attempts to jump towards the home map edge (HOMEDR). If that fails, it attempts random directions until it finds a way out or it runs out of tries.

After supplies have been figured, any Russian units in supply have two points added to their muster strength. This is a Russian advantage.

ZONE OF CONTROL

The next routine tests for zones of control. Specifically, it answers the question, "Is there an enemy zone of control extending into square (LAT, LONG) for a German/Russian unit?" The algorithm used is as follows: Examine the square in question to see if it is occupied by an enemy unit. If so, the square is automatically considered in a zone of control. If it is occupied by a friendly unit other than the unit in question, then the square is automatically out of any zones of control. If the square is unoccupied, then we examine all surrounding squares to determine if they are occupied by enemy units. Units in corner squares add one to the ZOC counter. Units in directly adjacent squares add two to the ZOC counter. If the ZOC counter equals or exceeds two, a zone of control is cast into the square.

The routine begins by zeroing the ZOC counter. Then it sets the TEMPR register with a value that identifies the original unit's enemy as either Russian ($40) or German ($C0). Then it examines the contents of the square by calling TERRB. If the square is unoccupied, it branches ahead to A74. If it is occupied, it compares the nationality of the occupying unit (AND #$C0) with that of the original unit (CMP TEMPR). If they are equal, it is an enemy unit and the routine immediately sets the ZOC counter and terminates. If they are unequal, it is a friendly unit and the routine must find out if it is the same as the friendly unit. This is done by comparing coordinates (lines 4410-4460).

If the square is unoccupied, the surrounding squares are examined by a sneaky scheme. There is a table in memory called JSTP+16 that holds jump vectors for a walk around a square. The system works like this:

Starting at X, and proceeding in sequence around X as indicated by the numbers, the sequence of steps is:

(0=north 1=east 2=south 3=west)

0, 1, 2, 2, 3, 3, 0, 0

These are the values seen in the JSTP+16 table, backwards for the 6502's countdown capability. Thus, to execute a walk around the square X, we execute jumps in the directions specified in the JSTP+16 table. The complete walk around the square is executed in lines 4510-4740.

THE IMPORTANCE OF ALGORITHMS

This routine demonstrates a very important principle of software design: the best way to improve performance is to re-examine your algorithms very closely. When I first wrote this routine it was very large and slow. The original algorithm was simple and obvious, but much too slow. It examined each and every unit in turn, subtracting its coordinates from those of the square in question. If the difference of both sets of coordinates was one, the two units were diagonal to each other and I incremented the ZOC counter. If the difference of one pair of coordinates was zero and the other difference was one, then I added two to the ZOC counter. The algorithm is fairly obvious but it required over 200 bytes of code and a very long time to

execute. I tried many of the standard means of speeding it up, but they made it even bigger. I finally grew desperate enough to carefully rethink the entire algorithm. After much brainstorming I came up with the current algorithm, which is subtler but much more efficient. I saved nearly a hundred bytes of code and cut the execution time for typical operations to a third of its previous value. The moral of the story is, rethinking your algorithms will frequently net you far more performance than any amount of clever coding.

THINKING MODULE

This module handles but one task: the artificial intelligence for the Russian player. It has one entry point at $4700 and one exit point at $4C22. It includes several subroutines and data tables for its own use. Thus, this is the most direct and straightforward routine of the entire program. Unfortunately, it is also the most involved routine of the program. It is also the biggest, including about 1.5K of code. To make matters worse, it is almost devoid of comments. This module was one of the best-planned modules in the entire program. For this reason I felt little need to comment on it as I was writing the code. That just makes the task more difficult now.

The basic goal of this routine is to plan the moves of the units. This translates into the specific task of producing values of WHORDS and HMORDS for each Russian unit. Many factors must be considered in computing the orders for each unit. The routine must determine the overall strategic situation as well as the local situation that the unit finds itself in. This will tell whether the unit should think in terms of attack or defense. The overall situation is determined by computing the danger vector. The danger vector tells how much danger is coming from each of the four directions.

The unit must evaluate the four possible directions it can move in. Each direction must be evaluated in terms of the danger vector, the nature of the terrain, the impact of the move on the integrity of the Russian line, the possibility of traffic jams, and the presence of German units. All of the surrounding squares must be evaluated and the best one chosen.

The really difficult aspect of the decision-making process is the necessity of coordinating the moves of all Russian units. The problem is made vastly more difficult by the fact that we must coordinate each unit's possible move with the possible moves of all the other units. The possibilities multiply in a truly mind-boggling manner. My solution was rather esoteric. Imagine the Russian army lying in its positions at the beginning of a turn. Imagine now a ghost army of virtual Russian units, initially springing from the real army, but with each ghost army plotting a path of its own across the map. Each ghost plans its path based on the assumption that the other ghost armies represent the concrete reality that must be conformed to. Thus, each ghost in turn says, "Well, if you guys are gonna move there, I'm gonna move here." One at a time, the ghost army adjusts itself into new positions. This process can continue until each ghost can say, "If you guys are gonna be there, I'm gonna stay right where I am." In practice this situation is almost achieved after only about ten iterations. However, if the player presses the START button, the iterations stop and the ghost army becomes the destinations for the real army. In this way hypothesis is converted into plans.

OVERALL FORCE RATIO

The module begins at line 1680. The first task is to calculate the overall force ratio. This is the ratio of total German strength to total Russian strength, and is a useful indicator of the overall strategic situation. To calculate this number, we must first add up the total German strength and the total Russian strength. This calculation is made in lines

1730-1870. The upper byte of the total strengths is stored in TOTGS (total German strength) and TOTRS (total Russian strength).

The next problem is to calculate the ratio of these two numbers. This is a simple long division. Unfortunately, I was not prepared to do a long division. Such arithmetic takes many machine cycles to crunch and many bytes of code to do properly. The floating point arithmetic package provided in the Operating System ROM did not interest me. So I wrote my own special routine to handle the problem. This is an example of individual crotchetiness, not judicious planning. I probably should have used the floating point package, or at least a decent 16-bit integer arithmetic package, but I was too lazy and impatient.

The first problem I must solve arises from the high probability that the total German strength is going to be very close to the total Russian strength. If I take a straight ratio of the two I will very probably get a result of 1. Since I will have integer arithmetic, my result won't be very sensitive to changes in the total strengths. I solved this problem by arbitrarily multiplying the ratio by 16. It's my program and I can cheat on the arithmetic if I want to.

Unfortunately, multiplying by 16 creates a new problem. Should I multiply the quotient by 16 or divide the divisor by 16? Either approach will have the same effect, and both approaches have the simplicity of being executed with simple logical shifts. But dividing by 16 loses some precision in the quotient, and multiplying by 16 runs the risk of losing the whole number. For example, what if total German strength is 17 and I multiply by 16 by ASLing four times? I don't get 272 for an answer, I get 16. Check it out for yourself.

Here's the clunky solution I came up with: ASL the dividend (line 1950) until a bit falls off the high end of the byte into the Carry bit (line 1960). Put it back where it belongs (line 1970) and then LSR the divisor (line 1980) the remaining number of shifts.

Now I am prepared to do a dumb long division (lines 2070-2140). Load the dividend into the accumulator. Keep subtracting the divisor from it until it is all gone. The number of times you subtract the divisor is the quotient. It's dumb, it's slow, but it works. More important, I can understand it. The final result is stored in OFR, the overall force ratio.

INDIVIDUAL FORCE RATIOS

The next task is to calculate the individual force ratios. The war might be going really well for Mother Russia, but the 44th Infantry Army may not find conditions as rosy if it is surrounded, out of supply, and being attacked by four Panzer Corps. It is necessary to supplement global planning with a local assessment of the situation. This is expressed in the individual force ratios. There are five individual force ratios: Four express the amount of German danger bearing down on a Russian army from the four cardinal directions. The fifth expresses the average of these four.

The fifth is called the individual force ratio (IFR). The other four are called the IFRN, IFRS, IFRE, and IFRW, for the directions they represent.

SUBROUTINE CALIFR

Subroutine CALIFR (lines 8390-9690) calculates the individual force ratios. This is an extensive computation which requires a great deal of time and memory. The fundamental idea behind this subroutine is that danger is a vector, having both a magnitude and a direction. This subroutine determines aggregate magnitude and the aggregate sum of the danger to the unit.

The subroutine begins by zeroing the local variables IFR0, IFR1, IFR2, IFR3, and IFRHI. These correspond to the IFRN, IFRE, IFRS, IFRW, and IFR tables, but are easier to use in the routine. After initializing some coordinate variables, the first large loop begins.

This loop, beginning with line 8520, extends all the way to line 9230. Its purpose is to calculate the directional IFRs, so it is really the meat of the subroutine. It sweeps through each unit, first checking if the unit is on the map (lines 8520-8540). If so, it determines the separation between the tested unit and the unit whose IFR is being computed. It measures this in terms of both the total distance between the two (ignoring Pythagoras) and the X-separation (TEMPX) and the Y-separation (TEMPY). Units further than eight squares away are considered to be too far to be of any local consequence (lines 8680-8690). The range to closer units is halved and stored in TEMPR.

The unit's combat strength determines the magnitude of the unit's threat. We must also calculate the direction to the unit. This is done in lines 8750-9020. These lines test the direction vectors to determine the overall direction to the unit. The result of these tests is a value in X of 0, 1, 2, or 3. This value specifies the direction of the threat.

In lines 9030-9150 we determine the magnitude of the threat. We get the combat strength of the tested unit, divide by 16, and check to see if the tested unit is Russian or German. If Russian, the result is added to the running sum of local Russian strength (RFR). If German, it is added to the running sum of local German strength in the direction specified in the X register. This done, program flow loops back to the next unit in sequence.

The next chunk of code, lines 9250-9320, add up all the danger values from all four directions and leave the result in the accumulator.

The next chunk of code, lines 9350-9570, calculates the final individual force ratio in much the same manner that the overall force ratio was calculated. The dividend is multiplied by 16 (lines 9350-9420), and then the divisor is subtracted from the dividend repeatedly until the dividend is all gone (lines 9450-9510). The count of the number of subtractions equals the quotient. This quotient is averaged with the overall force ratio (lines 9540-9560) and the result is stored in the IFR for the unit. The only remaining function is to move the local directional IFRs to the unit-specific

IFR tables (lines 9610-9680).

Subroutine INVERT is a simple absolute value routine. It takes a value in the accumulator and returns the absolute value of the number in the accumulator. You may have noticed that it was used heavily in the code. By JSRing to INVERT+2, we get the negative value of the accumulator returned.

Back in the main part of the module, we complete the IFR loop by setting the army's current position (CORPSX, CORPSY) to the objective position (OBJX, OBJY). OBJX and OBJY are the coordinates of our ghost armies. This completes the initialization loop. We now enter the main loop of the program.

MAIN LOOP STRUCTURE

The main program loop begins on line 2340 and extends all the way to line 7290. It is obvlously a gigantic loop, and it takes a long time to execute. It is also an indefinitely terminated loop. It does not terminate after a specific number of passes. It keeps looping until the player presses the START key. The main loop sweeps over the entire Russian army. The inner loop sweeps over each unit in the Russian army.

The first task of the loop is to ignore militia armies and armies that are not on the map. Militia are not allowed to move. If an army does not fail these two tests in lines 2360-2420, then the local military situation for the army is evaluated. This is done by comparing the army's individual force ratio with the overall force ratio. If IFR=OFR/2, then the army must be more than eight squares from the nearest German unit. This conclusion can be made from the way that CALIFR calculates the IFR. If the army is far from the front, then it is treated as a reinforcement. If not, it is treated as a front-line unit, and a different strategy is used.

REINFORCEMENT STRATEGY

The job of a reinforcement is to plug weak spots in the line. This requires that the unit be able to figure out where the line is weak, no easy task. The trick is to use the existing Russian front-line units as gauges for the seriousness of the situation at any segment of the front. Where the front is solid, the IFRs of the front-line units will be low. Where the front is weak, their IFRs will be large. So we need merely examine the IFRs of all Russian units, select the largest, and head in that army's direction. Well, not quite. We don't want all the reinforcements heading for the same spot or the beleaguered Russian army will find himself trampled by his rescuers. More important, we need to take into account the distance between unit in distress and rescuer. There is no point in rushing to save somebody several thousand miles away.

The code to do all this extends from line 2470 to line 2870. The section starts by initializing BVAL to the value of OFR/2. BVAL stands for "best value" and is used to store the value for the most beleaguered Russian

army. Then a loop begins at line 2520 which sweeps through all Russian armies, rejecting off-map armies and calculating the separation between the tested army and the reinforcing army. This separation is divided by 8 (lines 2660-2680). I cannot now figure out the purpose of the branch in line 2690. It throws out the tested army if the separation had bit D3 set. A very strange test indeed. Lines 2700-2760 subtract the separation from the tested unit's IFR and compare the result with the best previous result. If the new result is bigger, then this unit has a better combination of proximity and (get this) beleagueredness. This unit becomes the preferred unit. Its value is stored in BVAL and its ID number is stored in BONE (best one). Then we move on to test another unit. When all units have been tested the best one is selected for support. Its coordinates become the objective of the reinforcing army. The job of planning that army's move is done and the routine jumps to the end of the loop (TOGSCN).

STRATEGY FOR FRONT-LINE ARMIES

Front-line armies have a very complex strategy. They must evaluate a large number of factors to determine the best possible objective square. These factors are: the army's IFR, its supply situation, the accessibility of the square, the straightness of the line that would result, the vulnerability to being surrounded, the danger imposed by nearby Germans, the possibility of a traffic jam, the terrain in the square, and the distance to it. Let's take it slowly.

In lines 2990-3050 we perform a simple test to see if the unit should take emergency measures. We ask, is the army seriously outnumbered? Is it out of supply? If either answer is yes, then this army is probably trapped behind German lines and it must escape to the east. It is given an objective square directly east of its current position. It will frantically crash eastward, regardless of the circumstances. It will even attack vastly superior German units in its haste.

This may strike you as pretty stupid. I gave a good deal of thought to the problem and I am convinced that this is the best all-round solution. My first solution was much more intelligent: I had such Russian units run away from the Germans. This normally meant that they ran to the west, straight for Germany. This is not very realistic. It also forced the player to assign large numbers of troops the boring job of tracking down and finishing off the forlorn Russian armies. I considered having cut-off Russians sit down and stay put, but then they would never have any chance of escaping. Quite a few Russians do indeed escape with this system, so I think it has proven to be a successful way of dealing with a difficult problem.

NORMAL FRONT-LINE ARMIES

If an army is not in trouble then it must choose a direction in which to move. The computations for this choice begin in line 3130, with DRLOOP, the direction loop. The critical loop variable is DIR, the direction of movement being evaluated. For the purposes of this loop, DIR takes the following

meanings: 0=north, 1=east, 2=south, 3=west, FF=stay put. This loop answers the question, "Should this army move in direction DIR?" It first determines the square being moved into (lines 3160-3240). The coordinates of this target square are TARGX, TARGY. The square being left is a ghost army square at OBJX, OBJY. The value of this target square is SQVAL. After verifying that the square can be entered (lines 3290-3340), the primary logic begins.

LINE INTEGRITY COMPUTATIONS

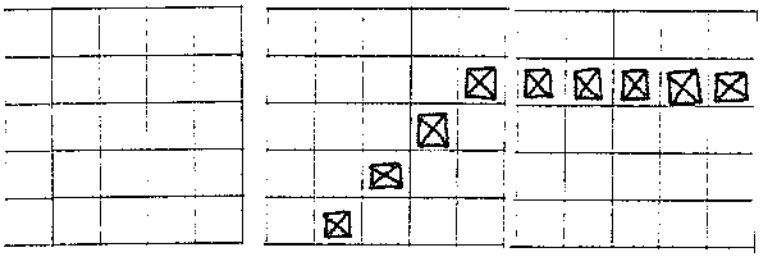
To figure whether a move will result in a solid line or a weak line, it is first necessary to give the computer some image of what that line looks like. I did this by creating two arrays. The first array is called the direct line array and is stored in LINARR. This array is 25 bytes long and covers a 5-by-5 square. The square being tested is always at the center of the big square. The routine will not evaluate the entire Russian line, for that task is impossibly large. Instead, it will treat it as a collection of short line segments and evaluate each segment for desirable configuration.

The big square is addressed by starting at the central square, whose coordinates are TARGX, TARGY, and stepping outward in a spiral from this square. The direction vectors for this spiral path are specified in a table called JSTP. The counter for the steps is called JCNT. The coordinate of a little square being considered within the big square is always SQX, SQY.

The contents of the big square are computed with two nested loops, LOOP56 and LOOP55 (lines 3450-3800). The outer loop steps through each of the 25 squares in the big square (except the central square, which we assume will contain the ghost army). The inner loops sweeps through all Russian armies to see if one's objective is in the square being tested. Note that we check not for the presence of the unit itself (CORPSX, CORPSY) but rather for the intention of the unit to go to the square (OBJX, OBJY). This is how we coordinate the plans of the different armies. If a match is obtained, the muster strength of that army is stored into the array element (lines 3760-3780). We then store the muster strength of the army whose plans are being made into the array element for the central square. When this task is completed we have an array, LINARR, which tells us how much Russian muster strength is in each of the 25 squares surrounding the square in question. We can now examine the structure of this configuration. We will examine it from four different directions: north, south, east, and west. We will keep track of which direction we are looking from with the variable SECDIR (secondary direction).

THE LINE VALUE ARRAY

A very useful tool for examining this two-dimensional array is to construct a one-dimensional representation of its most important feature. This one-dimensional representation will answer the question, "How far forward is the enemy in each column?" A picture might help:



LV ARRAY: 5, 5, 5, 5, 5 5, 4, 3, 2, 1 1, 1, 1, 1, 1

If a particular column is not populated at all, the value in the corresponding LV entry is five.

Lines 3920-4220 build the LV array from the LINARR. The variable POTATO (remember I told you I sometimes used funny variable names?) counts which column we are in. The Y-register holds the row within the column, and the X-register holds the LINARR index. The loop searches each column looking for the first populated square. When it finds one, the row index of the square is stored in the LV arrray. If it finds no populated square in the column, it assigns a value of 5 to the corresponding LV element. The sequence of CPX, BNE, LDX instructions in lines 4060-4220 translate the current row count in X into an index for LINARR and resume the loop. This is the clumsiest kind of code. It is special purpose code, code that is executed only once per condition. During program execution, much of the code is effectively useless, testing for conditions that do not exist. A more elegant solution is called for here. I was too lazy to be elegant; I just slopped the code together.

EVALUATING THE STRENGTH OF THE LINE (LPTS)

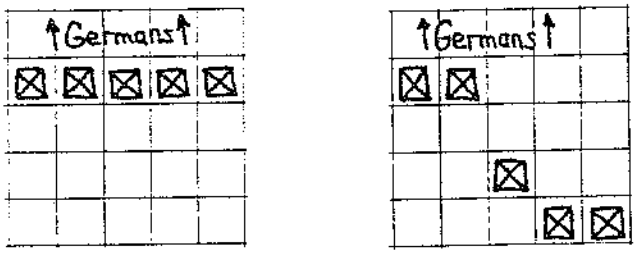
Now that the analytical tools we need are in place, we are ready to begin analysis of the position. We shall analyze the strength of a given line configuration by assigning points to it. We will assign various points for the various features we look for in a good line. These points will be stored in a variable called LPTS. Initially, we shall set this variable to zero and during the course of the evaluation we shall add to it or subtract from it.

The calculation begins on line 4240. We first evaluate the configuration for its completeness. Is there a unit in every single column in the array? For each populated column, we add $28 to LPTS (now in the accumulator). This is done in lines 4240-4320.

We then test if the contemplated presence of our army would fill an otherwise empty column. The test for this is simple and inelegant (lines 4360-4460). An easier way to have done this would have been LDA LV+2/CMP #$02/BNE Y95. It seems so simple and obvious now. In any event, if the condition is satisfied, we add $30 to LPTS.

We don't want to create a traffic jam, so we must evaluate the degree of blocking in this array. This is done by testing the frontmost unit in each column and looking behind it; if somebody is in that square the retreat route of the front unit and the attack route of the rear unit are both blocked. This is undesirable. Subtract $20 points for each such case (lines 4500-4730).

Our next concern is with penetrations. We do not want to create a line configuration which is easily flanked. A picture illustrates the problem.



GOOD BAD

The right arrangement is bad because it allows the enemy to easily penetrate to the rear of the most forward units before engaging the line. This places these forward units in jeopardy. We want a tight, parallel line as in the example on the left. It took me a lot of thinking to translate this concept into terms that the machine could execute. The final result was surprisingly easy to program. It is not so easy to explain. We have five columns in our square. We are going to take each column in turn, calling it OCOLUM, and compare its LV value with each of the other columns. While we are doing this test, we refer to the other column as COLUM. I know, the labelling seems backwards. Forgive me, I was feverish with effort. The comparison is made by subtracting the LV value of the one column from that of the other. If they are equal, there is no problem and we proceed to the next other column. If the latter column is more forward than the first, then we move to the next column; the discrepancy will be handled when the other column is directly tested. If the latter column is more rearward than the primary column, then a penalty must be extracted. The penalty I use is a power of two, one power for each row of discrepancy. The evaluation is done in lines 4880-4990.

We have now calculated the strength of the line and stored it in LPTS. However, the importance of this strength depends on the amount of danger coming from the direction in question. A line which is strong facing north will probably be weak to an attack from the west. We must therefore evaluate the strength of the line in light of the danger vector on the army. I do this by multiplying LPTS by the IFR value for the direction for which the line was evaluated. This multiplication is done in lines 5100-5370. The first 14 lines select the IFR to be used by some more inelegant code. The preparation for the multiplication is done in lines 5240-5280; the multiplication itself is done in lines 5290-5370. As with the long division,

this routine is a triumph of pedestrian programming. To multiply A by B, I add A to itself B times. It is a two-byte add, and only the upper byte (ACCHI) is important to me. I throw away the lower byte in the accumulator.

NEXT SECONDARY DIRECTION

I have now calculated the line configuration value of the square from one direction. I must now perform the same evaluation for each of the other directions. First I increment the secondary direction counter (SECDIR). Then I rotate the array. It is easier to rotate the array in place and evaluate it than to write code that can look from any direction. My code is customized to look at the 25-square array from the north. To look at it from other directions, I simply rotate it to those directions. This is done with an elegant piece of code (at last!) in lines 5480-5580. First I store the array LINARR into a temporary buffer array (BAKARR). Then I rotate it by a pointer array called ROTARR. This array holds numbers that tell where each array element goes when the array is rotated 90 degrees to the right. Thus, the zeroth element of ROTARR is a 4; that means that the zeroth element of LINARR should now be the fourth element. With the rotation done, the program flow loops back up to the beginning of this huge loop.

In developing this code I made heavy use of flowcharts. When I was satisfied with these I then wrote a small BASIC program that performed most of the manipulations in this chunk of code. It took only a few hours to write and test the BASIC code and verify that the fundamental algorithms would work as I had intended. Only then did I proceed to write the assembly code. This shows the value of BASIC: it is an excellent language for tossing ideas together and checking their function. I firmly believe that almost any assembly language project on a personal computer should have several BASIC tools developed just for supporting the effort. I wrote four different BASIC programs as part of the EASTERN FRONT development cycle. They are no longer useful, so I have discarded them.

EVALUATING IMMEDIATE COMBAT FACTORS

It is not good enough to analyze the danger in a square in terms of some obscure danger vector. It is also necessary to ask the simple question, how close is the nearest German unit? The proximity of a German unit will be of great significance to a Russian unit, although the precise significance will depend on whether the Russian is pursuing an offensive or a defensive strategy. In considering the direct combat significance of a square, we must also consider the defensive bonus provided by the terrain in the square.

These factors are considered in lines 5620-6310. After storing the modified line points value into SQVAL (square value), we determine the range to the nearest German unit. This is done with a straightforward loop that subtracts the coordinates of each German unit from the target square's coordinates, takes the absolute value, and adds the two results together. If the resulting range is less than the best previous value, it becomes the new best value (NBVAL).

This range to the closest German unit, when multiplied by the IFR, will give us the specific danger associated with the square. However, IFR is not a signed value; it is always positive. If the Russians are doing well, then IFR will be small but still positive. In such a case the value of IFR*NBVAL would be a measure of the opportunity presented to the Russian, not a measure of danger. Thus, small values of IFR demand that IFR*NBVAL be interpreted differently. The logic to do this is managed in lines 5930-6050. The IFR is subtracted from $F; if the result is greater than zero it is doubled and stored into TEMPR to act as a fake IFR; NBVAL is replaced by 9-NBVAL. The effect of these strange manipulations is to invert the meaning of the code about to be executed. This succeeding code was intended to determine the importance of running from a square. With the inversion, it will also determine the importance of attacking the same square.

The fooled code (lines 6090-6250) begins by checking the square to see if it is occupied by a German. If so, it immediately removes the square from consideration; we don't go around picking fights with Germans when we are the underdogs. Note that this will never happen when the Russians are using offensive strategy. If the square is unoccupied, we add the terrain bonus to NBVAL; this is a crude way of including terrain into the computation. I now think that this was not the correct way to handle terrain.

In lines 6200-6250 I execute one of my disgustingly familiar Neanderthal multiplications. I then add this value to SQVAL (lines 6270-6310).

TRAFFIC AND DISTANCE PENALTIES

The final tasks are to include penalties for traffic jams and long-distance marching. The former is necessary to make sure that Russians don't waste time crowding into the same square. The latter reflects the brutal reality that things sometimes do not go as expected, and so plans that call for armies to march long distances in the face of the enemy are seldom prudent.

The code for making these tests is simple (lines 6350-6870). The first test (lines 6350-6540) is a loop that tests all the other Russians, looking for one that has already chosen this square as an objective. If so, a penalty is extracted from SQVAL. The second test (lines 6580-6870) calculates the range from the army's current position to the target square. If it is greater than 6, the target is unreachable and the square is ruled out; SQVAL is set to zero. If not, 2 raised to the power of the range is subtracted from the SQVAL. With this work done, we have completed our calculation of the value of this square.

FINAL SQUARE EVALUATION

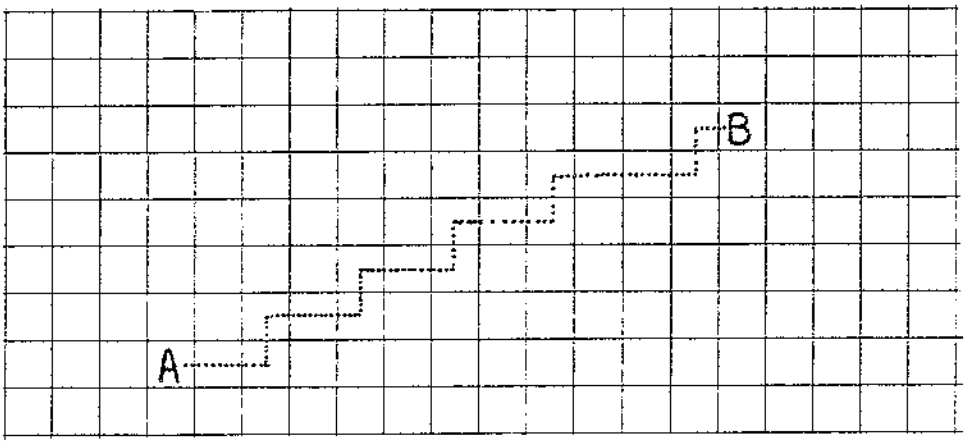
We now compare the value of this square with the best value we have obtained so far (lines 6910-6970). If our new value is better, it becomes the best. If not, we forget it. In either event, we go to the next square

and loop back to the far beginning (lines 6980-7020).

Upon completion of this gigantic process we have obtained a best square. In lines 7040-7150 we make this square our newest objective for this army. We then look at the START key to see if the human player has finished his move. If not, we continue our analysis, evaluating more and more squares without end. If so, we jump to a completely different section.

TRANSLATING TARGETS INTO ORDERS

If the human has pressed the START key, we must convert the targets figured by the previous routines into orders for execution in the mainline routine. This task in done in the remaining section of the module. The fundamental problem solved in the module is a very standard problem in computer graphics. It is depicted in the following diagram:



Starting at square A, what is the straightest path to square B? Specifically, what sequence of single steps will take you from A to B in the straightest possible line? For reasons of computational efficiency, we desire to find the answers without resorting to multiplications or divisions. The problem has been solved in its most general case, and the solution is so powerful that it is easily adapted to circles, ellipses, parabolas, and other curves. Unfortunately, I was unaware of this solution when I wrote these routines, so I had to make up my own, and thereby hangs a tale.

The obvious solution is to compute the slope of the line joining A and B, and then walk from A towards B, measuring the slope generated by each proposed step and comparing this resultant step with the desired slope; if the slope resulting from a proposed step is the closest that can be obtained, then that step is the best. Unfortunately, calculating a slope requires dividing a delta-y by a delta-x, and division is not allowed.

I found my solution in the calendraic system of the Mayan Indians. They never developed the concept of the fraction, and so they had a terrible time expressing the length of the year. Do you know how hard it is to measure the length of the year when you have no number for one-quarter day? They developed a novel solution: instead of declaring that one year is 365 and 1/4

days long, they declared that 4 years are 1461 days long. They refined the method to state that 25 years are 9131 days long. This procedure can be extended to arbitrary precision. Indeed, the Mayans did just that; their measurement of the length of the year was more accurate than the contemporary European value.

The basic idea of the technique is simple: your quantity is divided into a whole part and a fractional part. You can't get your hands on the fractional part, so you keep a running sum, adding both whole and fractional parts until the accumulated fractional parts add up to one; then the whole part will tick over an extra time. That's the event to watch for; it tells you what the fractional part is.

The first step in implementing this algorithm is to calculate some intermediate values. These are HDIR and HRNGE, the horizontal direction from A to B, and the horizontal range (delta-x). VDIR and VRNGE are the corresponding values for the vertical separation. These four values are calculated in lines 7370-7540.

Next we calculate the larger range LRNGE and the smaller range SRNGE, as well as the corresponding directions LDIR and SDIR (lines 7550-7690). Then we prepare some counting variables by setting them equal to zero: RCNT, the number of steps taken, and RORD1 and RORD2, the actual orders assigned. RANGE is the total distance from A to B in non-Pythagorean measure. CHRIS (I was getting desperate for variable names) is the rollover counter. I initialized myself to half of LRNGE.

We now begin the walk from A to B. On each step we shall assume that we should take a step in the larger direction (LDIR). In so doing we add SRNGE to CHRIS; if CHRIS exceeds RANGE then we must instead take a small step in direction SDIR. The figuring for this is done in lines 7830-7940. The code runs fast. The orders that result from this are folded into the main orders in lines 8110-8140, another case of that weird code that first popped up in the interrupt module. If you didn't figure it out then, you might as well figure it out now.

A few more manipulations loop back to finish the walk to point B; then the army's orders are stored and the next army is given its orders until all armies have been taken care of. With that, the routine is complete and it returns to the mainline routine in line 8340. That was simple enough, wasn't it?

NARRATIVE HISTORY

A common misconception among non-programmers is that a program is a static product, something that springs complete from the hand of the programmer. What they do not realize is that a truly original program like EASTERN FRONT 1941 does not leap out of the programmer's mind and into the computer. It starts with an inspiration, a vision that sketches the outlines broad and clear but leaves the individual brushstrokes undefined. The programmer labors long and hard to translate this vision into a cybernetic reality. But the process of converting the pastels and soft shades of the vision into the hard and sharp lines of machine code inevitably produces contradictions between the fine details. As many small ideas crystallize into a single whole, mismatches and discord are frequent. The programmer flits over the design, rearranging ideas, polishing rough edges, and reconciling differences. In this process many of the original ideas are warped and twisted until they no longer resemble their original forms. It is very easy, on examining a program closely, to unearth many of these convoluted elements and conclude that the programmer lacks common sense. In truth, the only way to understand a program is to follow its evolution from start to finish. I have tried to explain some of the odder aspects of this program in terms of historical happenstance. In this essay I will narrate the history of the entire project. I hope that this will make the final product more understandable.

ORIGINS

EASTERN FRONT (1941) began as OURRAH POBIEDA in June of 1979. The original name is Russian for "Hooray for the Motherland!" and was the Russian war cry. It was retained until the last minute; I was finally convinced that the simpler name would sell better.

OURRAH POBIEDA was initially conceived as a division-level game of combat on the Eastern Front. The emphasis of the design was on the operational aspects of combat in that campaign. I wanted to demonstrate the problems of handling division-sized units. The design placed heavy emphasis on mechanical aspects of warfare. Thus, it had strong logistics and movement features. It also had a major subsystem dealing with operational modes. The player could place each unit into one of many different modes such as movement, assault, reconnaissance in force, probing assault, and so on. Each mode had different combinations of movement capabilities, attack strength, and defense strength. There was also a provision for the facing of a unit that allowed flanking attacks.

I wrote the program in BASIC on a PET computer in May and June of 1979. When I got the program up and running on the machine, I quickly realized that I had a dog on my hands. The game had many, many flaws. There were good ideas in it---the logistics routines, the combat system, and the movement system were all very good. But the game as a whole did not work. It was dull, confusing, and slow. I wisely consigned all of my work into a file folder and started on a new design. Someday, when I had shaken off whatever preconceptions were contaminating my mind, I would come back to the game and start over with a fresh outlook.

REBIRTH

Fifteen months passed. I went to work for Atari, programming first on the Video Computer System and then on the Home Computer. In September of 1980 I saw a program written by Ed Rothberg of Atari that finely scrolled the text window. It was a short demo that did nothing other than move the characters around, but it shouted out its potential. I showed it to several other wargame designers and pointed out its implications for our craft. They listened politely but did not act on my suggestion that they use the capability.

Several weeks later I began exploring the fine scrolling capabilities of the machine myself. I took apart Ed's code and wrote a new routine that was more useful for me. I then generalized this routine to produce SCRL19.ASM, a demonstration scrolling module. This module has been spread around in an effort to encourage programmers to use the scrolling. By mid-November I had completed SCRL19.ASM and was finishing up another wargame project. I was beginning to think about my next project. I decided it was time to pull out all the stops and do a monster game, a game with everything. It would be a 48K disk-based game with fabulous graphics. It seemed obvious that the Eastern Front was the ideal stage for such a game. I therefore began planning this new game. In the meantime, I began converting SCRL19.ASM to produce a map of Russia. This map was completed on December 10. It impressed many people, but it was only a map; it didn't do anything other than scroll.

DESIGNING A NEW GAME

Game design is art, not engineering. During December I took many long walks alone at night, sorting through my thoughts and trying to formulate my vision of the game clearly. I sifted through all of my old documents on the PET version of OURRAH POBIEDA, trying to glean from that game the essence of all that was good and all that was bad. Mostly, I thought about what it would be like to play the game. What will go through the head of a person playing my game? What will that person experience? What will he think and feel?

During all this time I never once put pencil to paper to record my thoughts. I deliberately avoided anything that would constrain creative flights of fancy. I also fought off the urge to rush the job. I spent four weeks just thinking. I didn't want to start designing a game that wasn't fully conceived yet.

Then, in January, the vision was clear. I knew (or thought I knew) exactly what the game would be like. I wrote a one-page description of the game. The original document is reproduced at the end of this essay. You will note that it is a surprisingly accurate description of the final product. Also note what is specified. The information given represents my judgment of the critical design and technical factors that would dominate the development

of the game. Note especially the importance I attached to human interface and graphics. This reflects my belief that computation is never a serious problem, but interface is always the primary problem.

PLUNGING INTO THE MORASS

I now began the serious task of implementing the design. At first I proceeded slowly, cautiously. I documented all steps carefully and wrote code conservatively. I didn't want to trap myself with inflexible code at an early stage of the game. First I rewrote the map routine, which involved the data module and the interrupt module. (I decided at the outset that I would need separate modules, as I fully expected the entire program to be too big to fit in one module.) As part of this effort I redesigned the display list and display list interrupt structure. This gave me a much better display. By this time, early February, I was in full gear and was working nights and weekends, perhaps 20 hours per week. I made last changes in the character sets and nailed down the map contents. Next came the unit displays. I wrote the swapping routine and began putting units on the map. They couldn't move or do anything, but they sure looked pretty.

In late February I began work on the input routines. So far everything had gone in smoothly. There had been a lot of work, but most routines had worked properly on the first or second try. My first real headache came when I tried to design the input routines. I had decided that most of the game would be playable with only the joystick. The player would use the START key to begin a move, but otherwise the keyboard was to be avoided. I hung up on the problem of cancelling orders. There seemed to be no way to do it with the joystick. This caused me great consternation. I finally gave in and used the SELECT key for cancelling orders. This may surprise you, for the final product uses the space bar and the initial spec clearly states that space bar would be used. I didn't want to use the keyboard, so I insisted on using the yellow buttons. My playtesters (most notably Rob Zdybel) convinced me to go back to the space bar.

My next problem with the input routines arose when I tried to display a unit's existing orders. I had no end of problems here. My original idea had been to use player-missile graphics to draw some kind of dotted path that would show a unit's planned route instantly. Unfortunately, there weren't enough players and missiles to do the job properly. It could only be done if I used single dots for each square entered. I put the display up on the screen and decided that it did not look good enough. So it was back to the drawing board. The solution I eventually came up with (after considerable creative agony) is the system now used---the moving arrow that shows a unit's path. This takes a little longer but the animation effect is nice.

THE LIGHT AT THE END OF THE TUNNEL

By now it was early March and I paused to consider the pace of the effort. I could see how much effort would be needed to complete the task. I listed each of the remaining tasks and estimated the amount of time necessary

for each. I then realized that the program would not be finished until late June. This was an unpleasant surprise, for I had been planning all along to unveil the game at the ORIGINS wargaming convention over the 4th of July weekend. The schedule appeared to give me very little extra time in the event of problems. I did not like the looks of it. I resolved to redouble my efforts and try to get ahead of the schedule.

MAINLINE MODULE

With the input routines done it was time to work on the mainline module. The very first task was to take care of calendraic functions. I wrote the routines to calculate the days and months; this was easy. Next came the tree color changes with seasons; this was also easy. The first problem developed with the freezing of rivers and swamps during the winter. I was unable to devise a simple way of doing this. I plunged into the problem with indecent haste and threw together a solution by force of effort. The result was impressive, but I'm not sure I did the right thing. It cost me a week of effort, no great loss, and a lot of RAM, which at the time seemed inconsequential because I was still planning on the game taking 48K of memory. Later, when I chose to drop down to 16K, I found myself cramped for RAM, and the expenditure of 120 bytes began to look wasteful.

Fortunately, I emerged from these problems unscathed. I was not tired yet, the project seemed on track and my morale was still high. Morale is important---you can't do great work unless you are up for it.

The next task was movement execution. This went extremely well. I had planned on taking two weeks to get units moving properly; as luck would have it, the routines were working fine after only one week. I was hot!

COMBAT ROUTINES

As March ended, I was beginning work on the combat resolution routines. I had some severe problems here. My routines were based closely on the systems used for the original OURRAH POBEIDA. After some thought, I began to uncover serious conceptual problems with this system. A combat system should accomplish several things. It should provide for attrition due to the intensity of combat. It should also provide for the collapse of a unit's coherence and its subsequent retreat. The routines I had were too bloody. They killed many troops by attrition but did not retreat units readily. I analyzed them closely and concluded that the heart of the problem lay in the fact that combat was completely resolved in a single battle. From this I came up with the idea of the extended battle covering many movement subturns during the week. By stretching out the battle in this way I was able to solve the problem and achieve a much better combat system. I still retained the central idea of the earlier system, which broke a unit's strength up into muster strength and combat strength.

ARTIFICIAL INTELLIGENCE

In early April I turned to the last major module of the project: the artificial intelligence routines. This module frightened me, for I was unsure how to handle it. Looking back, I cannot believe that I invested so much time in this project in the blithe expectation that the artificial intelligence routines would work out properly. I threw myself into them with naive confidence. I carefully listed all of the factors that I wanted the Russian player to consider. Then I prepared a flowchart for the sequence of computations. This flowchart was subsequently rewritten many times as I changed the design.

My biggest problems came with the method of analyzing the robustness of the Russian line. My first approach was based on the original OURRAH POBEIDA method. I started at one end of the line and swept down the line looking for holes. When a hole was found I marked it and jumped onward to the other side of the hole. When the line was fully traced I sent reinforcements to the holes and weak spots in the line. This worked in OURRAH POBEIDA but would not work in the new program. The Russian line in the new program would be far more ragged than in the original game. In some places, the holes would be bigger than the line. In such cases, the algorithm would almost certainly break down.

A new algorithm was required. After many false starts, I came up with the current scheme, which broke the line up into small segments 5 squares wide. This 5-square chunk is then applied to each unit in the Russian army, providing a kind of moving average to smooth the line and bind together the different units in the line. I am very proud of this design, for it is quite flexible and powerful in its ability to analyze a line structure. An interesting aspect of this design is that I originally designed it to handle a smaller segment only three squares wide. After the code had been written, entered, and partially debugged I decided that it would work better with a 5-square width. I modified the code to handle the new width in a few days. The transition was really quite clean. This indicates that I wrote the original code very well, for the ease with which code can be rewritten is a good measure of its quality.

FIRST STARTUP

It was now mid-May. Six months had passed since I had begun the first efforts on the game. One evening, rather late, I finished work on the artificial intelligence routine and prepared to actually play the game for the first time. Many, many times I had put the game up to test the performance of the code, but this was the first time I was bringing the game up solely to assess the overall design. Within ten minutes I knew I had a turkey on my hands. The game was dull and boring, it took too much time to play, it didn't seem to hang together conceptually, and the Russians played a very stupid game.

THE CRISIS

I remember that night very well. I shut off the machine and went for a long walk. It was time to do some hard thinking. The first question was, can the game be salvaged? Are the problems with this game superficial or fundamental to the design? I decided that the game suffered from four problems: There were too many units for the human to control. The game would require far too long to play. The game was a simple slugfest with little opportunity for interesting ploys for the German. The Russians were too stupid. The second question I had to answer was, should I try to maintain my schedule, or should I postpone the game and redesign it?

That was a long night. One thing kept my faith: my egotism. Most good programmers are egomaniacs, and I am no exception. When the program looked hopeless, and the problems seemed insurmountable, one thing kept me going---the absolute certainty that I was so brilliant that I could think up a solution to any problem. Deep down inside, every good programmer knows that the computer will do almost anything if only it is programmed properly. The battle is not between the programmer and the recalcitrant computer; it is between the programmer's will and his own stupidity. Only the most egotistical of programmers refuses to listen to the "I can't do it" and presses on to do the things which neither he nor anybody else thought possible. But in so doing, he faces many lonely nights staring at the ceiling, wondering if maybe this time he has finally bitten off more than he can chew.

I threw myself at the task of redesigning the program. First, I greatly reduced the scale of the program. I had intended the game to cover the entire campaign in the east from 1941 to 1945. I slashed it down to only the first year. That suddenly reduced the projected playing time from a ridiculous 12 hours to a more reasonable 3 hours. I then drastically transformed the entire game by introducing zones of control. Before then units were free to move through gaps in the line at full speed. This single change solved a multitude of problems. First, it allowed me to greatly reduce the unit count on both sides. One unit could control far more territory now, so fewer units were necessary. With fewer units, both players could plan their moves more quickly. Second, Russian stupidity was suddenly less important. If the Russians left small holes in the line, they would be covered by zones of control. Third, it made encirclements much easier to execute, for large Russian forces could be trapped with relatively few German armored units.

My third major change to the game design was the inclusion of logistics. I had meant to have supply considerations all along, but I had not gotten around to it until this time. Now I put it in. This alone made a big change in the game, for it permitted the German to cripple Russian units with movement instead of combat. Indeed, the encirclement to cut off supplies is the central German maneuver of the entire game.

It was about this time that I also committed to producing a game that

would run on a 16K system. I had suspected since April that the entire program would indeed fit into 16K but I did not want to constrain myself, so I continued developing code with little thought to its size. Yet it is hard to deny one's upbringing. I had learned micros on a KIM with only 1K of RAM, later expanded to 5K. I had written many of my early programs on a PET with 8K of RAM, later 16K. I had written programs at Atari to run in 16K. My thoughts were structured around a 16K regime. When the first version of the program ran in May, it fit in almost exactly 16K. I never took anything out to meet the 16K requirement; I simply committed to maintaining the current size.

FRANTIC JUNE

During the first two weeks of June I worked like a madman to implement all of these ideas. The program's structure went to hell during this time. I was confident of what I was doing, and was willing to trade structure for time. I had all the changes up and running by mid-June. It was then that I released the first test version to my playtesters. I also began the huge task of polishing the game, cleaning out the quirks and oddities. This consumed my time right up to the ORIGINS convention on July 3-5. We showed the game to the world then, and it made a favorable impression on those who saw it. The version shown there was version 272. It was a complete game, and a playable game, and even an enjoyable game. It was not yet ready for release.

THE POLISHING STAGES

Two of the most critical stages in the development of a program are the design stage and the polishing stage. In the former, the programmer is tempted to plunge ahead without properly thinking through what he wants to achieve. In the latter, the programmer is exhausted after a major effort to complete the program. The program is now operational and could be released; indeed, people are probably begging for it immediately. The temptation to release it is very strong. The good programmer will spurn the temptation and continue polishing his creation until he knows that it is ready to be released.

Polishing occupied my attentions for six weeks. I playtested the game countless times, recording events that I didn't like, testing the flow of the game, and above all looking for bugs. I found bugs, too. One by one, I expurgated them. I rewrote the zone of control routine to speed it up and take less memory. I made numerous adjustments in the artificial intelligence routines to make the Russians play better. Most of my efforts were directed to the timing and placement of reinforcements. I found that the game was balanced on a razor-edge. A good player would have victory within his reach right up through December, but then the arrival of a large block of Russian reinforcements would dash his chances. I spent a great deal of time juggling reinforcements to get the game tightly balanced.

During this time playtesters were making their own suggestions for the

game. Playtesters are difficult to use properly. At least 90 percent of the suggestions they make are useless or stupid. This is because they do not share the vision that the designer has, so they try to take the game in very different directions. The tremendous value of playtesters lies in that small 10 percent that represents valuable ideas. No designer can think of everything. Every designer will build personal quirks into a game that can only hurt the design. The playtesters will catch these. The good designer must have the courage to reject the bad 90 percent, and the wisdom to accept the good 10 percent. It's a tough business.

DELIVERY AND AFTERMATH

I delivered the final product to Dale Yocum at the Atari Program Exchange around the 20th of August. It was the 317th version of the program. The program went on sale 10 days later. It has generated favorable responses. I was not able to embark on a new project for ten weeks; I was completely burned out. I do not regret burning myself out in this way; anything less would not have been worth the effort.

Eastfront Game Preliminary Description.

Map: 64x64 squares

Unit count: 32 German corps
up to 64 Russian armies

Time scale: "Semi-time" of one week/turn. German enters moves for the next week (meanwhile, computer figures Russian move). When ready, move proceeds in real time.

Human interface: Map window on screen. ~~[illegible]~~ Joystick scrolls map + players. Putting unit under crosshairs, activates it, arrows show Then holding down button while twiddling joystick enters next order. arrows (players) pop onto screen showing orders. Space bar clears orders. Releasing button resumes scrolling.

START button starts ~~the~~ turn.

Colors:

~~Brown~~ background		Brown	
PF0	1	Green	(forests, ~~swamps~~) DLI to ~~red~~ orange mountains
PF1	3	Blue	(rivers, lakes, seas)
PF2	0	Grey	(Germans, cities)
PF3	2	Red	(Russians)
P0-P3		Pink	(arrows)
M0-M3		"	
DLI's		borders	
		red-orange text window	

[not enough color! Use dli's or time-multiplexed color.

CHARACTER SET DESCRIPTIONS

There are three character sets used in EASTERN FRONT 1941. The first is the standard text character set. The other two are graphics character sets used for the display of the map. These character sets allow 64 distinct characters in each set; each character can be presented in one of four colors. The two charts that follow give the critical assignments of characters in the character sets. I do not include the actual bit assignments for each character, as this information is not of primary interest to a designer.

Each chart gives the 6-bit number, which is the number that specifies the shape of the character, and the 8-bit number, which specifies the combination of color and shape that is used in the program. There are a few exceptions. For example, the river characters are normally presented in blue, but during winter they are presented in white to indicate that they have frozen. The character value must be changed to accomplish this. Another case is the solid character, which is normally white for the boundaries. It is also used in its red incarnation to show that a Russian unit has been attacked.

The character descriptions are also cryptic. The river characters are described in terms of the sides of the squares through which the river passes. For this usage, 1 means north, 2 east, 3 south, and 4 west. Thus, a 13 river goes from the northern edge of the square to the southern edge, while a 23 river goes from the eastern edge to the southern edge. River junctions are specified by the three edges that contain rivers.

Coastlines are specified in a similar fashion, with an additional convention. Coastlines are specified directionally, with the land on the right side of the path drawn. For example, a 24 coastline runs from the east side of the square to the west side, with the land on the north and the sea on the south. A 42 coastline would be similar with the land and sea on opposite sides.

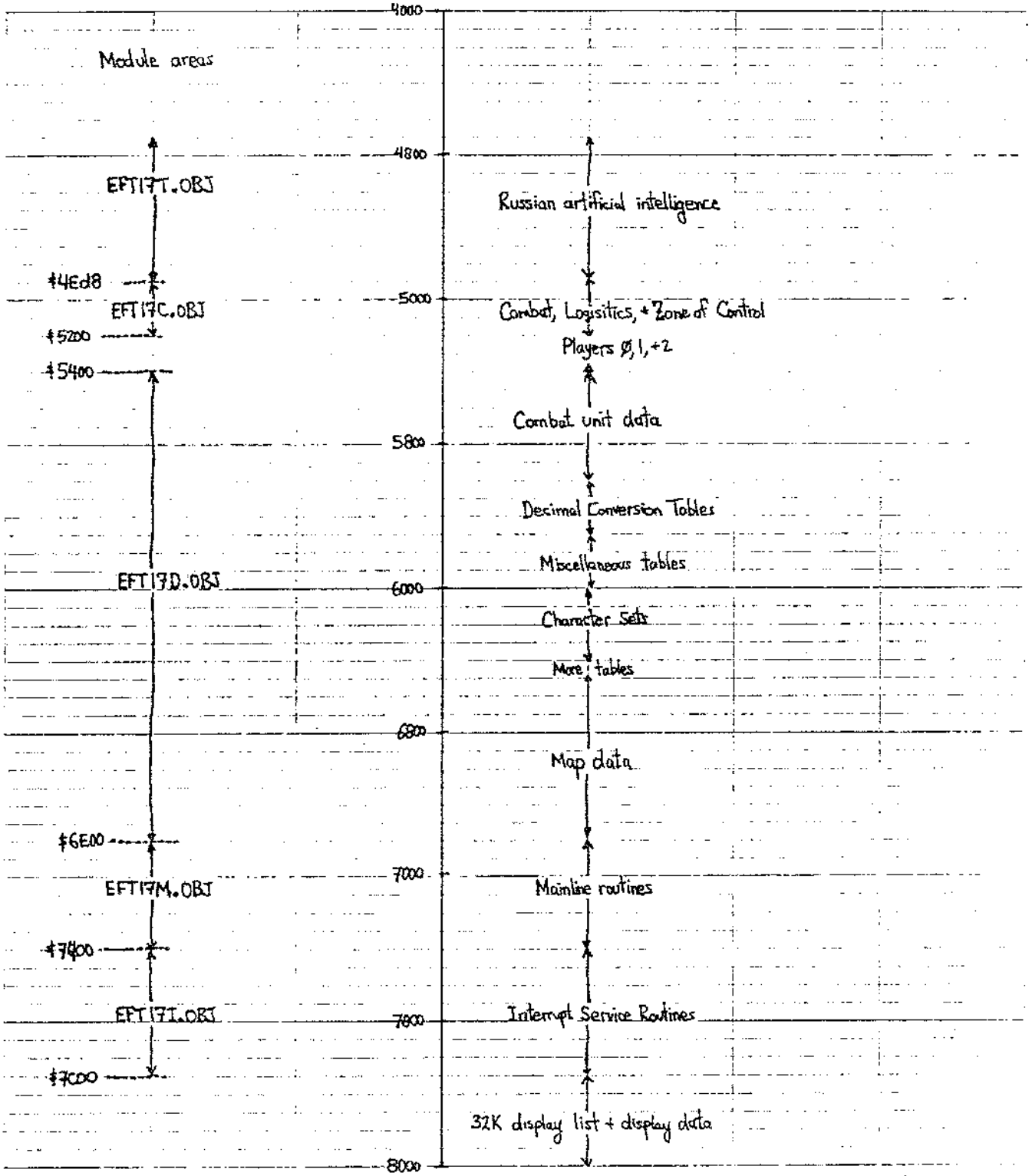
NORTHERN CHARACTER SET SUMMARY

6-BIT #	DESCRIPTION	8-BIT #	6-BIT #	DESCRIPTION	8-BIT #
0	clear	0	32	river 24	160
1	forest	1	33	river 24	161
2	forest	2	34	river 24	162
3	forest	3	35	river 24	163
4	forest	4	36	river 34	164
5	forest	5	37	river 34	165
6	forest	6	38	river 34	166
7	city	71	39	river 34	167
8	city	72	40	river 134	168
9	city	73	41	coastline 31	169
10	city	74	42	coastline 31	170
11	swamp	139	43	coastline 31	171
12	swamp	140	44	coastline 42	172
13	swamp	141	45	coastline 42	173
14	swamp	142	46	coastline 42	174
15	river 12	143	47	coastline 21	175
16	river 12	144	48	coastline 41	176
17	river 12	145	49	coastline 32	177
18	river 12	146	50	coastline 34	178
19	river 13	147	51	coastline 12	179
20	river 13	148	52	Finnish coastline	180
21	river 13	149	53	Finnish coastline	181
22	river 13	150	54	Finnish coastline	182
23	river 13	151	55	Finnish coastline	183
24	river 13	152	56	Finnish coastline	184
25	river 14	153	57	Finnish coastline	185
26	river 14	154	58	Lake Peipus	186
27	river 14	155	59	estuary 1	187
28	river 23	156	60	estuary 2	188
29	river 23	157	61	infantry	125 or 253
30	river 23	158	62	armor	126 or 254
31	river 24	159	63	solid	191

SOUTHERN CHARACTER SET SUMMARY

6-BIT #	DESCRIPTION	8-BIT #	6-BIT #	DESCRIPTION	8-BIT #
0	clear	0	32	river 34	160
1	mountain	1	33	river 34	161
2	mountain	2	34	river 124	162
3	mountain	3	35	Kerch straits	163
4	mountain	4	36	coastline 13	164
5	mountain	5	37	coastline 24	165
6	mountain	6	38	coastline 24	166
7	city	71	39	coastline 24	167
8	city	72	40	coastline 21	168
9	city	73	41	coastline 21	169
10	city	74	42	coastline 14	170
11	swamp	139	43	coastline 14	171
12	swamp	140	44	coastline 14	172
13	swamp	141	45	coastline 41	173
14	swamp	142	46	coastline 41	174
15	river 12	143	47	coastline 23	175
16	river 12	144	48	coastline 23	176
17	river 12	145	49	coastline 23	177
18	river 12	146	50	coastline 32	178
19	river 13	147	51	coastline 32	179
20	river 13	148	52	coastline 34	180
21	river 13	149	53	coastline 34	181
22	river 14	150	54	Crimea	182
23	river 14	151	55	Crimea	183
24	river 23	152	56	Crimea	184
25	river 23	153	57	Crimea	185
26	river 24	154	58	estuary 1	186
27	river 24	155	59	estuary 2	187
28	river 24	156	60	estuary 3	188
29	river 24	157	61	infantry	125 or 253
30	river 34	158	62	armor	126 or 254
31	river 34	159	63	solid	191

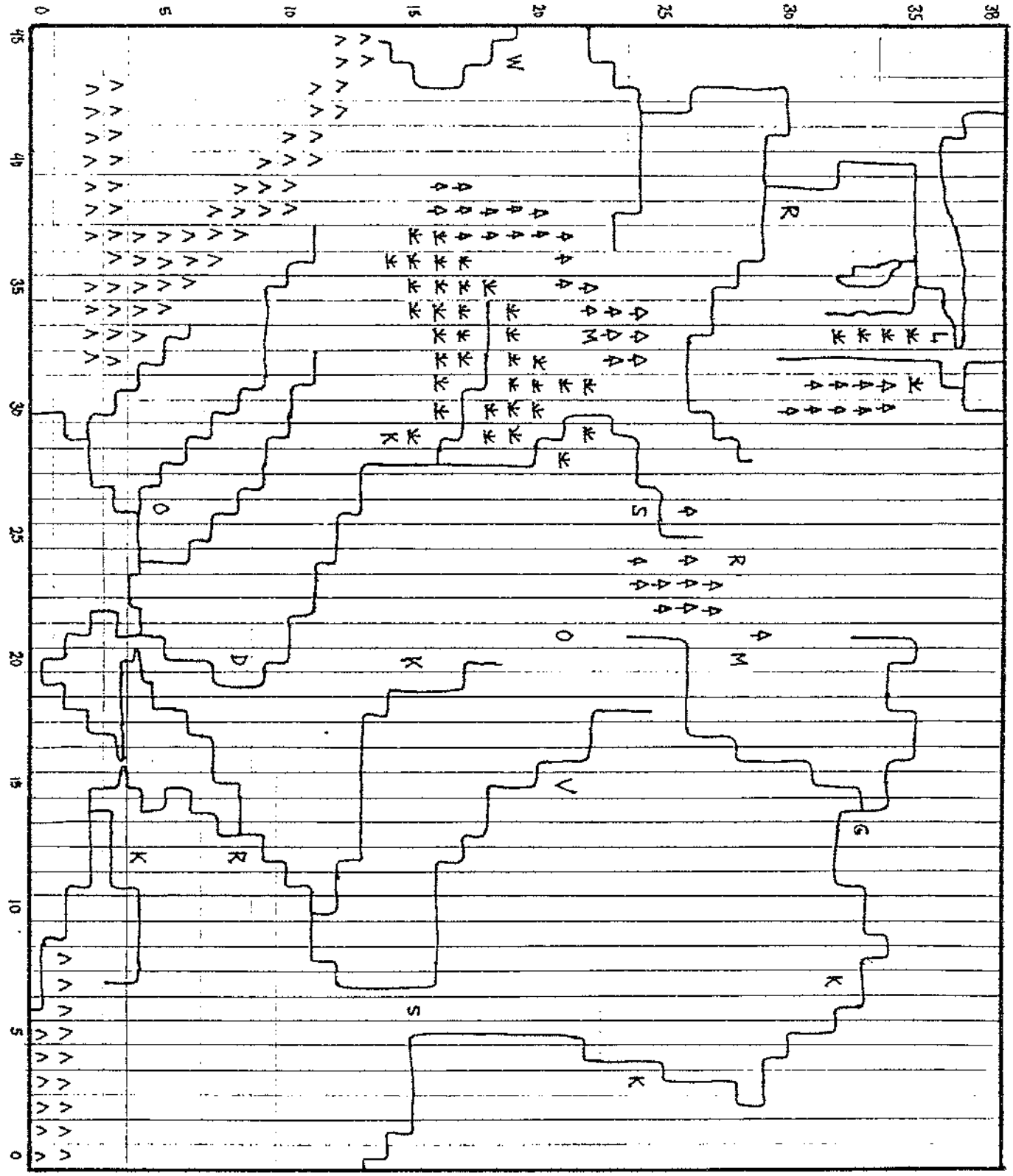
Final Memory Map, Eastern Front Scale is 1 division = 256 bytes Diskette version (Subtract $4000 for cassette)



Other Areas:

Page Zero: $BØ → $CE

Page Six: $600 → $[redacted]6FF



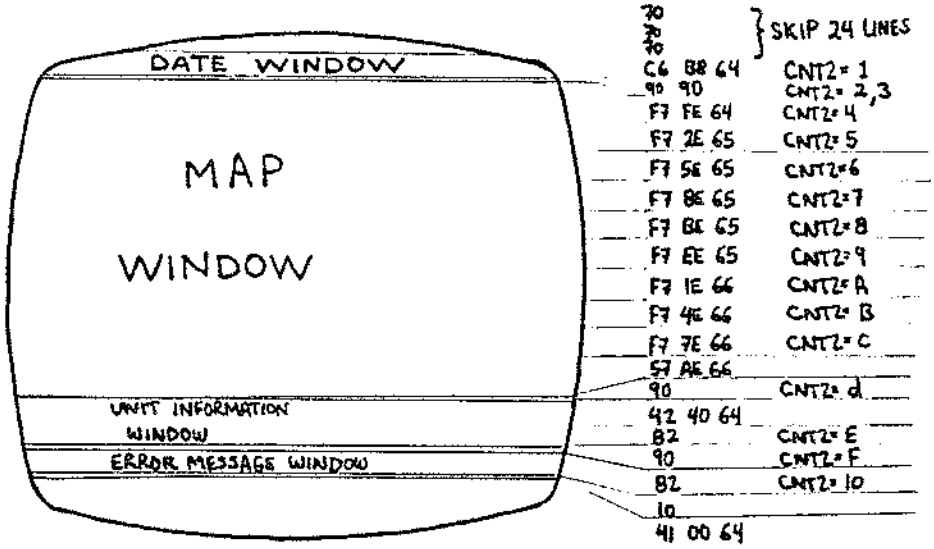
UNIT CHARACTERISTICS

SEQUENCE #		NAME	CORPSX	CORPSY	MSTRNG	SWAP	ARRIVE	CORPT
0	0	INFANTRY CORPS	0	0	0	0	255	0
1	24	PANZER CORPS	40	20	203	126	0	3
2	39	PANZER CORPS	40	19	205	126	255	3
3	46	PANZER CORPS	40	18	192	126	0	3
4	47	PANZER CORPS	40	17	199	126	0	3
5	57	PANZER CORPS	40	16	184	126	0	3
6	5	INFANTRY CORPS	41	20	136	125	0	0
7	6	INFANTRY CORPS	40	19	127	125	0	0
8	7	INFANTRY CORPS	41	18	150	125	0	0
9	8	INFANTRY CORPS	41	17	129	125	0	0
10	9	INFANTRY CORPS	41	16	136	125	0	0
11	12	INFANTRY CORPS	42	20	109	125	255	0
12	13	INFANTRY CORPS	42	19	72	125	255	0
13	20	INFANTRY CORPS	42	18	70	125	255	0
14	42	INFANTRY CORPS	42	17	81	125	255	0
15	43	INFANTRY CORPS	43	19	131	125	255	0
16	53	INFANTRY CORPS	43	18	102	125	255	0
17	3	ITAL INF CORPS	43	17	53	125	255	64
18	41	PANZER CORPS	41	23	198	126	0	3
19	56	PANZER CORPS	40	22	194	126	0	3
20	1	INFANTRY CORPS	40	21	129	125	0	0
21	2	INFANTRY CORPS	41	21	123	125	0	0
22	10	INFANTRY CORPS	41	22	101	125	0	0
23	26	INFANTRY CORPS	42	22	104	125	0	0
24	28	INFANTRY CORPS	42	23	112	125	0	0
25	38	INFANTRY CORPS	42	24	120	125	0	0
26	3	PANZER CORPS	40	15	202	126	0	3
27	14	PANZER CORPS	41	14	195	126	0	3
28	48	PANZER CORPS	42	13	191	126	0	3
29	52	PANZER CORPS	41	15	72	126	255	3
30	49	INFANTRY CORPS	42	14	140	125	0	0
31	4	INFANTRY CORPS	42	12	142	125	0	0
32	17	INFANTRY CORPS	43	13	119	125	0	0
33	29	INFANTRY CORPS	41	15	111	125	0	0
34	44	INFANTRY CORPS	42	16	122	125	255	0
35	55	INFANTRY CORPS	43	16	77	125	255	0
36	1	RUM INF CORPS	30	2	97	125	0	48
37	2	RUM INF CORPS	30	3	96	125	0	48
38	4	RUM INF CORPS	31	4	92	125	0	48
39	11	INFANTRY CORPS	33	6	125	125	0	0
40	30	INFANTRY CORPS	35	7	131	125	0	0
41	54	INFANTRY CORPS	37	8	106	125	0	0
42	2	FINN INF CORPS	35	38	112	125	0	32
43	4	FINN INF CORPS	36	37	104	125	0	32
44	6	FINN INF CORPS	36	38	101	125	255	32
45	40	PANZER CORPS	45	20	210	126	2	3
46	27	INFANTRY CORPS	45	15	97	125	255	0
47	1	HUN PZR CORPS	38	8	98	126	2	83
48	23	INFANTRY CORPS	45	16	95	125	5	0
49	5	RUM INF CORPS	31	1	52	125	6	48
50	34	INFANTRY CORPS	45	20	98	125	9	0
51	35	INFANTRY CORPS	45	19	96	125	10	0
52	4	ITAL INF CORPS	32	1	55	125	11	64
53	51	INFANTRY CORPS	45	17	104	125	20	0
54	50	PZR GRNDR CORPS	45	18	101	126	24	7

SEQUENCE #		NAME	CORPSX	CORPSY	MSTRNG	SWAP	ARRIVE	CORPT
55	7	MILITIA ARMY	29	32	100	253	4	4
56	11	MILITIA ARMY	27	31	103	253	5	4
57	41	INFANTRY ARMY	24	38	110	253	7	0
58	42	INFANTRY ARMY	23	38	101	253	9	0
59	43	INFANTRY ARMY	20	38	92	253	11	0
60	44	INFANTRY ARMY	15	38	103	253	13	0
61	45	INFANTRY ARMY	0	20	105	253	7	0
62	46	INFANTRY ARMY	0	8	107	253	12	0
63	47	INFANTRY ARMY	0	18	111	253	8	0
64	48	INFANTRY ARMY	0	10	88	253	10	0
65	9	TANK ARMY	0	14	117	254	10	1
66	13	TANK ARMY	0	33	84	254	14	1
67	14	TANK ARMY	0	11	109	254	15	1
68	15	TANK ARMY	0	15	89	254	16	1
69	16	TANK ARMY	0	20	105	254	18	1
70	7	CAV ARMY	0	10	93	254	7	2
71	2	TANK ARMY	21	28	62	254	0	1
72	19	INFANTRY ARMY	21	27	104	253	0	0
73	18	INFANTRY ARMY	30	14	101	253	0	0
74	1	CAV ARMY	30	13	67	254	0	2
75	27	INFANTRY ARMY	39	28	104	253	0	0
76	10	TANK ARMY	38	28	84	254	0	1
77	22	INFANTRY ARMY	23	31	127	253	0	0
78	21	INFANTRY ARMY	19	24	112	253	0	0
79	13	INFANTRY ARMY	34	22	111	253	0	0
80	6	TANK ARMY	34	21	91	254	0	1
81	9	MILITIA ARMY	31	34	79	253	0	4
82	2	INFANTRY ARMY	27	6	69	253	0	0
83	1	MILITIA ARMY	33	37	108	253	0	4
84	8	INFANTRY ARMY	41	24	118	253	0	0
85	11	INFANTRY ARMY	40	23	137	253	0	0
86	1	TANK ARMY	39	23	70	254	0	1
87	7	TANK ARMY	42	25	85	254	0	1
88	3	INFANTRY ARMY	39	20	130	253	0	0
89	4	INFANTRY ARMY	39	22	91	253	0	0
90	10	INFANTRY ARMY	39	18	131	253	0	0
91	5	TANK ARMY	39	17	71	254	0	1
92	8	TANK ARMY	39	21	86	254	0	1
93	3	CAV ARMY	37	20	75	254	0	2
94	6	CAV ARMY	39	19	90	254	0	2
95	5	INFANTRY ARMY	39	16	123	253	0	0
96	6	INFANTRY ARMY	39	15	124	253	0	0
97	12	INFANTRY ARMY	40	14	151	253	0	0
98	26	INFANTRY ARMY	41	13	128	253	0	0
99	3	TANK ARMY	41	12	88	254	0	1
100	4	TANK ARMY	39	11	77	254	0	1
101	11	TANK ARMY	36	9	79	254	0	1
102	5	CAV ARMY	34	8	80	254	0	2
103	9	INFANTRY ARMY	32	6	126	253	0	0
104	12	TANK ARMY	35	9	79	254	0	1
105	4	CAV ARMY	30	4	91	254	0	2
106	2	CAV ARMY	28	2	84	254	0	2
107	7	INFANTRY ARMY	25	6	72	253	1	0
108	2	MILITIA ARMY	29	14	86	253	1	4
109	14	INFANTRY ARMY	32	22	76	253	1	0

SEQUENCE #		NAME	CORPSX	CORPSY	MSTRNG	SWAP	ARRIVE	CORPT
110	4	MILITIA ARMY	33	36	99	253	1	4
111	15	INFANTRY ARMY	26	23	67	253	1	0
112	16	INFANTRY ARMY	21	8	78	253	2	0
113	20	INFANTRY ARMY	29	33	121	253	2	0
114	6	INFANTRY ARMY	0	28	114	253	2	0
115	24	INFANTRY ARMY	28	30	105	253	3	0
116	40	INFANTRY ARMY	21	20	122	253	3	0
117	29	INFANTRY ARMY	21	28	127	253	4	0
118	30	INFANTRY ARMY	21	33	129	253	4	0
119	31	INFANTRY ARMY	20	27	105	253	5	0
120	32	INFANTRY ARMY	20	30	111	253	5	0
121	33	INFANTRY ARMY	12	8	112	253	6	0
122	37	INFANTRY ARMY	0	10	127	253	6	0
123	43	INFANTRY ARMY	0	32	119	253	7	0
124	49	INFANTRY ARMY	0	11	89	253	8	0
125	50	INFANTRY ARMY	0	25	108	253	8	0
126	52	INFANTRY ARMY	0	12	113	253	8	0
127	54	INFANTRY ARMY	0	23	105	253	9	0
128	55	INFANTRY ARMY	0	13	94	253	9	0
129	1	GD CAV ARMY	21	29	103	254	5	114
130	34	INFANTRY ARMY	25	30	97	253	5	0
131	1	GD INF ARMY	0	31	108	253	2	112
132	2	GD INF ARMY	0	15	110	253	9	112
133	3	GD INF ARMY	0	27	111	253	10	112
134	4	GD INF ARMY	0	17	96	253	10	112
135	39	INFANTRY ARMY	0	25	109	253	6	0
136	59	INFANTRY ARMY	0	11	112	253	11	0
137	60	INFANTRY ARMY	0	23	95	253	5	0
138	61	INFANTRY ARMY	0	19	93	253	17	0
139	2	GD CAV ARMY	0	21	114	254	2	114
140	1	TANK ARMY	0	33	103	254	11	1
141	1	GD TANK ARMY	0	28	107	254	20	113
142	5	GD INF ARMY	0	13	105	253	21	112
143	2	TANK ARMY	0	26	92	254	22	1
144	6	GD INF ARMY	0	10	109	253	23	112
145	3	TANK ARMY	0	29	101	254	24	1
146	4	TANK ARMY	0	35	106	254	26	1
147	38	INFANTRY ARMY	0	27	95	253	28	0
148	36	INFANTRY ARMY	0	15	99	254	30	0
149	35	INFANTRY ARMY	38	30	101	253	2	0
150	28	INFANTRY ARMY	21	22	118	253	3	0
151	25	INFANTRY ARMY	12	8	106	253	3	0
152	23	INFANTRY ARMY	20	13	112	253	3	0
153	17	INFANTRY ARMY	21	14	104	253	3	0
154	8	MILITIA ARMY	20	28	185	253	6	4
155	10	MILITIA ARMY	15	3	108	253	6	4
156	3	MILITIA ARMY	21	3	94	253	4	4
157	5	MILITIA ARMY	20	3	102	253	4	4
158	6	MILITIA ARMY	19	2	98	253	4	4
159	0	INFANTRY ARMY	0	0	0	255	32	0

Display List Interrupt Sequencing

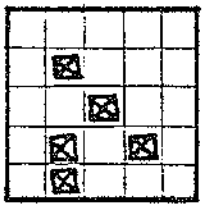


CNT2 value	Register changed					
	CHBAS	COLBAK	COLPFØ	COLPF1	COLPF2	COLPF3
0 (vert. blank)	E0	B0	6A	0C	94	46
1	60	1A	TRCOLR	–	–	–
3	–	EARTH	–	–	–	–
3≤CNT1≤d	62	–	28	–	–	–
d	E0	–	–	–	22	–
E	–	8A	–	–	–	–
F	–	–	–	00	3A	–
10	–	d4	–	–	–	–

Point System for Artificial Intelligence

A. Line Points - LPTS

(Values for this example)



LINARR = 0, 0, 0, 0, 0, 0, M1, 0, M2, M3
0, 0, M4, 0, 0, 0, 0, 0, M5, 0
0, 0, 0, 0, 0

LV = 5, 1, 2, 3, 5

+ 40 points for each occupied column	LPTS = 120
+ 48 points if central column is otherwise empty	LPTS = 168
- 32 points for each front unit whose retreat is blocked	LPTS = 168
-2^{Δ} points for each column pair, where Δ is the difference in LV (iff $\Delta > 0$)	

Δ values for this example:

Lead Column \ Lag Column	1	2	3	4	5
1	–	<	<	<	0
2	4	–	1	2	4
3	3	<	–	1	3
4	2	<	<	–	2
5	0	<	<	<	–

Hence total penalty in this example is:
$2^4 + 2^1 + 2^2 + 2^4 + 2^3 + 2^1 + 2^3 + 2^2 + 2^2 = 64$

LPTS = 104 final

B. Accumulated Points [ACCLO, ACCHI]

$$ACC = \sum_{SECDIR=0}^{3} LPTS_{SECDIR} * IFRX_{SECDIR}$$

C. Computation of Square Value [SQVAL]

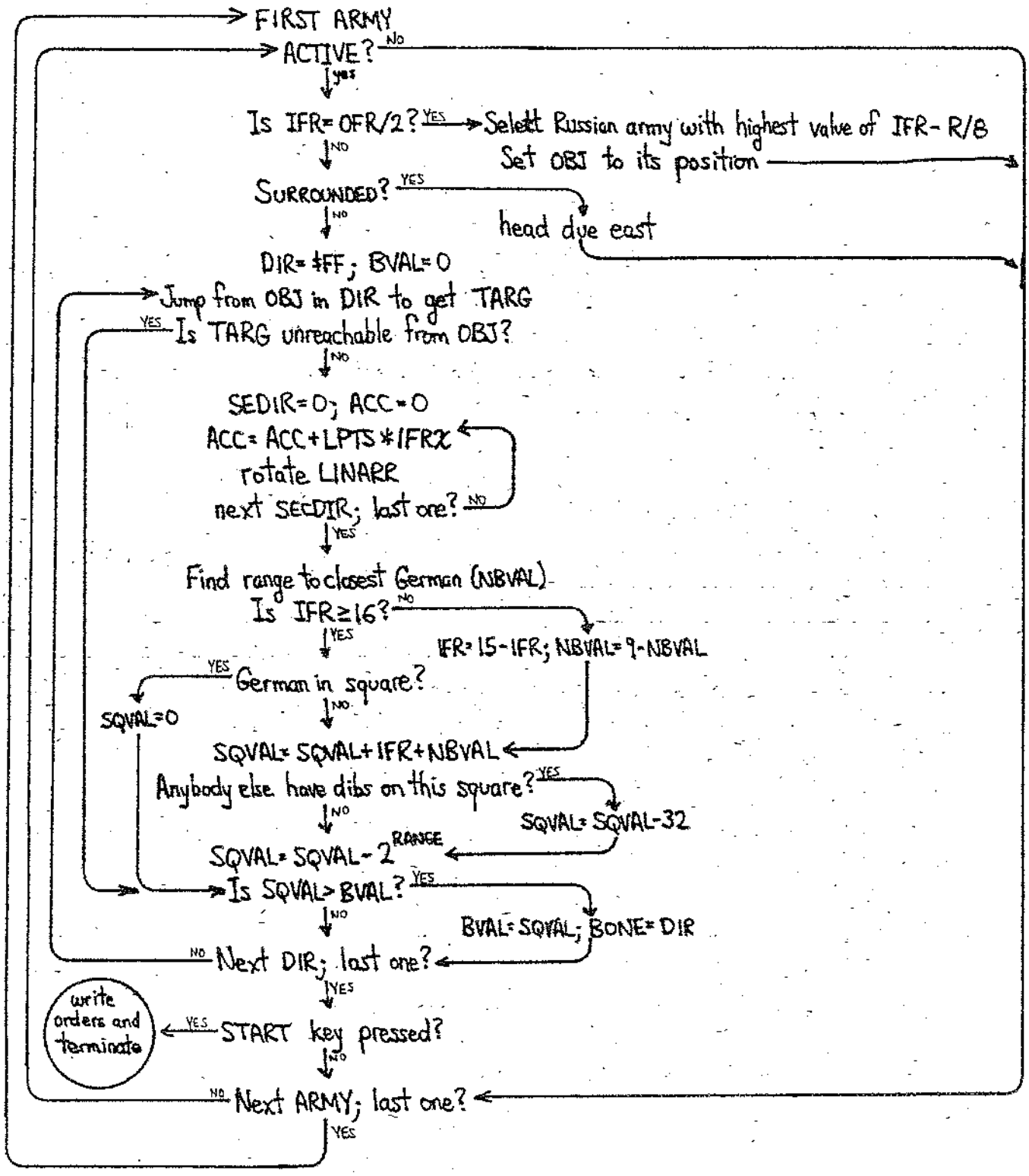
Start with SQVAL = ACCHI
Determine NBVAL, range to nearest German

If IFR≥16 (defensive strategy):
    If NBVAL=0 (i.e., German in square), then SQVAL=0, exit?
    Add IFR* (NBVAL+defensive bonus) to SQVAL
If IFR<15 (offensive strategy):
    Add 2*(15-IFR)* (9-NBVAL+defensive bonus) to SQVAL

If somebody else has dibs on this square, SQVAL= SQVAL-32

SQVAL= SQVAL- 2^R where R is range from unit to objective

TUMBLECHART FOR RUSSIAN MOVE (CENTRAL PORTION)



TERRAIN VALUES

TERRAIN TYPE	SUBTURN DELAY						DEFENSIVE VALUE	OFFENSIVE VALUE
	DRY		MUD		SNOW			
	Inf	Arm	Inf	Arm	Inf	Arm		
Clear	6	4	24	30	10	6	2	1
Mountain/Forest	12	8	30	30	16	10	3	1
City	18	6	24	30	10	8	3	1
Frozen Swamp	0	0	0	0	12	8	2	1
Frozen River	0	0	0	0	12	8	2	1
Swamp	18	18	30	30	24	24	2	1
River	14	13	30	30	28	28	1	2
Coastline	8	6	26	30	12	8	1	2
Estuary	20	16	28	30	24	20	2	1
Open Sea	127	127	127	127	127	127	0	0