ULTRA DISASSEMBLER

For the Atari 400 / 800 / 1200

by Ralph Jones

Published by

Adventure International

A Subsidiary of Scott Adams, Inc.

BOX 3435 • LONGWOOD, FLORIDA 32750 • (305) 862-6917

TABLE OF CONTENTS

INTRODUCTION

Adventure International is pleased to introduce the latest in the Ultra Utility Series - Ultra Disassembler.

Ultra Disassembler is a utility which enables you to analyze existing machine language programs and modify them to fit your own requirements. Of course, it does what all disassemblers do — it translates machine language into assembly language. What makes Ultra Disassembler unique is that it also formats the output into highly readable pseudo-source code with standard system labels where appropriate, and writes it to disk in a form suitable for editing and reassembly with all of the major Atari disassemblers (Atari Macro, Atari Assembler/Editor, DATASM-65, EASMD, etc.).

To make best use of Ultra Disassembler, you need a working knowledge of 6502 assembly language and the Atari Operating System. The program is written on the assumption that after disassembling a program you will reassemble it with the Atari Macro Assembler (hereafter we'll refer to it by its program file name AMAC), because the full-symbol reference map this assembler produces is an invaluable aid in analyzing programs. Along the way, instructions will be given for adapting it to all the other popular assemblers.

READ THE MANUAL!

Before you try to use the program, please read this manual carefully; then go to the 'Getting Started' section and follow the example in which you will disassemble and recreate the DUP.SYS file in the DOS.

SYSTEM REQUIREMENTS

The minimum equipment needed to run the Disassembler is:

(1) An Atari 400/800/1200 computer with at least 32K RAM
(2) One Atari 810 disk Drive (or other 810-compatible Drive)
(3) A video monitor or TV set (there is no use of color in this program, so a black-and-white monitor is adequate.)

OPTIONAL EQUIPMENT

The following optional equipment will make Ultra Disassembler even more useful:

(1) 48K RAM
(2) Additional disk Drives
(3) A printer and its associated interface adapter such as the Atari 825 or Epson MX-80 with the Atari 850 Interface Module

Additional RAM increases execution speed, since the Disassembler uses all the empty RAM space as an input buffer. Additional drives will allow you to run a full disassembly with less disk swapping, and enable you to reassemble programs too large to fit on one disk (if your assembler supports file linking). A printer will deliver a listing of a disassembly similar to an assembly listing so you can study the logic flow of a program at your leisure.

Finally, a copy of Adventure International's DISKEY may prove extremely useful.

Chapter 1
Getting Started

Here comes some advice you've seen before, just in case any of you missed it the first time:

MAKE A BACKUP COPY!!!

Format a disk, write DOS 2.0S to it, and then copy file DISKDIS* from the distribution disk to the new one with DOS Option O. Put the original Ultra Disassembler disk in a safe place and use the newly created disk as your working copy. Also format several blank disks for use as pseudo-source code output disks.

Next, boot up DOS 2.0S (if you haven't done so already), and use Option L (Binary Load) to load file DISKDIS*. Ultra Disassembler will start up automatically and display a message asking how many disk drives you're using — enter 1 or 2 and press RETURN.

You'll be asked whether the disassembly is to be done from a file, a sector list or memory. For this example we'll use your DUP.SYS file, so enter F RETURN. Now you'll be asked for an input (object code) file name; enter D1:DUP.SYS RETURN. Now you'll be asked for an 'output file name'; enter D1 (for 1 Drive; D2 for 2 Drives):DUP, but DO NOT add a file name extender (we'll see why soon), and press RETURN. You will be prompted to insert the input disk (in this case, any disk with DUP.SYS on it) in Drive 1, and, if you're using two drives, the output disk (a blank one) in Drive 2. Do whichever is appropriate and press RETURN. If you have one drive, you'll be asked repeatedly for the input or output disk; whenever you get the prompt (along with a console bell to get your attention), insert the appropriate disk and press RETURN.

You'll hear the usual disk-read beeps from the monitor until the machine has read the input file. The message 'CREATING LABELS' will appear repeatedly on the screen, as the program examines every instruction in the program and creates an assembly label for each address referenced by an instruction.

Next, you'll hear some disk-write sounds, and the disassembled code will begin scrolling up the screen. At this point, the program is disassembling instructions one at a time and writing them to the output disk. When it runs out of space on the output disk (this happens with object files longer than about 10K; DUP.SYS will only use one disk), you will be asked for a fresh disk. Remove the present output disk, insert a blank one in its place and press RETURN.

Whenever you want to take a close look at the output code, press the START button — the program will freeze until you press START again. The code you see is identical to what goes to the output disk, except that the screen version has a memory address at the start of each line, just like an assembly listing.

If you have a printer available, you can also route the screen output to it with the OPTION button. The printer will start when you press OPTION and stop when you press it again.

When the last instruction has appeared on the screen, you'll get the message 'DISASSEMBLY COMPLETE'. Reload DOS. (As long as a disk containing DOS is in Drive 1, you can do this just by pressing SYSTEM RESET.) At this point, the output disk contains a complete disassembly of DUP.SYS. Let's see what it looks like.

Run a directory of the output disk. You'll see files named DUP.000 through DUP.003, each of them 132 sectors long except the last one. This is how DISKDIS keeps the 'source' files within the size limits of your program text editor; output files are limited to 4000(hex) bytes long and only four files are written to each output disk. This leaves enough room on the disk for you to do a moderate amount of editing on the 'source' files before you reassemble the program. It also explains why you were cautioned not to give an extender with the output file name: DISKDIS adds its own numerical extender to identify the file.

Now look at the contents of file DUP.000. If you didn't select printer output, there are three ways to do this:

* Load the file into the program text editor you use with your assembler
* Copy it to device P: with DOS Option C if you have a printer
* Copy it to device E: with DOS Option C and watch it go by on the screen, pressing CTRL 1 whenever you want to stop and look.

The file will begin with a series of equate instructions assigning addresses to the names of various Atari system locations, such as:

```
MEMTOP = $02E5
```

Each of these instructions defines the address of a system location which is referenced by an instruction in the program (except for some 'phony' locations which we'll look into later).

Next there will be a list of equate instructions for locations which were referenced on memory page 0 (notice that these labels begin with a Z instead of the L used in other nonsystem labels). The need for this zero-page equate list will be discussed in the section on the 'Zero-Page Address Problem'.

The next item after the equate instructions is

```
ORG $1F0C
```

This sets the program origin to the start address in the binary file header; you'll find another one wherever the file skips over a section of memory. (If you aren't used to AMAC, ORG is equivalent to *= in most other assemblers.)

Now we come to the actual CPU instructions. Notice that address references in the instructions are given as labels and that the referenced instructions themselves are labeled. The labels begin with L1 and run up to L1023 (except zero-page labels which have a Z instead of an L); if the object program uses more than 1023 addresses, further instructions will contain absolute addresses.

Scan through the file and you'll see one of the most powerful features of Ultra Disassembler: addresses that represent documented Atari Operating System locations are labeled with their standard system mnemonics, such as

```
STA DOSVEC
```

The better your understanding of the OS, the more you'll appreciate this in analyzing a program. Also notice that each instruction that references a label is followed by a number set off by a semicolon, indicating it's to be interpreted as a comment; this is the absolute address represented by the label, for your convenience in tracing the program flow.

Next, look at the first four instructions after the ORG instruction:

```
EOR XMTDON
.BYTE $9B
ADC L1,X
.BYTE $53,$4B
```

Looks like garbage, doesn't it? Now the machine code that produced this was

```
45 3A 9B 7D 44 49 53 4B
```

If we convert this to ATASCII characters we get

```
E:(EOL)(CLR SCREEN)DISK
```

which is obviously text information. Here we see one of the fundamental limitations of all disassemblers: the program does not analyze the logic flow of the object program, so it has no way of knowing data from instructions. When it saw 45 3A, it was able to interpret it as an EOR instruction; when it saw 9B, it assumed it had to be data (since there is no opcode 9B), and generated a .BYTE instruction. In both cases — and this was the overriding rule in designing this program — THE OUTPUT INSTRUCTION WILL REASSEMBLE TO AN EXACT DUPLICATE OF THE ORIGINAL MACHINE CODE. For your convenience in analyzing the program, you could use your program editor to alter these instructions to

```
        .BYTE 'E:',$9B,$7D
        .BYTE 'DISK'
```

and get the same result with more clarity.

This explains why some of the system equate instructions at the beginning of the disassembly file refer to system locations that aren't really used: they result from data being interpreted as instructions. In order to get the file to reassemble, these equate instructions must be present; otherwise the 'phony' instructions would be rejected as containing undefined labels.

Now look at the very end of this file. The last instruction is

```
        LINK Dn:DUP.001
```

where n is the number of the drive your output disk was in. This will cause the assembler to move on to the next file when you do a reassembly.

Next, look at the final output file (DUP.003). It consists mostly of another series of equate statements, this time giving values for numbered labels; this list began back in file DUP.002, after the last instruction in the program. These are the 'external equate statements': i.e., those addresses that were referenced from within the program but do not represent system locations or addresses internal to the program. Again, many of these addresses are 'phony' locations, resulting from data being interpreted as instructions. Note that there are no zero-page locations in this list; all the zero-page locations were equated at the beginning of the disassembly, as mentioned earlier.

Now look at the last line in the last output file:

```
        .END $2075
```

This is the assembly-end instruction. AMAC expects to see a run address in this instruction, and appends it to the end of the file when it assembles; if it doesn't see one, it will assume $0000 and append that. Since the Disassembler has no reliable way of knowing what the intended run address is, it simply uses $2075, which is the DOS menu entry point. Thus when a reassembled file is loaded from DOS the system will return you to the DOS menu. To see what the 'real' run address is, we can look at the last few instructions before the beginning of the external equate instructions in file DUP.002:

```
        ORG $02E0
        ADC ICHIDZ,X
```

The machine code equivalent of this is simply the two bytes 75 20, beginning at location $02E0, which is the run address location in the FMS. Thus the program file already had a run address of $2075 appended, and the reassembly will have it appearing twice (which doesn't hurt, since only the last one counts). If the run address was something else (which it will usually be for programs other than DUP.SYS), the reassembly would have two different run addresses appended, and the last one would rule; you could get rid of it by deleting the last 6 bytes of the assembled file, and the program would execute automatically on loading. If the run address wasn't present in the program, it would have been up to you to find out what it was and alter the .END instruction if you wanted the reassembled program to autorun.

Now it's time to reassemble what you've disassembled. If you're using AMAC, just request an assembly of file DUP.000. For any other assembler, you'll have to modify the source files according to the instructions in the section on reassemblies. Use a disk with DOS on it to receive the object file. Now delete the existing DUP.SYS file and rename the newly assembled object file to DUP.SYS. Turn the computer off, then on again and watch the DOS boot up. If you did everything right, the DOS menu will be displayed.

Now you have a DUP.SYS file containing all the same information the original had, but it isn't quite identical. Did you notice a hesitation after every 2 or 3 disk-read beeps as it loaded? That's because the assembler writes an object file as a series of appended binary segments with a new file header every hundred or so bytes. In order to get a compact, fast-loading file, you'll have to load the file into memory and save it back to disk. For this special case with the DUP file, you could cut certain corners, but let's do it as if this were any old program file. Look at the assembly listing and see if the assembled code overlays any of the DUP code (of course it does in this case), and create a MEM.SAV file on the disk if it does. Now load the file into memory with DOS Option L and the 'no-run' sub-option (i.e., enter the file name as DUP.SYS/N so it won't autorun). From the listing we know the file runs from address $1F0C to $3305, has no init address and runs at $2075. Save it to disk with DOS Option K as DUP.SYS,1F0C,3305,,2075. Boot up the DOS again and you should hear the file load without the hesitations.

There are a lot of subtleties to Ultra Disassembler detailed elsewhere in this manual, but at this point you've basically seen it do its job. Now you can start to do yours, which is to apply the human intelligence to use this tool in analyzing, modifying and customizing programs to your own ends. If you're a beginner at assembly language programming, here's a good exercise to start with: modify DUP.SYS to ring the console bell (by 'displaying' the bell character $FD) to prompt you to make a keyboard entry when necessary.

CHAPTER 2
Design Philosophy

The ideal disassembler would be one that would exactly reproduce the source program that was assembled to produce a given machine language program. In this section we'll discuss the various obstacles that make this goal impossible to achieve, and the compromises we have to accept in a real-life disassembler.

Suppose we're using a simple disassembler (like the ones that most assemblers incorporate in their debugging packages) and we come across the following sequence of bytes:

```
46 69 6C 65 6E 61 6D 65 3F
```

This will disassemble as:

```
        LSR $69
        JMP ($6E65)
        ADC ($6D,X)
        ADC $3F
```

This may indeed be what the programmer wrote. But the following instruction will produce the SAME object code:

```
        .BYTE 'File name?'
```

You and I know which of these versions is most likely correct, but it's a pretty tough decision for a 48K Atari to make! The point?

A DISASSSEMBLER HAS NO WAY OF DISTINGUISHING TEXT STRINGS AND DATA TABLES FROM CPU INSTRUCTIONS.

Now suppose you're analyzing a program by following the logic flow. If a given section of the program is actually a series of CPU instructions misinterpreted as data, you will eventually get lost in it; but in the reverse situation, you'll never try to read part of the section as an instruction in the first place because the program flow simply won't lead you into it. This suggests a simple rule:

OBJECT CODE SHOULD BE INTERPRETED AS CPU INSTRUCTIONS WHENEVER POSSIBLE, AND INSERTED IN .BYTE PSEUDOS OTHERWISE.

Now let's unleash this hypothetical simple disassembler on a longer sequence of object code, beginning at location $4000:

```
A2 10 A9 7F 9D 44 03 A9 60 9D 45 03 A9 05 85 CF A9 80 9D 48 03 A9 00 9D 49
03 20 56 E4 10 03 4C 00 45 C6 CF 30 14 18 BD 44 03 69 80 9D 44 03 BD 45 03 69
00 9D 45 03 4C A9 00
```

The disassembly will look like this:

```
4000 A210       LDX #$10
4002 A97F       LDA #$7F
4004 9D4403     STA $0344,X
4007 A960       LDA #$60
4009 9D4503     STA $0345,X
400C A905       LDA #$05
400E 85CF       STA $CF
4010 A980       LDA #$80
4012 9D4803     STA $0348,X
4015 A900       LDA #$00
4017 9D4903     STA $0349,X
401A 2056E4     JSR $E456
401D 1003       BPL $4022
401F 4C0045     JMP $4500
4022 C6CF       DEC $CF
4024 3014       BMI $403A
4026 18         CLC
4027 BD4403     LDA $0344,X
402A 6980       ADC #$80
402C 9D4403     STA $0344,X
402F BD4503     LDA $0345,X
4032 6900       ADC #$00
4034 9D4503     STA $0345,X
4037 4C1040     JMP $4010
403A A900       LDA #$00
.... ....
```

If we apply some knowledge of the Atari Operating System to this code segment, we see that it's a repeating call to the CIO utility. It reads 128 bytes via IOCB 1 into a buffer at location $607F; advances the buffer start pointer by 128 bytes; and repeats the IOCB read for a total of 5 times, then goes to some other code at $403A.

Notice that this program uses location $CF as the loop counter. Let's say we decide that this segment doesn't get executed often enough to merit dedicating a Page 0 address to the counter, so we want to use location $0600. We can change the address in the instructions at $400E and $4022 and reassemble the program - but it won't work. We've changed two two-byte instructions to three-byte instructions, which moved all the subsequent instructions up in memory and made a mess of the various branches and jumps. Which brings us to another rule:

THE DISASSEMBLER SHOULD REPRESENT ALL ADDRESS REFERENCES AS LABELS, AND ATTACH THE LABELS TO THEIR RESPECTIVE INSTRUCTIONS.

Ok, so we let the disassembler attach the required labels. Now we have a piece of 'source' code that we can modify and reassemble successfully, but it doesn't read as clearly as the original probably did. The programmer's comments are forever lost to us, but it would be nice to at least know the names of all those operating system locations. Why give them arbitrary labels when we can use the documented system labels? Hence another rule:

THE DISASSEMBLER SHOULD ATTACH STANDARD OS LABELS TO ALL REFERENCED OS LOCATIONS.

We're just about there, but now we need a master rule to resolve any ambiguities (something like Asimov's First Law of Robotics):

THE OUTPUT OF THE DISASSEMBLER, WHEN REASSEMBLED, SHOULD PRODUCE AN EXACT DUPLICATE OF THE ORIGINAL OBJECT CODE.

I have attempted to design Ultra Disassembler with these rules in mind. Its overall logic flow is detailed in the next section.

CHAPTER 3
Program Logic

Ultra Disassembler accepts object code from any of three sources:

* Binary DOS 2.0S load files
* A specified list of disk sectors (for direct-boot programs)
* A specified section of memory

Regardless of code source, the output assembly language program is written to a disk file under a user-specified file name. Output file length is limited to $4000 bytes. The output file is initially opened with an extender of '.000.' When it reaches maximum length, it is closed and a new file is opened with the extender incremented by 1, and so on, for as many files as necessary. A maximum of 4 files are written to one disk; at the end of the fourth file, the user is prompted to insert a fresh disk. Considering DOS files on the disk, this leaves 98 vacant sectors to allow for some in-place editing. Each file ends with a link instruction to the next file, except for the last file which ends with an assembly-end instruction.

Input code is stored in a buffer using most of the space between the top of the Disassembler and the bottom of display memory (except when the source is a memory area). If the program is too large for the buffer, it is disassembled in segments.

Ultra Disassembler operates in two passes. On the first pass, it takes the following actions:

(1) Examines every identifiable CPU instruction in the program which contains an address reference. Whenever a documented system location on pages 0, 2, 3, $D0, $D2, or $E4 is encountered it is flagged in a table of system locations. When any other address is encountered it is stored in a label address table with a capacity of up to a maximum of 1023 labels.

(2) Outputs a series of equate pseudo-ops for all system locations flagged during the pass. Offsets are taken into account; e.g., location $0B is interpreted as DOSVEC+1 and the equate DOSVEC = $0A is output. The hardware locations on pages D0 and D2 are equated separately for read and write instructions since they have distinct labels (and effects) when read or written.

(3) Outputs a series of equate pseudo-ops for all nonsystem locations on page 0 for which labels were stored, to prevent assembly errors associated with forward references to page 0 locations.

Next the object code read routine is reinitialized and the second pass does the following:

(1) It disassembles every byte of the code. Identifiable instructions are output in assembly language; everything else is output in .BYTE pseudo-ops. Whenever a program location that is referenced by a previously-generated label (system or nonsystem) is encountered, the label is attached unless it occurs in the middle of a CPU instruction. Nonsystem locations are assigned labels consisting of an L (or Z for zero-page locations) followed by a four-digit number. Address references in instructions are specified by label until the label table is full. Subsequent references are given as absolute addresses. Absolute values of all references are given in appended comments. Every label attached to an instruction is flagged as an internal reference.

(2) It outputs a series of equate pseudo-ops for all external references (i.e., all labels which were referenced but were not attached to instructions above). In practice, most of these are 'phony' addresses resulting from data and text being interpreted as CPU instructions.

The baseline program design assumes that disassemblies will be reassembled with the Atari Macro Assembler Ver 1.0. With the aid of a utility supplied with Ultra Disassembler, the user can customize the program for use with any assembler which conforms to the standard 6502 assembly mnemonics, including the Atari Assembler/Editor Cartridge, Datasoft's DATASM-65, and Optimized Systems Software's EASMD.

THE ZERO-PAGE PROBLEM

All 6502 assemblers have a subtle problem in interpreting zero-page addresses defined by forward references. If you haven't run across it, try assembling the following program:

```
LABEL1 = $80
         LDA LABEL1
         LDA LABEL2
LABEL2 = $90
```

The object code you get is

```
A5 80
AD 90 00
```

Holy mackerel! The first instruction came out correctly, but the second one turned into an absolute-mode instruction with a two-byte address even though the referenced label was on page 0.

This looks like a bug, but it's really an accommodation to the nature of the 6502. On the first pass, the assembler counted its way through the bytes represented by the instructions. On the 'LDA LABEL1' instruction, it consulted its label table and found that LABEL1 is a page-0 address; therefore it allotted two bytes for this instruction. On the 'LDA LABEL2' instruction, it found that LABEL2 hadn't been defined yet, so it flagged this one as 'waiting to be defined', and pressed on to the next instruction. Unfortunately, it had to make some decision as to how many bytes to count for the instruction, so it took the conservative course and counted three. On the second pass, it looked in the label table and found that LABEL2 is also a zero-page address, and was stuck with a problem. If it generated the zero-page instruction 'A5 90' it would have used up two bytes where it left room for three, and every subsequent instruction would occur one byte early. This would of course invalidate all the labels that were evaluated by counting bytes. So, the assembler observed the following rule:

EVERY ADDRESS DEFINED BY A FORWARD REFERENCE
WILL BE INTERPRETED AS A TWO-BYTE ADDRESS IF POSSIBLE

It doesn't happen for the indirect indexed mode, for example, because this mode always references a zero-page address. Some assemblers (like DATASM-65) take a slightly different approach and simply refuse to permit forward page-0 references.

This presents a problem for the disassembler, because at the end of a disassembly it outputs a list of equate instructions for addresses that were referenced from the disassembled program, but did not occur as labels within it. Any zero-page locations equated in this list would produce absolute-mode instructions on reassembly and make a mess of all the subsequent labels. Ultra Disassembler defends against this by equating all the zero-page nonsystem labels at the BEGINNING of the disassembly, without waiting to see if they occur within the program. Most of the time, this causes no problems at all since these locations are usually external references — just as they are assumed to be. If a program has any code actually assembled on page 0, it will result in 'doubly defined label' errors. With most assemblers, this is a nonfatal error; if it does bomb the reassembly, it can be eliminated by simply deleting the offending equate from the zero-page equate instruction list.

There is a related problem that occurs when the object code contains an absolute-mode instruction which references a zero-page address. Suppose the Disassembler comes across the bytes AD 80 00. Now AD is the opcode for an absolute-mode LDA instruction, so this could be disassembled as LDA $0080. However, an assembler would interpret this as a zero-page instruction and produce A5 80, and everything that follows it would be one byte away from the right place. Ultra Disassembler has a special subroutine to prevent this — wherever it finds an apparent absolute-mode instruction which references a page-zero address, it assumes it's part of a data table (which it usually is) and outputs it as a .BYTE instruction. In a few cases the instruction is legitimate (it can result from a last-minute brute-force fix to a program, or be part of some sort of critical timing loop), but one way or the other, the disassembled code will reassemble to what you started with.

CHAPTER 4
Input Options

You can disassemble code from binary DOS files, specified disk sectors or machine memory. You will be asked which one to use as soon as you've selected the number of disk drives in use. Use these procedures for each code source:

(1) File input: You will be asked for an input filespec. Enter the full filespec INCLUDING THE DRIVE ID. For example:

```
 D:PROGNAME.OBJ
D1:BINFIL
D2:DOTEATER.BIN
```

The drive ID must of course reflect the drive that will contain the object program; it can be any drive number the Operating System will support. Ultra Disassembler will expect the file to conform to the specifications given in the Atari DOS II Reference Manual for binary load files, including compound files.

(2) Sector input: You can input a list of disk sectors to disassemble. These sectors will be read directly from the input disk without regard to file structure, forward file pointers, etc; this is the option you use to disassemble programs which boot directly without going through the File Manager. You can specify any number of sectors up to and including the entire disk.

Upon selecting the sector option, you will be asked to input a sector number or range. Do one of the following:

(a) To specify a single sector, type the sector number IN HEXADECIMAL and press RETURN.

(b) To specify a range of sectors, type the two sector numbers IN HEX, separated by a comma, and press RETURN.

Repeat this procedure as many times as necessary until you've input all the sectors you want disassembled. Then press RETURN without any entry to terminate the input specification. For example, suppose you want to disassemble sectors $1 through $3, $13 through $20, $30, $35 and $2BB. Input:

```
1 , 3 RETURN
1 3 , 2 0 RETURN
3 0 RETURN
3 5 RETURN
2 B B RETURN
RETURN
```

Next you'll be asked for a start address. Input the address at which the disassembly is to start, in hex (as usual) and press RETURN. Your input specification is now complete.

(3) Memory input: This feature lets you disassemble any code which is already present in memory. Since Ultra Disassembler is about 13K long and sits above the File Manager, this limits you to disassembling ROM cartridges, the Operating System and any small subroutines you load on Page 6 (which is otherwise untouched by Ultra Disassembler). The feature was included simply because it requires only a trivial amount of code over and above the rest of the program. If you select it, you'll be asked for a start and end address. Input the addresses IN HEXADECIMAL, separated by a comma, and press RETURN. For example, to disassemble a cartridge in the left slot (which you can only do with a cartridge that allows you access to DOS, such as BASIC or the Assembler/Editor), enter

A 0 0 0 , B F F F RETURN

and your input specification is complete.

CHAPTER 5
Output Options

Ultra Disassembler offers three forms of disassembled output:

* A pseudo-assembly listing to the screen at all times.

* A printer listing identical to the screen output, when selected.

* A pseudo source-code listing, in a format suitable for reassembly, written to disk when selected.

You can select these options as follows:

(1) Screen output: Occurs at all times. To take a close look at it, press the **START** button and it will freeze until you press **START** again.

(2) Printer output: Turn on your printer, bring it on line and press the **OPTION** button and the output going to the screen will also go to the printer. Toggle the printer off and on as often as you like with the **OPTION** button.

If you try to select the printer when you don't have one on-line, Ultra Disassembler will remind you by ringing the console bell and go on its merry way. If the printer goes off-line while a print output is in progress (by running out of paper, for example), Ultra Disassembler will stop and wait for you to rectify the problem. Bring the printer on-line and press **START** and the disassembly will resume.

(3) Disk output: This automatically occurs if you enter an output file name when asked for it. If you don't want to write to disk, just press **RETURN** when asked for the output file name and you will not be asked for an output disk.

CHAPTER 6

Reassembling Files

Under certain conditions, a disassembly file will reassemble without any modifications at all, but keep in mind the following special considerations:

ASSEMBLERS OTHER THAN AMAC: DATASM-65 and EASMD differ from AMAC in their specifications for the file linking instructions; DATASM uses 'FILE 'filespec'' instead of 'LINK filespec', and EASMD uses '.INCLUDE #filespec' with no nesting permitted. The Atari Assembler/Editor cannot link files.

DATASM-65, EASMD and the Atari Assembler/Editor use '* =' instead of 'ORG', and do not append a run address contained in the .END statement.

EASMD and the Atari Assembler/Editor Cartridge expect line numbers.

These differences can be accommodated by creating a customizing file with the utility provided; see the section on 'Using the Customizer'. The various flags and buffers which control these items can also be set in the program file itself with a sector editor; see the section on 'Altering the Disassembler'.

LONG PROGRAMS: Disassemblies written to disk are automatically broken into subfiles for convenience in editing; these files are nominally terminated by link instructions. It is vital to note that the filespec given in each of these instructions contains the same disk drive ID as the drive on which the disassembly files are written. If the disassembly occupies more than one disk, some of the LINK instructions will need to be altered to reflect the drive that contains the referenced file at assembly time. If you want to use multiple source disks with a single drive, you can do so with AMAC only by selecting the H=0 option (i.e., not writing any object code to disk).

CHAPTER 7
Using The Customizer

All 6502 assemblers use the same mnemonics for actual CPU instructions, but they tend to vary in their specifications for the various pseudo-operations. Also, your particular application of Ultra Disassembler may call for changing some of the options provided in the program. All of these items can be altered by changing various flags and buffers in the program file itself with a sector editor, but for most routine uses it's more convenient to use the customizing program provided on the distribution disk. Here's how to use it:

(1) Boot up DOS 2.0S with the Atari BASIC cartridge in the computer. Insert the distribution disk in the drive and enter RUN"D:CUSTOM.BAS" RETURN. The Customizer will load into the machine and display a menu. Items A through D on the menu allow you to set the text content of the major pseudo-ops, and items E through H allow you to alter the various Ultra Disassembler processing options. When you select an option, the screen will display the default state of the item and ask you for a new entry. Type your entry BEGINNING AT THE EXISTING CURSOR LOCATION and press RETURN. Now the new state of the option will be displayed so you can be sure you got it right. When you're satisfied, press RETURN without any entry and you'll return to the menu. The items you can alter are:

A. File link pseudo: This is the instruction that tells the assembler to find another source file on the disk and continue the assembly with that file. If your assembler requires single quotes around the filespec, simply end the entry with a single quote and Ultra Disassembler will supply the closing single quote. One embedded blank is permitted in the entry. Examples:

LINK produces LINK Dn:filename
.FILE ' produces .FILE 'Dn:filename'
INCLUDE # produces INCLUDE #Dn:filename

B. Origin pseudo: This sets the assembler's location counter. For almost all assemblers other than AMAC, set it to *=.

C. Assembly-end pseudo: This marks the logical end of the source file for the assembler. Assemblers other than AMAC use either END or .END.

D. Define-byte pseudo: This pseudo enters a list of specified bytes in the object file without processing. The default is .BYTE, which is compatible with almost all assemblers; AMAC can also use DB.

E. Comment insertion flag: This flag enables or disables the insertion of comments in the Ultra Disassembler output file to indicate absolute values of labels appearing in the disassembled instructions. Disabling the comments will reduce the output disk file volume by about 30%, which may be critical if you want to reassemble files with a single disk drive.

F. Output file size.

G. Number of files per disk: These options let you define the maximum length of the output files and the number of files that will be written to an output disk before a fresh one is requested. If you alter these values, it is up to you to insure that you don't try to write too much to one disk; filling a disk will crash Ultra Disassembler with a fatal I/O error.

H. Line number flag: This option will cause Ultra Disassembler to attach line numbers to the output instructions for assemblers that expect them. Lines will be numbered in increments of 10 beginning at 00010.

(2) When you've set all the options you want, place your WORKING COPY of Ultra Disassembler disk in Drive 1 (with no write-protect on it) and select menu item I (Write Customizing File). A one-sector-long file called DISKDIS.CUS will be written on the disk. Whenever you load Ultra Disassembler, it will search Drive 1 for this file. If it's present, Ultra Disassembler will read the customizing options from it and alter itself IN RAM ONLY. The program file on the disk will NOT be altered.

If you use more than one assembler, make up one working copy of the Ultra Disassembler disk (with appropriate customizing file) for each one. Alternatively, you can make permanently customized versions of the program file, as described in the next section.

ALTERING ULTRA DISASSEMBLER

If you prefer to make customizing changes in Ultra Disassembler on a permanent basis, you can do so easily with a sector editor such as Adventure International's DISKEY. Dump the last few sectors of the program file in ATASCII, and you'll find the various pseudo-ops and option flags set off by text comments and asterisks. For example, after the entry 'Link file pseudo ***' you'll find the skeleton of the file-linking instruction followed by three more asterisks. If you alter any of the pseudo-op buffers, be sure to begin the text in the same byte as the original, since this will determine the spacing in the output file. The line number and comment flags are OFF if the byte value is zero (ATASCII heart) and ON for any other value. The maximum output file size is given in hex, and the number of files per disk is specified as one more than the desired value (i.e. a 5 will cause 4 files to be written).

> WARNING: Directly altering a file in this manner offers plenty of opportunity to get in trouble. You should know EXACTLY what you're doing, or use the Customizer instead. In any case, maintain an unaltered backup copy of the original disk.

CHAPTER 8
Modifying Programs

Ultra Disassembler is designed to produce a pseudo-source file which will reassemble to an exact duplicate of the original object code if it isn't altered before the reassembly. Unfortunately, altering the code before reassembly can produce some special problems. For example, consider this fragment of disassembler output:

```
0800            LDA #$0F
0802            STA Z0010 ;$90
0804            LDA #$08
0806            STA Z0020 ;$91
0808            LDY #0
080A            LDA (Z0020),Y
080C            JSR L0200
080F L0100      .BYTE $FD
```

This sequence loads the address $080F into a zero-page pointer and then accesses that address with an LDA (),Y instruction. This loads the accumulator with the byte at $080F, which is $FD (the console bell character). Now suppose you alter some previous part of the pseudo-source program so that this segment begins at $0801 instead of $0800. The label L0100 will now have the value $0810, but the LDA-immediate instructions will not be altered; now the zero-page pointer will point to the last byte of the JSR instruction, which is the high byte of label L0200. If you've made a lot of alterations, you won't have the foggiest idea what this is, and of course, the reassembled program won't work.

In this specific case, you could have solved the problem by changing the LDA-immediate instructions to:

```
LDA #LOW L0100
. . . .
LDA #HIGH L0100
```

This is almost surely what the programmer put in the source code. In fact, you will probably find that 90% of the reassembly problems you encounter stem from immediate-mode load instructions. It's a good idea to check out all the immediate-mode instructions in your disassembled program (you can use the string-search mode of your program text editor to look for the # character) for this kind of situation; however the point of this example is to show that Ultra Disassembler has its limits. If you want to modify a complex program, you'll have to apply enough understanding of assembly language and the Operating System to cope with the unexpected!

NOTICE

COPYRIGHT